“Cloud Computing” Notes

Prof. Emiliano Casalicchio. Notes written by Alessio Lucciola during the a.y. 2022/2023.

1. Introduction

The first idea of **utility computing** was introduced in 1961, but the first implementation was only made in 2005/2006 thanks to Jeff Bezos (Amazon) that let users rent data storage and computer server time from Amazon like a utility. Despite the original ideas, today computing services are readily available on demand, just as other utility services: for this reason, the term utility computing was replaced with cloud computing. Users (consumers) pay providers only when they access the services, there are no investments or difficulties in building and maintaining complex IT infrastructure and the services are based on their requirements without regard to where the services are hosted.
Cloud computing is the product of a set of enabling technologies:

1. **Virtualization**: Virtualization is a collection of solutions allowing the abstraction of some of the fundamental elements for computing. We can have **Hardware Virtualization** (compute, storage, and network) that allows the coexistence of different software stacks (contained in a Virtual Machine instance) on top of the same hardware or **Application Virtualization** that allows isolating the execution of applications and providing a better control on the resource they access (e.g. Docker container).
2. **Web 2.0**: Web 2.0 is a rich platform for application development. Applications can be “synthesized” simply by composing existing services and integrating them. The difference with Web 1.0 is that pages were static/dynamic so there were no web applications.
3. **Service Oriented Computing (SOC)**: Service orientation is the core reference model for cloud computing. A service is an abstraction representing a self-describing and platform-agnostic component that can perform any function. It is supposed to be **loosely coupled** (to make it reusable), **programming language independent** (to increase service accessibility), and **location transparent** (to consume the service independently from the location). So basically the idea is to develop a specific component that performs a certain task and build it in a way that it is executable in each platform. They should also be able to interact with each other through specific network protocols. The two generations of SOC were **SOA** (service oriented architecture) and **Microservices**.

There are 3 **business drivers** for cloud computing:

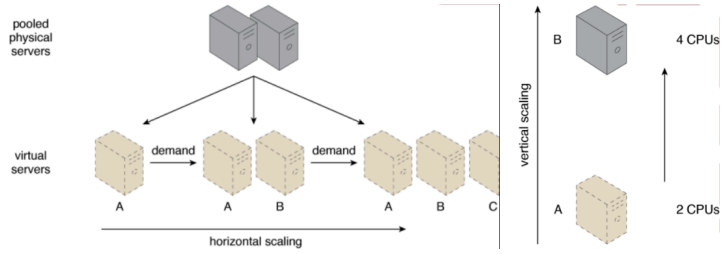
- **Capacity planning**: Capacity planning is related to providing the right amount of capacity when needed and this is done in order to avoid overprovisioning or underprovisioning. In order to do so, we should know in advance what is the workload and the total capacity of the system and finally use some mathematical approach to understand what is the right amount of resources to allocate. There are 3 different strategies:
  - Lag Strategy (reactive): Adding capacity when the IT resource reaches its full capacity;
  - Lead Strategy (proactive): Adding capacity to an IT resource in anticipation of demand;
  - Match Strategy (proactive): Adding IT resource capacity in small increments, as demand increases.
- **Cost reduction**: A company should expand its IT capacity to cope with the increasing workload demand but, of course, companies should reduce the costs as much as possible. Also, think of the case in which the company needs computational power only for a short period: in this case on-premise solutions could be really inconvenient. Usually, cloud-oriented solutions are cheaper than on-premise solutions.

- **Organizational agility**: Organizational agility is the measure of an organization's responsiveness to change. The time needed to respond to a change should be in hours or a few days.

Cloud computing, an **Internet-centric way** of computing. The Internet plays a fundamental role in cloud computing Cloud computing services are delivered and made accessible through the internet Now, let's give a proper **definition of cloud computing** (by NIST):
*Cloud computing is a model for enabling ubiquitous (location independent), convenient, **on-demand network access** to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*

This cloud model is composed of 5 essential characteristics:

- **On-demand self-service**: It allows the user to **fulfill the service demand**. A consumer can achieve computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider. Usually a pay-per-use strategy (the user pays based on how and when he wants to use the service) is used;
- **Broad network access**: Capabilities are available over the network and can be accessed through different devices. It makes it possible to access online storage, rent virtual hardware, or use development platforms and pay only for their effective usage, with no or minimal up-front costs.
  Note: In order to connect to a cloud service, the request made by an external user has to pass through different nodes usually owned by the ISP. There are **problems with performance** (some links could be particularly slow) **and security** (data could be sniffed). So problems happen usually in the network and the edge, rarely in the cloud services.
- **Resource polling**: The provider computing resources are pooled to serve multiple consumers using a **multi-tenant model**, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. Cloud computing is **location independent**: we can usually choose the location in which we put the data (e.g. country, state) but this is not always guaranteed so we have a really low control or no control at all. Multi-tenancy enables an instance of a program to serve different consumers, whereby each is isolated from the other;
- **Rapid elasticity**: Capabilities can be elastically provisioned and released, in some cases automatically (auto-scaling and elastic load balancing), to scale rapidly outward and inward commensurate with demand. Elasticity is related to the concept of **Scalability** and it can be:
  - Horizontal Scaling: Scaling out and scaling in;
  - Vertical Scaling: Scaling up and scaling down.

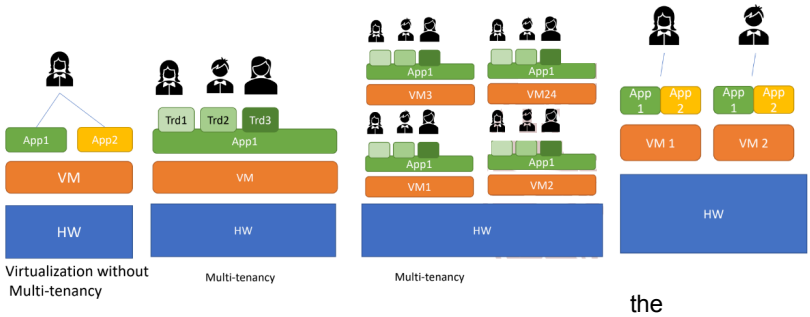


- **Measured services**: Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service. In general, also consumers should monitor the usage of their resources in order to control the costs (e.g. with alerts), scale the service properly (e.g. autoscaling thresholds), guarantee specific SLA (Service-Level Agreement) (e.g. throughout, response time) and increase **resiliency** (the ability to provide and maintain an acceptable level of service in the face of faults and challenges to normal operation).

Note: There are some differences between performance and scalability. Performance measures how fast and efficiently a system can complete certain tasks while scalability measures the trend of performance with increasing load.

So far we talked about **multi-tenancy** where multiple users (tenants) access the same application logic simultaneously. Each tenant has its own view of the application as a dedicated instance and is unaware of other tenants. Multi-tenant applications isolate data and configuration information and this can be done with virtualization and distributed software technologies. There are some **differences between multi-tenancy and virtualization**:

- **Virtualization**: Multiple virtual copies of the server environment (OS + applications) can be hosted by a single physical server and it is the physical resource that is shared.
- **Multi-tenancy**: An application is designed to allow usage by multiple different users where each user feels as though they have exclusive usage of the application. The application can run on one or more VM or physical servers. To address scalability multiple instances can be used (see the third example). Multi-tenancy can also be used to refer to the fourth scenario because Host OS/hypervisor (that is located between the HW and the VMs) is shared by VMs (the tenants). Each VM has the illusion to be the only user of the host OS, and hence the HW Tenants (VMs) are isolated.



the

SUM. Cloud computing is a phenomenon touching on the entire stack: from the underlying hardware to the high-level software services and applications. It introduces the concept of everything as a service, mostly referred to as **XaaS**. Another important aspect of cloud computing is its **utility-oriented** approach. More than any other trend in distributed computing, cloud computing focuses on delivering services with a given pricing model, in most cases a pay-per-use strategy. Even though many cloud computing services are freely available for single users, enterprise-class services are delivered according to a specific pricing scheme. In this case users subscribe to the service and establish with the service provider a Service-Level Agreement (SLA), denying the quality-of-service parameters under which the service is delivered.

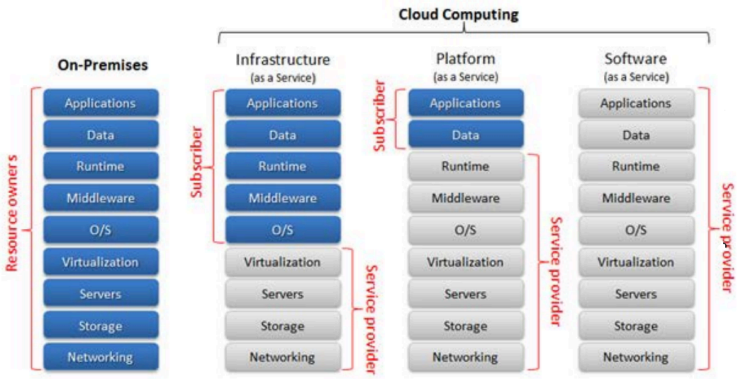
There are some ways to understand if a service is a cloud service that can be summarized in 4 criteria:

- The service is accessible via a web browser or a web service application programming interface (API);
- Zero capital expenditure is necessary to get started;
- ❗ You pay only for what you use as you use it;
- ❗ You should have the illusion of having **infinite resources**.

There is a worksheet that one can compile to better understand if a service is a cloud one[1].

Some Cloud computing paradigms are the **Service Models**:

- **Software as a Service (SaaS)**. The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user specific application configuration settings;



- **Platform as a Service (PaaS)**. The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications developed using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment;

[1] See Paper "NIST.SP.500-322"

- **Infrastructure as a Service (IaaS)**. The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).

So, the differences are mainly linked to the separation of responsibilities of the cloud consumer and the cloud (service) provider. Moving from IaaS to SaaS the control of the consumer on the resources decreases, the expertise needed to manage the cloud service decreases and, of course, the ease of use increases. We can also combine cloud service models, noticing that there is a hierarchy: IaaS + PaaS, IaaS + SaaS, IaaS + PaaS + SaaS.

There are also some **Deployment Models** that controls who can access the cloud services:

- Private cloud: The cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers (e.g., business units). It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises;
- Community cloud: The cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on or off premises.
- Public cloud: The cloud infrastructure is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider.
- Hybrid cloud: The cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load balancing between clouds).

There are **five major actors** in the cloud computing environment:

- Cloud Consumer: A person or organization that uses services provided by Cloud Providers;
- Cloud Provider: An entity responsible for making a service available to consumers;
- Cloud Auditor: The intermediary that provides connectivity between providers and consumers;
- Cloud Broker: An entity that manages the use, performance and delivery of cloud services, and
- negotiates relationships between Cloud Providers and Cloud Consumers;
- Cloud Carrier: An intermediary that provides connectivity and transport of cloud services from Cloud Providers to Cloud Consumers.
- Cloud Administrator: Person/organization responsible for administering a cloud-based IT resource;
- Cloud Service Owner: Person/organization that legally owns a cloud service. It can be the cloud consumer, or the cloud provider. A cloud consumer that owns a cloud service hosted by a third-party cloud does not necessarily need to be the end-user (or consumer) of the cloud service.

The **risks and challenges of Cloud computing** are:

- Increased security vulnerabilities: There are overlapping trust boundaries (access grants) to resources because they are commonly shared among users and can be accessed from providers. This increases the exposure of data to potential attackers;
- Reduced operational governance control: Cloud consumers are usually allotted a level of governance control that is lower than that over on-premise IT resources;
- Limited portability between cloud providers: There aren't standards in the CC industry, thus it's challenging moving from one provider to another one;
- Legal issues: Cloud consumers will often not be aware of the physical location of their resources, and this can raise legal concerns.

2. Enabling technologies

2.1 Principles of Distributed Computing

When talking about **distributed systems**, we mean a system in which the computation is broken down into units executed concurrently on different computing nodes. The fact that the system is distributed often implies that:

- The computing nodes are in **different locations**;
- There is **heterogeneity** in terms of hardware and software features.

Examples of distributed computations are the MapReduce paradigm in which a simple word count can be distributed (in case the data is large), the data storage and retrieval can be distributed as well, the internet and the P2P file sharing and so on and so forth.

Two **definitions of distributed system** are:

- A distributed system is a collection of independent computers that appears to its users as a **single coherent system** (Focus on unified usage and aggregation of distributed resources);
- A distributed system is one in which components **located at networked computers** communicate and coordinate their actions only by **passing messages** (Focus on the networked nature and communication among processes by passing messages);

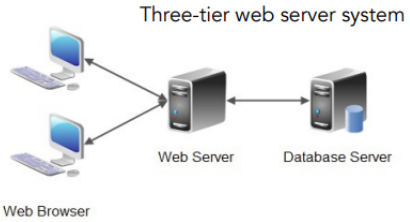
In a distributed system the components must communicate and coordinate their actions only by exchanging messages. A distributed system is the result of **components** that traverse the entire computing stack, from hardware to software. At the **bottom layer** we have the physical infrastructure (the hardware) and its components that are directly managed by the OS, that provides the basic services for **Inter-Process Communication (IPC)** so the process scheduling and the task and resource management for each node. These 2 layers become the platform on top of which software is deployed to turn a set of computers into a distributed system. The **middleware layer** leverages such services to build a uniform environment for the development and deployment of distributed applications, completely independent from the underlying operating system and hiding all the heterogeneities of the bottom layers. The **top layer** of the distributed system stack is represented by the applications and services designed and developed to use the middleware.

There are few differences between the definition of Distributed System and Cloud Computing. In general, **cloud computing is a specialized form of distributed computing** that introduces utilization models for remotely provisioning scalable and measured resources so we have the low layer that can be represented by the IaaS service model, the middleware by the PaaS and the top layer by the SaaS.

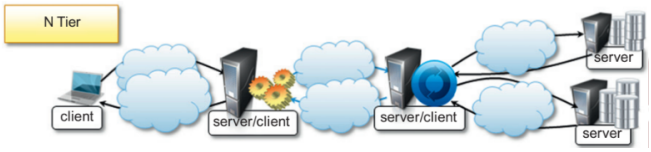
2.2 System Architectural Styles

The **system architectural styles** cover the physical organization of components and processes over a distributed infrastructure. There is a set of reference models to describe the major advantages and drawbacks of a given deployment and if it is applicable to a specific class of applications. There are two reference styles:

- **Client/Server**: These two components (client and server) interact with each other through a network connection using a given protocol (usually HTTP, TLS or FTP). The communication is **unidirectional**: The client issues a request to the server, and after processing the request, the server returns a response. The client/server model is suitable in **many-to-one scenarios**, where the information and the services of interest can be centralized and accessed through a single access point that is the server. In general, multiple clients want to connect to a certain service and the server must be able to efficiently serve requests coming from different clients. There are 3 main components that are the presentation (basically the interface through which it is possible to build the request to send to the server), the application logic (the logic that allows the server to interpret the requests and create the responses) and the data storage. Presentation, application logic, and data maintenance can be seen as conceptual layers,



which are more appropriately called **tiers**. The classical model is composed from two tiers (the client and the server) that is suitable for systems with limited size but it suffers from scalability issues. We may have three tiers (client, server/client, server - basically the server can act like a client and send a request to another server where the second server can be, for example, a database) that is more scalable or we can also build a N tier system.

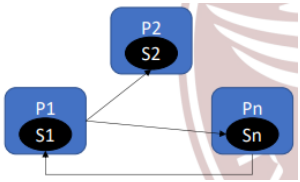


- **Peer-to-peer**: Introduces a symmetric architecture in which all the components, called peers, play the same role and can be both client and server. More precisely, each peer acts as a server when it processes requests from other peers and as a client when it issues requests to other peers. Therefore, this model is quite suitable for highly decentralized architecture, which can scale better along the dimension of the number of peers (if there are many peers, we reduce the overload that a server would have if everybody would make it a request). The disadvantage of this approach is that the management of the implementation of algorithms is more complex than in the client/server model.
  EX. The blockchain in which each node decides the new block to be added in the chain.
  EX. Distributed datastore in which each node has the same role: It keeps a portion of the dataset and synchronizes with peers that have the same data. It participates in the consensus algorithm that validates transactions (read and write).

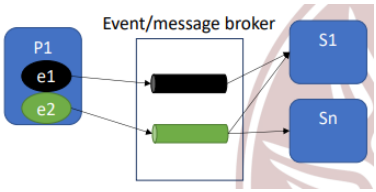
2.3 Software Architectural Styles

Software architectural styles are based on the **logical organization of software components**. They are helpful because they provide an intuitive view of the whole system, despite its physical organization. They define the **different roles of components in the distributed system** and how the **components are distributed across multiple machines**. There are different ways to organize an application and one of them is the **independent components**. Components have their own life cycles and interact with each other to perform their activities, so they receive a request and perform a certain action. There are different ways to implement these components:

- **Communicating processes**: Components are represented by independent processes that use IPC (inter-process communication) such as Web Services or REST to communicate and coordinate each other. Each **process provides services** and can **use services exposed by the other processes**. This is an abstraction that is suitable for modeling distributed systems that, being distributed over a network of computing nodes, are necessarily composed of several concurrent processes. The conceptual organization of these processes and the way in which the communication happens vary according to the specific model used, either peer-to-peer or client/server;



- **Event-based systems**: Components of the system are loosely coupled and connected. In addition to exposing operations for data and state manipulation, each component also **publishes** (or announces) **a collection of events** with which other components can subscribe. So, a process can generate an event and publish it to the **message broker**. Other processes can subscribe to that specific event and they will be executed when the event is activated. The main advantage of such an architectural style is that it promotes the development of open systems: new modules can be added and easily integrated into the system as long as they are able to register to the events.



With a microservice architecture we can implement both systems that use the communicating processes model or the event-based one.

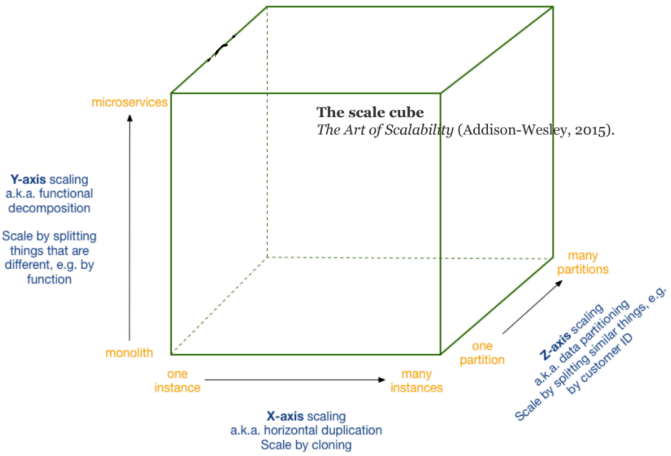
2.4 Microservices

A **Microservice** is an **architectural style** that structures an application as a collection of services that are:

- **Highly maintainable and testable**: The service usually provides a single functionality and the code used to implement that specific service is small and so, easily maintainable;

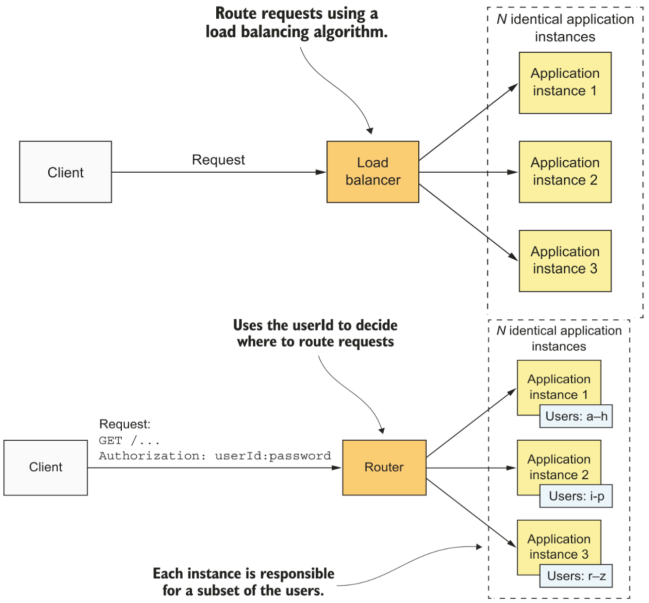
- **Loosely coupled**: There should be no dependencies between the different components. Everyone should be able to perform a certain activity without the help of other services;
- **Independently deployable**: The deployments of the components should be independent;
- **Organized around business capabilities**: Services built based on the business capabilities identified;
- **Owned by a small team**: It is possible to structure the engineering organization as a collection of small teams. Each team is solely responsible for the development and deployment of one or more related services.

In general, a microservice architecture enables the **rapid, frequent and reliable delivery of large, complex applications** and also enables an organization to **evolve its technology stack**. The **high-level definition of microservice architecture** (microservices) is an architectural style that functionally decomposes an application into a set of services.

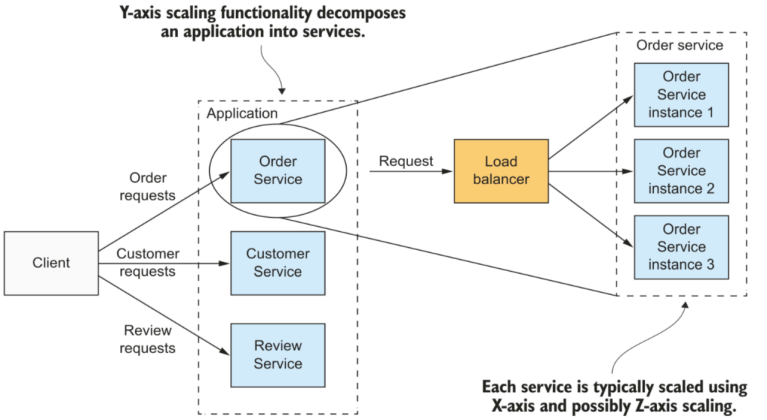


One important property of microservices is the **scalability** that can be described by the so-called **AKF scale cube**. Imagine a cube drawn with the aid of a three-dimensional axis. The initial point, with coordinates of (0, 0, 0), is the point of **least scalability** within any system. It consists of a **single monolithic** solution deployed on a single server. It might scale up with larger and faster hardware, but it will not scale out. In this way, the system is bound by how fast the server runs the application in question and how well the application is tuned to use the server.

The **X-axis** is very useful and easy to implement. X-axis scaling load balances requests across multiple instances. X-axis scaling is a common way to scale a monolithic application. It is possible to run multiple instances of the application behind a **load balancer**. The load balancer distributes requests among the N identical instances of the application. This is a great way of improving the capacity and availability of an application.



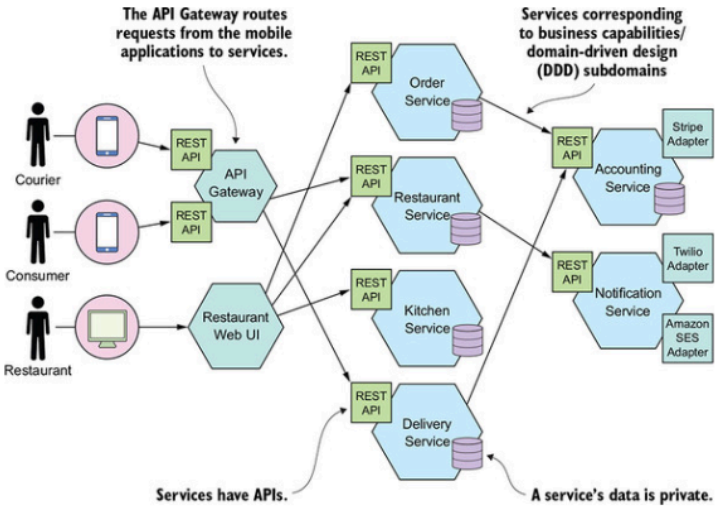
The **Z-axis** scaling routes requests based on an attribute of the request. Z-axis scaling also runs multiple instances of the monolith application, but unlike X-axis scaling, each instance is responsible for only a subset of the data. The **router** in front of the instances uses a request attribute to route it to the appropriate instance. An application might, for example, route requests using *userId*. Z-axis scaling is a great way to scale an application to handle increasing transaction and data volumes. X- and Z-axis scaling improve the application's capacity and availability but **neither approach solves the problem of increasing development and application complexity**. To solve those, you need to apply **Y-axis** scaling, or functional decomposition in which a monolithic application is splitted into a set of services. Each service is responsible for a particular function and it is scaled using X-axis scaling and, possibly, Z-axis scaling.



So, for what concerns the **monolithic architecture**, is a singular, large computing network with one code base that couples all of the business functionalities together. To make a change to this sort of application requires updating

the entire stack by accessing the code base and building and deploying an updated version of the service-side interface. It is simple to develop and it is quite easy to make radical changes, it is straightforward to test and deploy and it is easy to scale but when the complexity increases (and technologies evolve) development, testing, deployment and scaling become much more difficult. Moreover,it isn't possible to scale individual components, if there's an error in any module, it could affect the entire application's availability and a small change to a monolithic application requires the redeployment of the entire monolith.

For what concerns the **structure of microservices**: The frontend services include an API gateway and a Web UI. The API gateway, which plays the role of a facade, provides the REST APIs that are used by the consumers' and couriers' mobile applications. The Web UI implements the web interface that's used by an entity (a user) to interact with the application. The application's business logic consists of numerous backend services. Each backend service has a REST API and its own private datastore. Each one can be independently developed, tested, deployed, and scaled and this increases **modularity**. There might also be external services with adapters used to establish the connection.



So microservices deliver all sorts of benefits in terms of maintainability, scalability and so on, in particular:

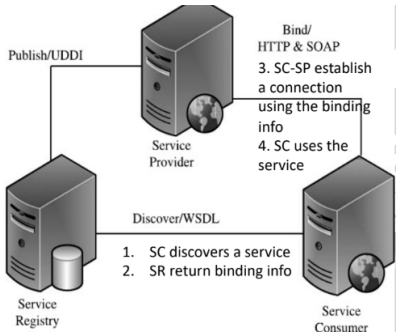
- It enables the continuous delivery and deployment of large, complex applications;
- Services are small and easily maintained, and can be independently deployable and scalable;
- Microservices naturally lend themselves to containerisation;
- The microservice architecture enables teams to be autonomous: They can focus on small functionalities assigned to them;
- Better fault isolation: If there is a fault in a service, it is easier to identify it and, most of all, it doesn't compromise the whole system but the single functionality:

On the contrary, **finding the right set of services is challenging** (what is the right amount of services that optimize our architecture) also because there isn't a concrete, well-defined algorithm for decomposing a system into services. Distributed systems are complex and so development, testing and deployment is challenging and may require several skills. Moreover, we may have features that span multiple services and this requires careful coordination (among development teams). In general, it is also difficult to choose when to use a microservice architecture (instead of a monolithic one): in general, in the long term a monolithic architecture is not a good choice so it would be better to opt for microservices.

2.5 Service Oriented Architecture (SOA)

Before the microservice architecture there was another architecture that is the **Service Oriented Architecture (SOA)**. SOA organizes a software system into a collection of interacting services. A service encapsulates a software component and it has four main characteristics:

- **Boundaries are explicit**: Interfaces to access the service are well defined and kept minimal to encourage reusability and simplify interaction;
- **Services are autonomous**: A service can be integrated in different systems at the same time;
- **Services share schema and contracts**: A contract describes the structure of messages the service can send and/or receive (APIs of the service), so how to interact with the service;



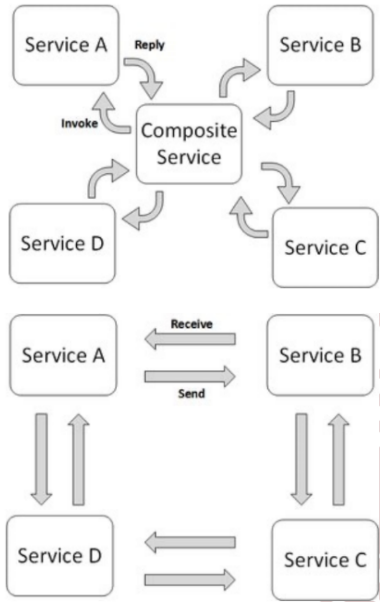
- **Service (semantic) compatibility is determined based on policies**: Policies define the capabilities and requirements of the service so how to interact with it and what are the expected results.

There are two major roles (and three main components) within SOA:

- **Service Registry**: It stores the service metadata.
- **Service Provider**: It is the maintainer of the service that publishes the service in a registry (the service registry) together with a service schema and a contract.
- **Service Consumer:** It locates the service metadata in the registry and develops the required client components to bind and finally use the service.

Service providers and consumers can belong to different organization bodies or business domains. Components can play the roles of both service provider and service consumer and **services can be aggregated** in workflows to satisfy a specific task (e.g. search for a flight, book hotel, etc..). There are some ways to coordinate services:

- **Service Orchestration**: Service orchestration represents a single centralized process (the orchestrator) that coordinates the interaction among different services. The orchestrator is responsible for invoking and combining the services when needed.
  EX. Travel planner: Select destination, display hotels, select hotel, suggest options (likerent car, buy attractions, etc..).
- **Service Choreography**: Service Choreography is a decentralized approach (there is **no central authority**), which is defined by exchange of messages between two or more services.
  EX. Online payment with credit card to conclude an online purchase. Seller Transaction Service sends a CC transaction request to the Credit Card company Service. Credit Card company service accepts/denies and sends the answer to the Seller Transaction Service.

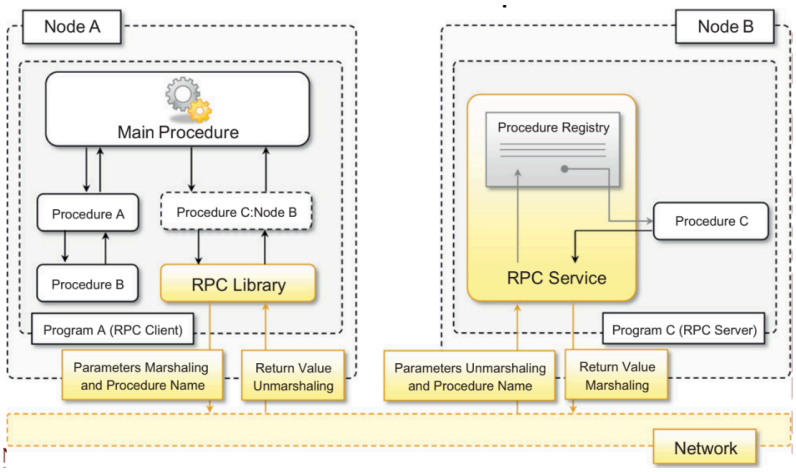


There are some differences with microservices:

- **Communication and interoperability**: In a microservices architecture each service is developed independently and usually uses lightweight technologies such as REST. On the other hand, each service in SOA must share a common Enterprise Service Bus (ESB) that is a middleware to make services communicate that represents a single point of failure, a slow communication between them and it is difficult to maintain. Besides, SOA uses heavyweight technologies such as SOAP;
- **Data handling**: SOA architecture includes a single database for all services within an application, microservices instead have dedicated servers or databases for each service;
- **Service size**: Microservices architectures are made up of highly specialized services, each of which is designed to do one thing. SOA is typically used to integrate large, complex monolithic applications where an application usually consists of a few large services.

2.6 Models for Inter-Process Communication

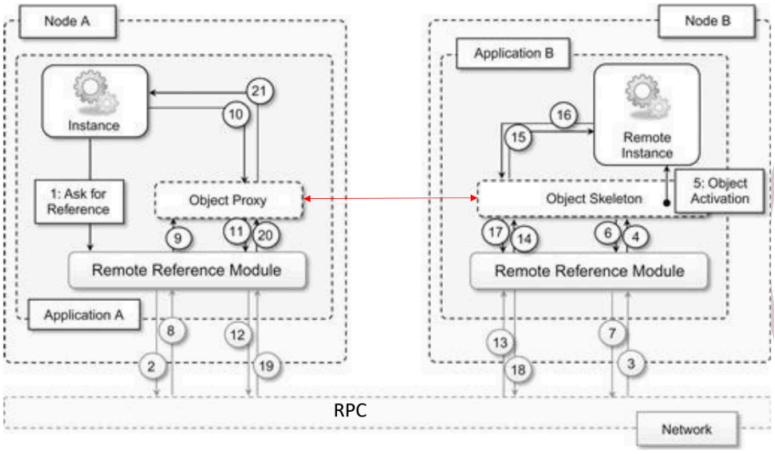
Inter-Process Communication (IPC) is what connects different components of a distributed system, making it act as a single system. There are several ways to make components interact with each other and they correspond to different IPC abstractions. IPC is realized through fundamental tools of network programming (e.g., sockets). By means of IPC, processes exchange messages where a message can be considered any discrete amount of information that is passed from one entity to another. There are



different ways to make IPC that are **Remote Process Call (RPC)**, **Distributed Objects** and **Web Services**.

Remote Process Call (RPC) is a paradigm that extends the concept of procedure call beyond the boundaries of a single process, thus triggering the execution of code in remote processes. In this case, an underlying **client/server architecture is implied**: the client is what calls the remote procedure and the server is the remote entity that hosts the procedure, executes the code and returns the result of the computation. In particular, it works like this:
The server process (node B) maintains a **registry** (**Procedure Registry**) of all the available procedures that can be remotely invoked and listens for requests from clients that specify which procedure to invoke, together with the values of the parameters required by the procedure. The client (node A) represents the node that is executing the main procedure: This procedure makes calls to procedures A and B that are executed locally. It also wants to call another process C that is not stored locally. It uses the **RPC Library**, that is responsible for making a call and understanding which is the node where the procedure C is running (that is node B). Then, node A can establish a connection with node B and make a call to procedure C. An important aspect of RPC is **marshaling**, which identifies the process of converting parameters and return values into a form that is more suitable to be transported over a network through a sequence of bytes. The term **unmarshaling** refers to the opposite procedure (so all the parameters sent by node A are marshaled in order to be sent through the network and are unmarshaled as soon as they arrive at their destination. The same for the response, so the data generated by node B).
RPC maintains the synchronous pattern that is natural in IPC and function calls. Therefore, the calling process thread remains blocked until the procedure on the server process has completed its execution and the result (if any) is returned to the client. So, RPC is responsible for the marshaling/unmarshaling operation and the handling of the request-reply interaction between the client and the server. The developer must design and develop the procedures, register them to the RPC server (in order to make them available) and design and implement the client code to invoke the remote procedures.

The **Distributed Objects** are an **implementation of the RPC model** for the **object-oriented paradigm**. Distributed object frameworks use the mechanism introduced with RPC and extend it to enable the remote invocation of object methods and to keep track of references to objects made available through a network connection.
Each process registers a set of interfaces that are accessible remotely and client processes can request a pointer to these interfaces and invoke the methods available through them. In this case, the **object implementing the method becomes the server**. The underlying runtime infrastructure is in charge of transforming the local method invocation into a request to a remote process and collecting the result of the execution.
Of course, distributed object models introduce some complexities with respect to simple RPC because of the **object state management** (RPC is stateless). The method remotely executed is associated to an object instance that is created for the sole execution of the method (the object is destroyed as soon as the execution is finished), exist for a limited interval of time (the object handles all the request in that amount of time and not only one like in the previous case), or are independent from the existence of requests.
The common schema is the following:



1. The server process maintains a **registry of active objects** that are made available to other processes;
2. The client process, by using a given addressing scheme, **obtains a reference to the active remote object** (to the **Remote Reference Model**). An **Object Skeleton** is created and activated and the reference to the object is sent to the client;
3. The client process invokes the methods on the active object by calling them through the reference previously obtained. An **Object Proxy** is created that is no more than a local representation of the remote object (when communication with the proxy, in reality is interacting with the remote procedure). Parameters and return values are marshaled as happens in the case of RPC.

Distributed object frameworks introduce objects as first-class entities for IPC. They are the principal gateway for invoking remote methods but **can also be passed as parameters and return values**. This poses an interesting problem, since object instances are complex instances that encapsulate a state and might be referenced by other components. Passing an object as a parameter or return value involves the duplication of the instance on the other execution context. This operation leads to two separate objects whose state evolves independently. The duplication becomes necessary since the instance needs to trespass the boundaries of the process. So, in general it is possible to:

- **Marshaling by value**: What we have said so far. A copy of the object is created on the server. It is easy to do but it can lead to inconsistency problems;
- **Marshaling by reference**: There is no duplication of the object but it is more complex to manage because remote references have to be tracked. Being more complex and resource demanding, marshaling by reference should be used only when duplication of parameters and return values lead to unexpected and inconsistent behavior of the system.

We also have two types of object activation:

- **Server-based**: The object has its own life and the remote method invocation is occasional;
- **Client-based**: The object is created with the purpose of executing the method invoked (e.g. the remote object is functional to modify/access data/database on the server). We could have a new object for each invocation or only one instance at time.

Web Services is an implementation of RPC over the HTTP protocol. Several aspects make Web services the technology of choice for SOA:

- They allow for interoperability across different platforms and programming languages;
- They are based on well-known and vendor-independent standards such as HTTP, XML and JSON;
- They provide an intuitive and simple way to connect heterogeneous software systems, enabling the quick composition of services in a distributed environment;
- They define facilities for enabling service discovery, which allows system architects to more efficiently compose SOA applications, and service metering to assess whether a specific service complies with the contract between the service provider and the service consumer.

In Web Services, Method invocations are transformed in HTTP requests using specific protocols that are:

- **SOAP**: The interaction is done with messages that are XML documents that are sent using the HTTP protocol. These messages can be used for method invocation and result retrieval. SOAP has often been considered quite innocent because of the excessive use of markup that XML imposes for organizing the information into a well-formed document.
- **REST**: It provides a model for designing network-based software systems utilizing the client/server model and leverages the functionalities provided by HTTP for IPC without additional burden. In a RESTful system, a client sends a request over HTTP using the standard HTTP methods and the server issues a response that includes the representation of the resource. So, it uses HTTP methods explicitly to build requests, it is stateless, it has a directory structure that is created using URIs (Basic REST design principle establishes a one-to-one mapping between CRUD operations and HTTP methods) and it can transfer data using data structures such as XML or JSON.

3. Virtualization

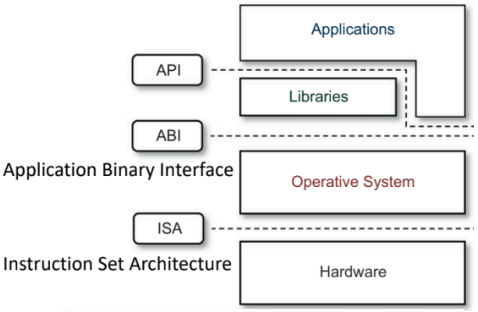
3.1 Introduction

Virtualization is meant to provide an abstract environment (whether virtual hardware or an operating system) to run applications. Virtualization technologies provide a virtual environment for not only executing applications but also for storage, memory, and networking.Virtualization technologies have gained renewed interest recently due to problems like: underutilization of hardware and software resources, lack of space, greening initiatives and rise of administrative costs. Three main characteristics of virtualization environments:

- **Increased Security**: All the operations of the guest are generally performed in the virtual machine, which then translates and applies them to the host. This level of indirection allows the virtual machine manager to

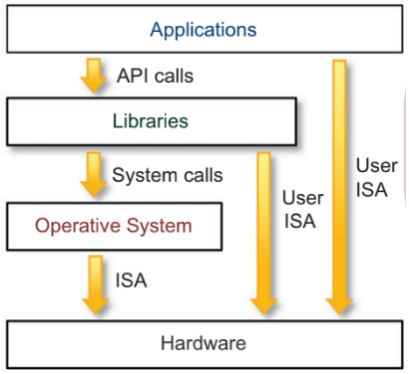
control and alter the activity of the guest, thus preventing some harmful operations from being performed. Resources exposed by the host can then be hidden or simply protected from the guest. Sensitive information that is contained in the host can be hidden without the need to install complex security policies;

- **Managed execution**: Virtualization of the execution environment not only allows increased security, but a wider range of features also can be implemented, such as sharing, aggregation, emulation, and isolation;
- **Portability**: In the case of a hardware virtualization solution, the guest is packaged into a virtual image that, in most cases, can be safely moved and executed on top of different virtual machines. In the case of programming-level virtualization, as implemented by the JVM or the .NET runtime, the binary code representing application components can be run without any recompilation on any implementation of the corresponding virtual machine.



In a virtualized environment there are three major components:

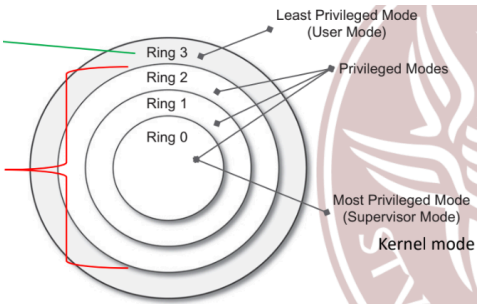
- The **guest** represents the system component that interacts with the virtualization layer rather than with the host, as would normally happen;
- The **host** represents the original environment where the guest is supposed to be managed.
- The **virtualization layer** is responsible for recreating the same or a different environment where the guest will operate.



In the case of hardware virtualization, the guest is represented by a system image comprising an operating system and installed applications. These are installed on top of virtual hardware that is controlled and managed by the virtualization layer, also called the virtual machine manager. The host is instead represented by the physical hardware, and in some cases the operating system, that defines the environment where the virtual machine manager is running.

Execution virtualization includes all techniques that aim to emulate an execution environment that is separate from the one hosting the virtualization layer. Virtualizing an execution environment at different levels of the computing stack requires a **reference model** that defines the interfaces between the levels of abstractions, which hide implementation details. To understand how the different virtualization techniques work it is fundamental to recall the layered structure of a computer system. In a machine we have the hardware, the OS that runs over the hardware and finally the applications. The OS communicates with the Hardware using Instruction Sets. Libraries can use system calls while applications usually use API calls. Notice that applications and libraries can perform some operations communicating directly with the hardware.
The instruction set exposed by the hardware has been divided into different security classes that determine who can operate with them. The first distinction can be made between:

- **Privileged instructions**: Those that are executed under specific restrictions and are mostly used for sensitive operations, which expose or modify the privileged state. Some example are the **behavior-sensitive instructions** that operate on the I/O and the **control-sensitive instructions** alter the state of the CPU registers;
- **Non Privileged instructions**: Those that can be used without interfering with other tasks because they do not access shared resources. Some examples are all the floating, fixed-point, and arithmetic instructions.



Usually, this type of architecture is build like a ring: the most inner circle (**Ring 0**) represents the most privileged mode (also called **Supervisor Mode**), then there are **Ring 1** and **Ring 2** that are privileged modes (often not used) and finally there is **Ring 3** that is the least privilege mode (also called **User Mode**).

All the current systems support at least two different execution modes:

- **Supervisor mode**: It denotes an execution mode in which all the instructions (privileged and non privileged) can be executed without

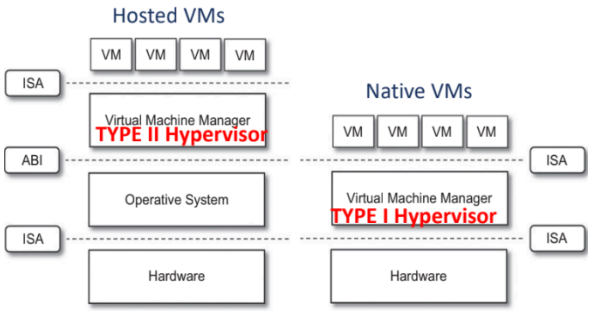
any restriction. This mode, also called kernel mode, is generally used by the operating system (or the **hypervisor**) to perform sensitive operations on hardware-level resources;

- **User mode**. In user mode, there are restrictions to control the machine-level resources. If code running in user mode invokes the privileged instructions, hardware interrupts occur and trap the potentially harmful execution of the instruction. Despite this, there might be some instructions that can be invoked as privileged instructions under some conditions and as non privileged instructions under other conditions.

Conceptually, the **hypervisor runs above the supervisor mode** but in reality, hypervisors are run in supervisor mode, and the division between privileged and non privileged instructions has posed challenges in designing virtual machine managers. It is expected that all the sensitive instructions will be executed in privileged mode, which requires supervisor mode in order to avoid traps. Without this assumption it is impossible to fully simulate and manage the status of the CPU for guest operating systems. More recent implementations of ISA1 (Intel-VTx and AMD-V) have solved this problem by redesigning such sensitive instructions as privileged ones.

3.2 Hardware-level Virtualization

Hardware-level virtualization is a virtualization technique that provides an abstract execution environment in terms of computer hardware on top of which a guest operating system can be run. In this model, the guest is represented by the operating system, the host by the physical computer hardware, the virtual machine by its emulation, and the virtual machine manager by the hypervisor. The hypervisor is generally a program or a combination of software and hardware that allows the abstraction of the underlying physical hardware.

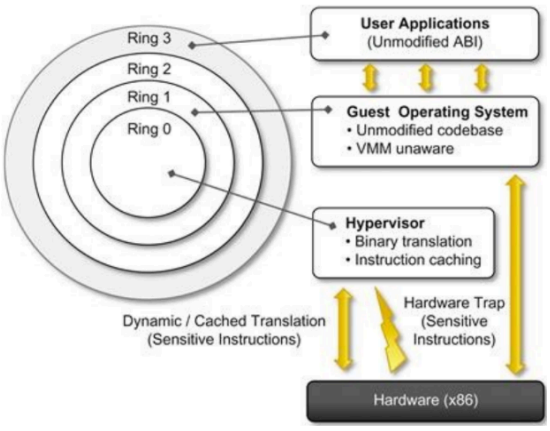


We already talked about the **hypervisor** (or **Virtual Machine Manager (VMM))** that recreates a hardware environment in which guest operating systems are installed. There are 2 types of hypervisors:

- **Type I**: It **runs directly on top of the hardware**. Therefore, they take the place of the operating systems and interact directly with the ISA interface of the underlying hardware.
- **Type II**: It requires the **support of an operating system** to provide virtualization services. This means that they are programs managed by the operating system, which interact with it through the ABI and emulate the ISA of virtual hardware for guest operating systems.

The hypervisor is formed by different elements that are the **dispatcher** that is invoked every time an instruction has to be executed and decides what to do, the **allocator** that is responsible for deciding the system resources to be provided to the VM and also granting the access to the specific resources and the **interpreter** that executes eventual privilege instructions (if not privileged it can executed directly on the hardware by the allocator).
Three properties should be satisfied by an hypervisor that are:

- Equivalence: A guest running under the control of a virtual machine manager should exhibit the same behavior as when it is executed directly on the physical host;
- Resource control: The virtual machine manager should be in complete control of virtualized resources;
- Efficiency: A statistically dominant fraction of the machine instructions should be executed without intervention from the VMM (so it should only intervene to sensitive instructions);



Hardware-level virtualization includes several strategies that differentiate from each other in terms of which kind of support is expected from the underlying hardware, what is actually abstracted from the host, and whether the guest should be modified or not. The different strategies are the following:

- **Full Virtualization**: This refers to the ability to run a program, most likely an operating system, directly on top of a virtual machine and without any modification, as though it were run on

the raw hardware (the OS is unaware that it is being emulated). To make this possible, virtual machine managers are required to provide a complete emulation of the entire underlying hardware. The principal advantage of full virtualization is complete isolation, which leads to enhanced security, ease of emulation of different architectures, and coexistence of different systems on the same platform. The VMM scans the instruction stream and decides what to do: Noncritical instructions run on the hardware directly while privileged, control- and behavior-sensitive, instructions (critical) are identified and trapped into the VMM, which emulates the behavior of these instructions.
The used emulation is the **binary translation** where sequences of instructions are translated from a source instruction set to the target instruction set. It can be:
    - **Static**: It aims to convert all of the code of an executable file into code that runs on the target architecture without having to run the code first (it might happen that some part of the application will never be executed but all the code is still translated);
    - **Dynamic**: It focuses on the short set of instructions that have to be executed by the hardware and then translates them. Code is only translated as it is discovered and when possible, branch instructions are made to point to already translated and saved code. It is slightly better than the static approach (even if there is a higher overhead) thanks to the re-execution of instruction and caching.
- **Hardware Assisted Virtualization**: This term refers to a scenario in which the hardware provides architectural support for building a virtual machine manager able to run a guest operating system in complete isolation. At present, examples of hardware-assisted virtualization are the extensions to the x86-64 bit architecture introduced with Intel-VTx and AMD-V, which are meant to reduce the performance penalties experienced by emulating x86 hardware with hypervisors (by design the x86 architecture did not meet the equivalence requirements because 17 sensitive instructions were called in user mode). Today there exist many hardware-assisted solutions like Kernel-based Virtual Machine (KVM), VirtualBox, Xen, VMware and Hyper-V.
- **Paravirtualization**: This is a not-transparent virtualization solution that allows implementing thin VMMs. Paravirtualization techniques expose a software interface to the virtual machine that is slightly modified from the host and, as a consequence, guests need to be modified. A paravirtualized VM provides special APIs requiring substantial OS modifications in user applications. Paravirtualized OS are assisted by an intelligent compiler to replace the non virtualizable OS instructions by hypercalls. The aim of paravirtualization is to provide the capability to demand the execution of performance-critical operations directly on the host, thus **preventing performance losses** that would otherwise be experienced in managed execution. This technique has been successfully used by Xen for providing virtualization solutions Linux-based operating systems specifically ported to run on Xen hypervisors (more info on slides).
- **Partial Virtualization**: This provides a partial emulation of the underlying hardware, thus not allowing the complete execution of the guest operating system in complete isolation. Partial virtualization allows many applications to run transparently, but not all the features of the operating system can be supported, as happens with full virtualization.

Besides being an enabler for computation on demand, virtualization also gives the opportunity to design more efficient computing systems by means of **consolidation**, which is performed transparently to cloud computing service users. Since virtualization allows us to create isolated and controllable environments, it is possible to serve these environments with the same resource without them interfering with each other. This practice is also known as server consolidation, while the movement of virtual machine instances is called **virtual machine migration**. Because virtual machine instances are controllable environments, consolidation can be applied with a minimum impact, using:

- **Off-line migration**: Temporarily stopping its execution and moving its data to the new resources;
- **On-line migration**: There is no disruption of the activity of the virtual machine instance so the VM is transferred without generating service discontinuity. This leads to a **small degradation in performance** when the migration is being performed but this isn't usually noticeable. There are few steps:
    - **Pre-migration**: Determine the migrating VM and the destination host. This can be performed manually, but in most circumstances it is automatically started by strategies such as load balancing and server consolidation;

- **Transferring the memory**: The whole execution state of the VM is stored in memory and sent to the destination node, ensuring continuity of service. **Dirty memory is copied iteratively until its last portion is small enough to be handled** (so the two memory are keeped synched until most of the data has been copied and the execution can be transferred);
- **Suspend the VM and copy the last portion of the data**: The VM is suspended and its apps no longer run. This “service unavailable” time is called the “downtime” of migration and it should be as short as possible. The migrating VM’s execution is suspended when the last round’s memory data is transferred. Other non-memory data such as CPU and network states should be sent as well;
- **Commitment**: The VM reloads its state and recovers the execution of its programs. The services provided by this VM continues;
- **Activation**: The network connection is redirected to the new VM and the dependency to the source host is cleared, removing the original VM from the source host.

3.3 System-level Virtualization

When talking about **System-level Virtualization**, we also mean **Containerization** that leverages multiprogramming techniques at OS level. This offers the opportunity to create different and separated execution environments for applications that are managed concurrently. Differently from hardware virtualization, there is no VMM (or hypervisor), and the **virtualization is done within a single operating system**, where the OS kernel allows for multiple isolated user space instances. The kernel is also responsible for sharing the system resources among instances.
Before continuing with this type of virtualization, it is better to explain what are the **differences with the Application containers and the System Containers**:

- Application containers usually contains a single process while the System containers contains an entire OS (an possibly application running on top of it);
- Application containers use a layered filesystem while the System containers use a neutral file system;
- Application containers can be used to run microservices/application while the System containers run a lightweight virtual machine;
- Application containers are used for distributed applications (e.g. when creating a distributed application, we can embed each microservice inside a container to deploy the application) while the System containers are used for providing underlying infrastructure (e.g. we can use containers as VM and then use it to build an infrastructure);

An example of an application container is Docker, while a system container is LXC (Linux container).

Containerization is an **evolution of the chroot mechanism**. The chroot operation changes the file system root directory for a process and its child processes so that they cannot have access to other portions of the file system than those accessible under the new root directory.

Compared to hardware virtualization, this strategy imposes little or no overhead because applications directly use OS system calls and there is no need for emulation. There is no need to modify applications to run them, nor to modify any specific hardware, as in the case of hardware-assisted virtualization. On the other hand, operating system-level virtualization does not expose the same flexibility of hardware virtualization, since all the user space instances must share the same operating system. This technique is an efficient solution for server consolidation scenarios in which multiple application servers share the same technology.
Some technologies to build containers are:

- **Namespace**: A namespace wraps a global system resource in an abstraction. We can associate processes to a namespace and these processes perceive they have their own isolated instance of the global resource. If changes to the global resource are made then those changes are only visible to members of the same namespace (so they are invisible to other processes out of the namespace).
- **Cgroups**: Control groups, usually referred to as Cgroups, are a Linux kernel feature that allows processes to be organized into hierarchical groups in which the usage of various types of resources can then be limited and monitored (e.g. we could assign only the 25% of the CPU to a certain container). We have different controllers (for I/O, CPU, Memory, PID, etc..) that allow us to define how resources are used. RIPARTI DA 18 MIN

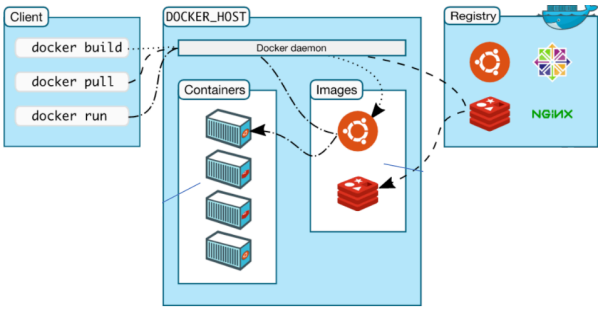
- **UnionFS**: UnionFS allows administrators to keep files separate physically and merge them logically into a single view. Each physical directory is called a branch. UnionFS is a layered filesystem on top of several filesystems or directories and can also be hierarchical. It exposes its own interface to the kernel so that the system call can operate on the filesystem.
  EX. Practical usage of UnionFS:
  We have 2 filesystems (Fruits, Vegetables) with 2 files where one of them is a duplicate (Tomato). We want to create a merge of the 2 and call it “healthy”.
  We call use the “mount” command specifying that we want to use UnionFS. We also have to specify the dirs to merge and doing so we establish a hierarchy: the first folder specified has a higher priority than the second one. This also removes the duplicates (in the new view we only find one “Tomato”).

```
$ ls /Fruits
Apple Tomato
$ ls /Vegetables
Carrots Tomato
$ cat /Fruits/Tomato
I am botanically a fruit.
$ cat /Vegetables/Tomato
I am horticulturally a vegetable.
# mount -t unionfs -o dirs=/Fruits:/Vegetables > none
/mnt/healthy
$ ls /mnt/healthy
Apple Carrots Tomato
$ cat /mnt/healthy/Tomato
I am botanically a fruit.
```

```
%%% here Fruits has
higher priority than
Vegatables
```

  To each branch is assigned a precedence so a branch with a higher precedence overrides a branch with a lower precedence. UnionFS operates on directories so if a directory exists in two underlying branches, the contents and attributes of the UnionFS directory are the combination of the two lower directories. UnionFS also automatically removes any duplicate directory entries so if a file exists in two branches, the contents and attributes of the UnionFS file are the same as the file in the higher-priority branch, and the file in the lower-priority branch is ignored.
  Unionfs also can **mix read-only and read-write branches**. In this case, the **union as a whole is read-write**. **Copy-on-write** semantics **gives the illusion that we can modify files and directories on read-only branches**.
  EX. /mnt/cdrom is read only (the cd rom is read only by definition)
  /tmp/cdpatch is read and write (created by the admin)
  # mount -t unionfs -o dirs=/tmp/cdpatch,/mnt/cdrom none /mnt/patched-cdrom
  Through /mnt/patched-cdrom, it appears you can write to the /mnt/cdrom so all writes actually will take place in /tmp/cdpatch. Writing to read-only branches results in an operation called a **copyup**.
  When a read-only file is opened for writing, the file is copied over to a higher-priority branch.
  If required, UnionFS automatically creates any needed parent directory hierarchy.

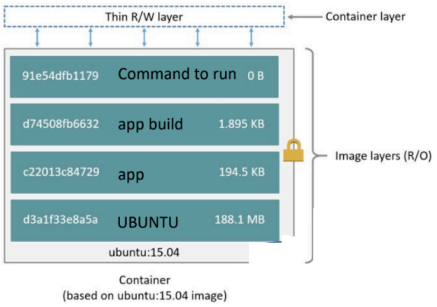
3.3 Process-level Virtualization with Docker



Docker provides the ability to package and run an application in a loosely isolated environment called a **container**. Unlike virtual machines, Docker containers do not use any hardware virtualization but it builds on top of the container technology already provided by the operating system kernel. Programs running inside Docker containers interface directly with the host's Linux kernel. Many programs can run in isolation without running redundant operating systems or suffering the delay of full boot sequences.
Docker uses a **client-server architecture**. The Docker client talks to the **Docker daemon**, which does the heavy lifting of building, running, and distributing your Docker containers. We usually make use of **CLI commands** to interact with the Docker daemon but we can also use the REST API already provided by Docker. The **Docker client** can interact with the daemon and they both can run on the same system, or you can connect a Docker client to a remote Docker daemon. The command used to build an image of a container is “*docker build*”. We also have a **registry** in which all the images of the containers are stored. When we want to execute a certain image, we can retrieve it using “*docker pull*” and execute it with “*docker run*” and the container is instantiated. Another Docker client is **Docker Compose**, which lets you work with applications consisting of a set of containers.

EX. A Docker command is as follows: docker run -i -t ubuntu /bin/bash
It locates and eventually downloads the Ubuntu image (if not locally stored), it creates a new container and it allocates a R/W file system as the final layer of the image (the file system is isolated from the host file system). It then creates a network interface connected to the default network, starts the container and runs the /bin/bash command.

What we call an image is a collection of image layers. A **layer** is a set of files distributed as an atomic unit. Images maintain parent/child relationships. The files available in a container are the union of all layers in the descendants of the image from which we create the container. **Images** can have relationships with any other image.
Programs inside containers know nothing about image layers. From inside a container, the filesystem runs like it's not in it and from the perspective of the container it has copies of the files provided by the image. This is possible thanks to **Union FileSystem (UFS)**. The filesystem is used to create mount points on the host filesystem that abstract the use of layers that are united in Docker image layers. In the same way the layers of a Docker image can be unpacked and configured for use by the specific filesystem provider. chroot is used to make the root of the image file system also the root in the container. This prevents anything running inside the container from accessing any other part of the host filesystem. We can specify the structure of a container using the **Dockerfile** in which we can specify the layers and the operations we need.



Dockerfile
```
FROM ubuntu:15.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

EX. Here we specify that we want to create a container from Ubuntu (we also specify the version). We want to copy a specific application inside the container and build it as soon as we run the container by executing the command in the last row. When executing the container, a R/W container layer is built that is the in memory filesystem binded to that specific container. To execute the Dockerfile, we can use the command "*docker build -t image_name_tag .*".

By default all files created inside a container are stored on a **writable container layer**. This means that:

- The data does not persist when that container no longer exists, and it can be difficult to get the data out of the container if another process needs it;
- A container writable layer is tightly coupled to the host machine where the container is running. You can not easily move the data somewhere else;
- Writing into a container writable layer requires a storage driver to manage the filesystem. The storage driver provides a union file system, using the Linux kernel. This extra abstraction reduces performance as compared to using data volumes, which write directly to the host filesystem.

Docker has two options for containers to store files in the host machine, so that the files are persisted even after the container stops:

- **Bind mounts**: A storage area that is in the Docker host when the container is running. It is a portion of the host storage that is shared with the host file system. A file or directory on the host machine is mounted into a container. In this case also the host machine can access that storage area and it can also be shared with other host processes. It offers a high performance (since it is directly managed by the host machine) but it is not portable because the full path of the storage area depends on the specific configuration of the machine. Some usages:
    - To share configuration files from the host machine to containers;
    - To share source code or to build artifacts between a development environment on the Docker host and a container;
    - The file or directory structure of the Docker host is guaranteed to be consistent with the one the containers require.
- **Volumes**: Volumes are the preferred way to persist data in Docker containers and services. A storage area created and managed by Docker and isolated from the host (so we can't access it from the host only through containers or Docker API). So, it is stored within a directory on the Docker host. It can be mounted into multiple containers simultaneously R/W or R. Some advantages are: Easier to back up or migrate than bind mounts; Volumes can be more safely shared among multiple containers, Volume drivers let you store

volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality; New volumes can have their content pre-populated by a container; Volumes work on both Linux and Windows containers; Volumes can be managed using Docker CLI commands or the Docker API. Some usages:
    - To share data among multiple running containers;
    - No guarantee to have a given directory or file structure in the Docker host;
    - To store container's data on a remote host or a cloud provider, rather than locally;
    - To back up, restore, or migrate data from one Docker host to another.
- **Tmpfs (Temporary filesystem)**: An area of memory outside the container writable layer. It is associated with a certain container and it is not shareable with other containers. The problems are that it is temporary, only persisted in the host memory and it is removed when the container stops. Moreover, it is only available in Linux hosts. Some usages:
    - Data shouldn't persist either on the host machine or within the container (e.g. for security reasons, or to manipulate large volumes of non-persistent data with high performance).

If running Docker on Windows we can also use a **named pipe**.

A **swarm** consists of multiple Docker hosts which run in swarm mode and act as managers and/or workers, which run swarm services. Any processes, functionality, or data that must be discoverable and available over a network is called a **service**. A service is defined by its optimal state (number of replicas, network and storage resources available, exposed ports, and more). It is possible to describe a service using a file "*docker-compose.yml*".

```yaml
version: "3"
services:
  web:
    # replace username/repo:tag
    with your name and image details
    image: username/repo:tag
    deploy:
      replicas: 5
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
      restart_policy:
        condition: on-failure
    ports:
      - "4000:80"
    networks:
      - webnet
networks:
  webnet:
```

There are different ways to connect the containers together (**networking**):

- **Bridge**: It's the default network driver. Bridge networks are useful when we need multiple containers to communicate on the same Docker host;
- **Host**: It's for standalone containers, removes network isolation between the container and the Docker host, and uses the host networking directly. It is the best solution when the network stack should not be isolated from the Docker host, but you want other aspects of the container to be isolated.
- **Overlay**: It connects multiple Docker daemons together and enables swarm services to communicate with each other. It is the best solution when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.
- **Macvlan**: It allows you to assign a MAC address to a container, making it appear as a physical device on your network. Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack, or when you are migrating from a VM setup.
- **None**: The container must be used only locally.

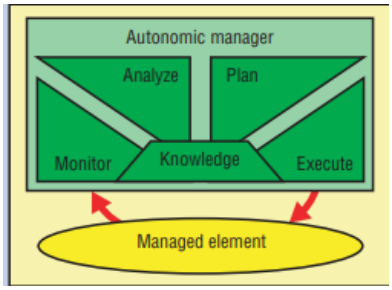
4. Autonomic Computing, Automation, Orchestration, Kubernetes

4.1 Autonomic Computing

First of all, let's analyze what are the main challenges in a datacenter:

- To consolidate servers: We would like to be able to pack the server in a VM so that it is possible to migrate it easily and rapidly in case of, for example, maintenance. Of course the service must be up most of the time during this operation;
- To guarantee Service Level Agreements: The necessity to reach specific service level such as throughput;
- To optimize energy consumption;
- Manage hw/sw failures: Failures must be discovered (possibly automatically), remediated and finally fixed;
- Software updates;
- Vulnerabilities/bugs fix: It must be done in multiple machines and not only one and this is way more difficult.

These actions are difficult to do by a single human that can only be involved in very critical decisions or any other physical tasks (e.g. replacing a broken disk) that cannot usually be performed by a machine. There are systems that were built to automate these operations.



Autonomic computing systems are computing systems that can manage themselves the high-level objectives from administrators. At the base **autonomic manager** that is responsible for monitoring the system state (e.g. performance, availability, health state), analyzing the monitored data (e.g. data aggregation and/or prediction and comparison of monitored data with thresholds) to verify the system is in the desired state or not, planning an action to keep the system in the desired state (e.g. scaling, replication, migration, re-routing, etc..) and executing the selected action (e.g. allocate/deallocate VMs). These 4 phases (Monitor, Analyze, Plan and Execute) are also named **MAPE-K** where K stands for "knowledge" that represents the set of information that allows the system to perform those actions (e.g. the list of system components on which to execute a certain action). Notice that there might be different components to manage so this principle can be also extended to a whole system with several components that have their autonomic computing system that act independently or coordinate with other ones to perform a certain operation.
There are some **self-* properties** in autonomic computing that are:

- **Self-Configuration**: The ability to accommodate varying and possibly unpredictable conditions so if there is something that changes in the state of system, it should be able to change its own configuration and accommodate to that state;
- **Self-Healing**: The ability to remain functioning by itself when problems (both hardware or software) arise (e.g. Docker monitors how many containers are running and if that number decreases then a new container is started). An example is to guarantee a given number of VMs or containers are always available (up & running). So monitor the health state of a VM/container (e.g. sending pings or service requests) and if a VM/container is down (does not respond) then start a new VM/container and add it to the application/load balancing pool;
- **Self-Protection**: The ability to detect threats and take appropriate actions. An example is when we want to keep intruders out of the system and possibly block future attacks;
- **Self-Optimization**: The ability to constant monitoring for optimal operation An example is to provide the right amount of VMs to guarantee SLA and minimize costs. So monitor metric used to define Service Level Objective (SLO) (e.g. CPU utilization, response time, availability), analyze if they are violated and then allocate/deallocate the right amount of VMs planned to solve the issue;

4.2 Autoscaling

Autoscaling is an example of self-optimization. There is a difference between performance or scalability: performance measures how fast and efficiently a system can complete certain tasks while **scalability measures the trend of performance with increasing load**. A system is scalable if it is **capable of maintaining performance under increased load** by adding more resources. This process can be done in **automatic way** and there are two possible approaches:

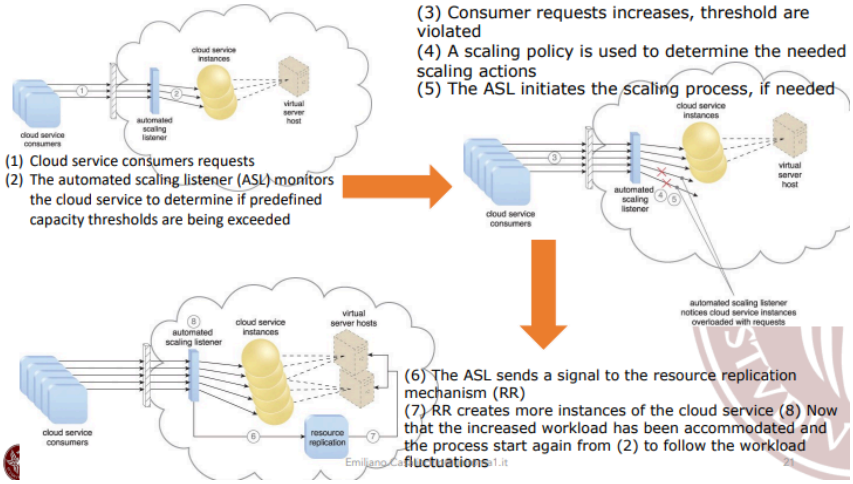
- Coarse grain: In this case the whole application is replicated and this typically imply the deployment of a new VM (with container technology could be also the replication of a container or a set of containers in a VM);
- Fine grain: In this case we perform the replication of an application component (micro-services);

The main components of the **dynamic scaling architecture** are the **automatic scaling listener** that is a component that looks at the amount of requests from a specific service and decides to add more VMs depending on the policy of the cloud administrator. This task should be accomplished in an automatic way.
In particular, there are predefined algorithms that define the scaling conditions that trigger the dynamic allocation of IT resources from resource pools. The automated scaling listener could take decisions based on the workload counters (e.g. number of service requests), the performance counters/metrics (e.g. CPU utilization, throughput, response time) or a combination of that and more complex counters/metrics. The auto scaling algorithm determines how many additional IT resources can be dynamically provided. There are possible actions:

- **Dynamic horizontal scaling**: IT resource instances are scaled out and in to handle fluctuating workloads. The automatic scaling listener monitors requests and signals resource replication to initiate IT resource duplication, as per requirements and permissions;
- **Dynamic vertical scaling**: IT resource instances are scaled up and down when there is a need to adjust the processing capacity of a single IT resource;
- **Dynamic relocation**: IT resources are relocated (migrated) to a host with more/less capacity.

There are some auto scaling algorithms:

- **Threshold based (reactive)**: When a threshold (on the workload or the resource utilization) is exceeded the scaling action is triggered. This approach is heuristic so we have to define the threshold beforehand but this isn't an optimal solution because it could lead to over/under provisioning;
- **Model based (reactive)**: It is defined by a mathematical model of the system that allows it to understand when the scaling action must be performed. So it is triggered by events set on performance metrics values and workload intensity;

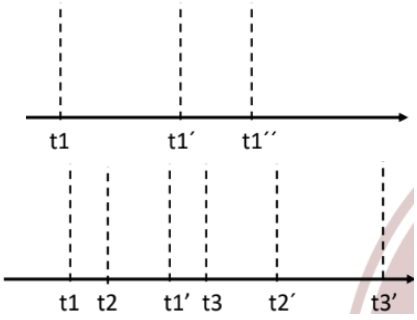


- **Proactive (threshold/model based)**: The values of the workload, resource utilization, performance metrics are predicted in the short term (e.g. 5 - 10 minutes ahead) and the algorithm (the same as above) takes scaling decisions on the predicted values. It could also be mixed with reactive approaches to adjust the decisions on the basis of the actual workload.

EX. Auto Scaling in AWS: In AWS there are some concepts concerning auto scaling that are the autoscaling group (a collection of EC2 instances that share similar characteristics and are treated as a logical grouping when scaling), launch configuration (a template that an auto scaling group uses to launch EC2 instances) and the scaling plan (the policy that determine the scaling actions). For what concerns the scaling plan, there are few approaches like maintaining the same instance level all the time (the amount of assigned resources is fixed), performing manual scaling, **scaling based on schedule** (allocating VMs based on the time of the day, e.g. allocate more VMs during the day and deallocate them during the night) and **scaling based on demand** (dynamic scaling). A scaling group can be configured with multiple scaling policies (at least one for scaling out and one for scaling in).

Some examples of **Auto Scaling algorithms** are the **simple scaling** and the **step scaling** in which we define how many instances to add/remove and "when" (that is usually determined by the occurrence of an event in which a threshold is exceeded). Let's see more slightly in detail how they work:

- In **simple scaling**, we can allocate a VM_1 that can perform certain actions. After a while, at time t1 a scaling action is determined (+/-) VMs or a VM is replaced because of health check conditions. In this case a new VM_2 is needed. The **time to scale** (to allocate VM2) is given by t1'-t1. One VM_2 is allocated, the system must balance the work among them and this is done during the **cold-down** (t1''-t1'). After that (at time t>t1''), a new scaling action can be taken;
- In **step scaling**, we can allocate a VM1 that can perform certain actions. At time t1 t2 and t3 a scaling action is determined (+/-) VMs or a VM is replaced because of health check conditions. So, a new VM_2 is allocated right after VM_1. The time needed to allocate a VM_i is the **time to scale** (ti'-ti). The main difference with the simple scaling is that there is **no cold-down** so a new action can be taken any time.



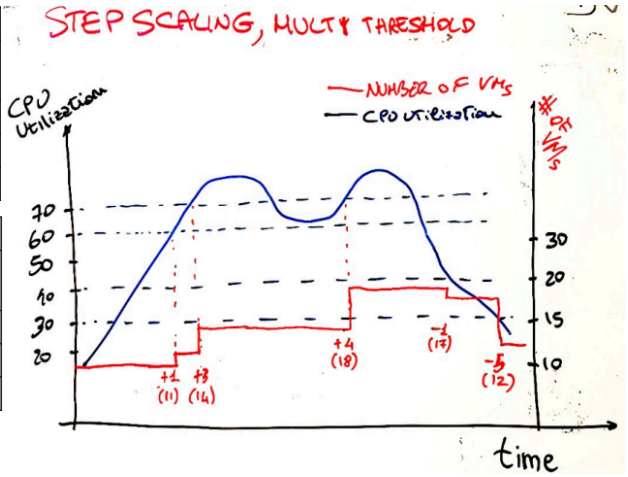
Another approach is the **target tracking scaling** in which we define a desired value for a performance or workload metric (e.g. CPU utilization, request count per target) and the system allocate/deallocate VMs to satisfy the desired value.

There are some **scaling policies** we can choose from:

- **Change in capacity** (capacity = number of VMs): It is requested to specify the adjustment in capacity (+/-). For example, If the current capacity of the group is 3 instances and the adjustment is 5, then when this policy is performed, Auto Scaling adds 5 instances to the group for a total of 8 instances;
- **Exact capacity**: It is requested to specify the new capacity of the scaling group. For example, if the current capacity of the group is 3 instances and the adjustment is 5, then when this policy is performed, Auto Scaling changes the capacity to 5 instances;
- **Percent capacity**: It is requested to specify the percentage of increment/decrement. For example, if the current capacity is 10 instances and the adjustment is 10 percent, then when this policy is performed, Auto Scaling adds 1 instance to the group for a total of 11 instances. Notice that there are some **rounding rules** (e.g. values greater than 1 are rounded down while values between 0 and 1 are rounded to 1).

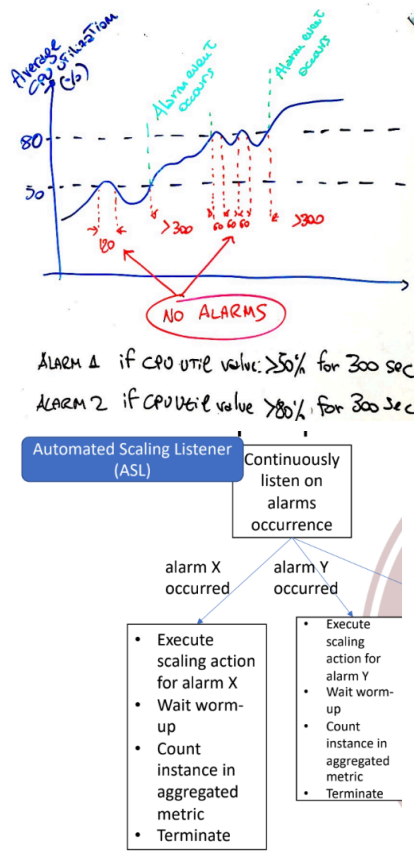
Scale out policy	
Adjustment (%)	Metric value
0	50 <= value < 60
10	60 <= value < 70
30	70 <= value < +infinity

Scale in policy	
Adjustment (%)	Metric value
0	40 < value <= 50
-10	30 < value <= 40
-30	-infinity < value <= 30



EX. An example for percent capacity:
Let's assume that we start from 10 VMs. CPU utilization starts rising and when it reaches 60% utilization, 10% more of VMs are allocated (+1 VM), then again when it reaches 70% where 30% more VMs are allocated (+3 VMs). When CPU utilization starts decreasing and it reaches 40%, the 10% of VMs are deallocated (-1 VM) and so on and so forth.

One important topic is the **aggregated metrics**. At a given instant of time t the "Performance Metric value" used by the scaling policy to take a scaling decision is the aggregated value of the "Performance Metric value" collected from all the instances, at time t. Number of seconds that it takes for a newly launched instance to work at its full capacity is called **warm-up**. Until its specified warm-up time has expired, an instance is not counted toward the aggregated metrics of the Auto Scaling group. In general, what happens is that a scale-in activity can't start while a scale-out activity is in progress so a better approach w.r.t. the previous one is that the alarm event occurs when a metric value is upper/lower than a desired threshold for a specific time period.
EX. In the example on the right, allocate/deallocate VMs only if CPU utilization remains stable for at least 5 min (300 sec). As it is possible to see, there are many times in which the CPU utilization increases or decreases a threshold, but only for a small interval of time (<300 sec) so in these cases we avoid performing the scaling operation frequently.



To sum up, the step scaling mechanism works this way:

1. ASL listens for the occurrence of an alarm;
2. If alarm X occur, pass the control to the component that:

- Execute scaling for X (change in capacity, exact capacity, percent capacity) so it add/remove EC2 instance, and/or launch an EC2 instance to replace a failed one;
- Wait warm-up period before counting the new instances toward the aggregated metrics of the Auto Scaling group;

3. Return listening (Step 1) without waiting for the new instances to be up and running and the warm up period expired.

4.3 Orchestration & Automation (Ansible and Kubernetes)

Cloud Orchestration (CO) is the process of automating the tasks needed to manage connections and operations of workloads. What Cloud Orchestration does is integrate automated tasks and processes into a workflow to perform specific business functions and ensure that processes have the proper permission to execute or connect to a workload. Some typical cloud orchestration tasks are:

- Provision or start server workloads;
- Provision storage capacity as needed;
- Instantiate virtual machines by orchestrating services, workloads, and resources in the cloud.

A practical example of cloud orchestration is **Ansible** that is an open source IT configuration management, deployment, and orchestration tool. Ansible works by connecting to your nodes and pushing out small programs (called **modules**) to these nodes. Modules are used to accomplish automation tasks in Ansible. These programs are written to be resource models of the desired state of the system.
Ansible is **agentless**, which means the nodes it manages do not require any software to be installed on them. Ansible reads information about which machines you want to manage from your inventory. Ansible has a default inventory file, but you can create your own and define which servers you want to be managed.
Ansible uses SSH protocol to connect to servers and run tasks. By default, Ansible uses SSH keys with ssh-agent and connects to remote machines using your current user name. Root logins are not required. You can log in as any user, and then su or sudo to any user. Once it has connected, Ansible transfers the modules required by your command or playbook to the remote machine(s) for execution. Ansible uses human-readable YAML templates so users can program repetitive tasks to happen automatically without having to learn an advanced programming language. **Ansible Playbooks** are used to orchestrate IT processes. A playbook is a YAML file (which uses a .yml or .yaml extension containing 1 or more plays), and is used to define the desired state of a system so it allows to state what each individual component of the IT infrastructure needs to do and allows components to react to discovered information and to operate with each other. This differs from an Ansible module, which is a standalone script that can be used inside an Ansible Playbook.
Plays consist of an ordered set of tasks to execute across a set of hosts (one, many or all of them) known as **inventory**. Tasks are the pieces that make up a play and call Ansible modules. In a play, tasks are executed in the order in which they are written. When Ansible runs, it can keep track of the state of the system. If Ansible scans a system and finds the playbook description of a system and the actual system state don't agree, then Ansible will make whatever changes are necessary for the system to match the playbook.
So the **core modules** are written in an manner to allow for easy configuration of desired state: they check that the task that is specified actually needs to be done before executing it (e.g. if an Ansible task is defined to start a web server, configuration is only done if the web server is not already started. This desired state configuration ensures that configuration can be applied repeatedly without side effects, and that configuration runs quickly and efficiently when it has already been applied).
The Playbook language includes a variety of features that allow for:

- Conditional execution of tasks;
- The ability to gather variables and information from the remote system;
- The ability to spawn asynchronous long running actions;
- The ability to operate in either a push or pull configuration, a “check” mode to test for pending changes without applying change;
- The ability to tag certain plays and tasks so that only certain parts of configuration can be applied.

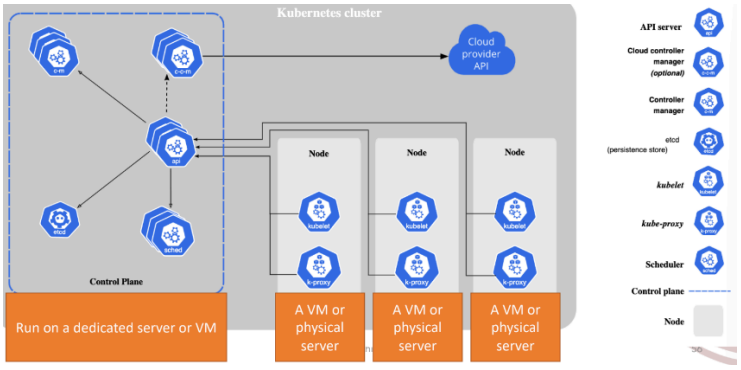
EX. Consider a traditional three-tier web application and its environment consisting of: application servers, database servers, content servers, load balancers and a monitoring system connected to an alert system such as a pager

notification service. Ansible can easily be used to implement a complex, clusterwide rolling update process that consists of:

1. Consulting a configuration/settings repository for information about the involved servers;
2. Configuring the base OS on all machines and enforcing desired state;
3. Identifying a portion of the web application servers to update;
4. Signaling the monitoring system of an outage window prior to bringing the servers off-line;
5. Signaling load balancers to take the application servers out of a load balanced pool;
6. Stopping the web application server;
7. Deploying or updating the web application server code, data, and content;
8. Starting the web application server;
9. Running appropriate tests on the new server and code;
10. Signaling the load balancers to put the application servers back into the load balanced pool;
11. Signaling the monitoring system to resume alerts on any detected issues on those servers;
12. Repeating this process for remaining application servers in a rolling update process;
13. Repeating these rolling update processes for other tiers such as database or content tiers;
14. Sending email reports and logging as desired when updates are complete.

Another example of cloud orchestration is **Kubernetes**, which is a platform for managing containerized workloads and services. Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications. Therefore, containers are considered lightweight. Similar to a VM, a container has its own filesystem, share of CPU, memory, process space, and more. As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions. Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more. For example: Kubernetes can easily manage a canary deployment for your system. Managing containerized workload includes:

- **Service discovery and load balancing**: Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
- **Storage orchestration**: Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks**: You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.
- **Automatic bin**: Packing You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
- **Self-healing**: Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.
- **Secret and configuration management**: Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.

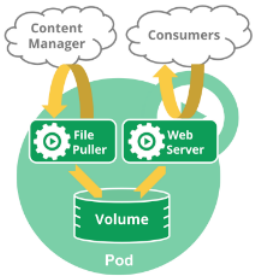


A **Kubernetes cluster** consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.

The worker node(s) host the Pods that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

Before explaining the different components, it is necessary to explain some concepts:

- **Objects**: K8s objects are persistent entities in the K8s system. K8s uses these entities to represent the state of the cluster. Specifically, they can describe:
    - What containerized applications are running (and on which nodes);
    - The resources available to those applications;
    - The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance;
    - A K8s object is a "record of intent" (once the object is created, the K8s system will constantly work to ensure that object exists);
    - An object tells the Kubernetes system what the cluster's workload should look like; this is the cluster's desired state.
- **Pods**: Pods are the smallest deployable units of computing that you can create and manage in Kubernetes. A Pod (as in a pod of whales or pea pod) is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled. In non-cloud contexts, applications executed on the same physical or virtual machine are analogous to cloud applications executed on the same logical host.The shared context of a Pod is a set of Linux namespaces, cgroups, and potentially other facets of isolation (the same things that isolate a Docker container). Within a Pod's context, the individual applications may have further sub-isolations applied.



- **Nodes**: A node is a virtual or physical machine, member of a K8s cluster. Each node is managed by the control plane and contains the services necessary to run Pods. Nodes run containerized workload. They can become member of a K8s cluster in two way:
    - The kubelet on a node self-registers to the control plane (API-server);
    - A human user manually adds a Node object to the API-server.

  Once a Node object is created the control plane checks whether the new Node object is valid (all the services are running). If the node object is valid it could run a Pod. The status of a node is usually described by the address, the condition (e.g. Ready, DiskPressure, MemoryPressure, PidPressure, NetworkUnavailable) and some info.

JSON manifest describing a node to be created

```
{
  "kind": "Node",
  "apiVersion": "v1",
  "metadata": {
    "name": "10.240.79.157",
    "labels": {
      "name": "my-first-k8s-node"
    }
  }
}
```

Now, the K8s **control plane components** are:

- **Kube-apiserver**: The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane. The main implementation of a Kubernetes API server is kube-apiserver. kube-apiserver is designed to scale horizontally—that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances;
- **etcd**: Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data. If your Kubernetes cluster uses etcd as its backing store, make sure you have a back up plan for that data;

JSON structure describing an healty node

```
"conditions": [
  {
    "type": "Ready",
    "status": "True",
    "reason": "KubeletReady",
    "message": "kubelet is posting ready status",
    "lastHeartbeatTime": "2019-06-05T18:38:35Z",
    "lastTransitionTime": "2019-06-05T11:41:27Z"
  }
]
```

- **kube-scheduler**: Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on. Factors taken into account for scheduling decisions include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines;

- **kube-controller-manager**: Control plane component that runs controller processes. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.Runs controller processes such as:
    - Node controller: Responsible for noticing and responding when nodes go down;
    - Job controller: Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion;
    - Endpoints controller: Populates the Endpoints object (that is, joins Services & Pods);
    - Service Account & Token controllers: Create default accounts and API access tokens for new namespaces.
- **cloud-control-manager**: A Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster. The cloud-controller-manager only runs controllers that are specific to your cloud provider. If you are running Kubernetes on your own premises, or in a learning environment inside your own PC, the cluster does not have a cloud controller manager.

The K8s **control plane components** are:

- kubelet: An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes;
- kube-proxy: kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept. kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster. kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself;
- container runtime: The container runtime is the software that is responsible for running containers. Kubernetes supports container runtimes such as containerd, CRI-O, and any other implementation of the Kubernetes CRI (Container Runtime Interface).

For what concerns **K8s controllers**: Controllers are control loops that watch the state of a K8s cluster, then make or request changes where needed. A controller tracks at least one Kubernetes resource type. These objects have a spec field that represents the desired state. The controller(s) for that resource are responsible for making the current state come closer to that desired state. A controller will send messages to the API server that have useful side effects.
EX. Job Controller: Job is a Kubernetes resource that runs a Pod, or perhaps several Pods, to carry out a task and then stop. Once scheduled, Pod objects become part of the desired state for a kubelet. When the Job controller sees a new task it makes sure that, somewhere in your cluster, the kubelets on a set of Nodes are running the right number of Pods to get the work done. The Job controller tells the API server to create or remove Pods. Other components in the control plane act on the new information (there are new Pods to schedule and run), and eventually the work is done. After a new Job is created, the desired state is for that Job to be completed. The Job controller makes the current state for that Job be nearer to your desired state: Creating Pods that do the work you wanted for that Job, so that the Job is closer to completion. Once the work is done for a Job, the Job controller updates that Job object to mark it Finished.

For what concerns **K8s schedulers**: A scheduler watches for newly created Pods that have no Node assigned. For every Pod that the scheduler discovers, the scheduler becomes responsible for finding the best Node for that Pod to run on. The scheduler reaches this placement decision taking into account the scheduling principles described below.

kube-scheduler is the default scheduler for Kubernetes and runs as part of the control plane. Kube-scheduler selects an optimal node to run newly created or not yet scheduled (unscheduled) pods. Since containers in pods - and pods themselves - can have different requirements, the scheduler filters out any nodes that don't meet a Pod's specific scheduling needs. Alternatively, the API lets you specify a node for a Pod when you create it, but this is unusual and is only done in special cases.

In a cluster, Nodes that meet the scheduling requirements for a Pod are called **feasible nodes**. If none of the nodes are suitable, the pod remains unscheduled until the scheduler is able to place it.
The scheduler finds feasible Nodes for a Pod and then runs a set of functions to score the feasible Nodes and picks a Node with the highest score among the feasible ones to run the Pod. The scheduler then notifies the API server about this decision in a process called binding.
Factors that need to be taken into account for scheduling decisions include individual and collective resource requirements, hardware / software / policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and so on.
kube-scheduler selects a node for the pod in a 2-step operation:

- The filtering step finds the set of Nodes where it's feasible to schedule the Pod. For example, the PodFitsResources filter checks whether a candidate Node has enough available resources to meet a Pod's specific resource requests. After this step, the node list contains any suitable Nodes; often, there will be more than one. If the list is empty, that Pod isn't (yet) schedulable.
- In the scoring step, the scheduler ranks the remaining nodes to choose the most suitable Pod placement. The scheduler assigns a score to each Node that survived filtering, basing this score on the active scoring rules.

There are two supported ways to configure the filtering and scoring behavior of the scheduler:

- Scheduling Policies allow you to configure Predicates for filtering and Priorities for scoring.
- Scheduling Profiles allow you to configure Plugins that implement different scheduling stages, including: QueueSort, Filter, Score, Bind, Reserve, Permit, and others. You can also configure the kube-scheduler to run different profiles.

5. Cloud Storage

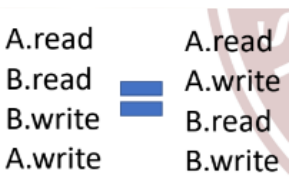
5.1 Basic Concepts

There are various form of data storage in cloud systems:

- Distributed file systems:
  - Google File System (GFS), Hadoop Distributed File System (HDFS);
  - GFS and HDFS rely on concept early developed in Network File Systems and Parallel File Systems;
- NoSQL database (or NoSQL datastore or simply datastore): e.g. Cassandra, MongoDB;
- Key-value storage systems: e.g. BigTable, Dynamo.

First of all, let's talk about **atomicity**. Parallel and distributed applications must take special precautions for handling shared resources. In many cases, a multistep operation should be allowed to **proceed to completion without any interruptions**, and the **operation should be atomic**. The instruction sets of most processors support the **test-and-set instruction**, which writes to a memory location and returns the old content of that memory cell as non-interruptible operations. Other architectures support **compare-and-swap**, an atomic instruction that compares the contents of a memory location to a given value and, only if the two values are the same, modifies the contents of that memory location to a given new value.
Two flavors of atomicity can be distinguished:

- **All-or-nothing** means that either the entire atomic action is carried out, or the system is left in the same state it was before the atomic action was attempted. This kind of atomicity is performed in two phases:
  - **Pre-commit phase**: Preparatory actions that can be undone. It includes the allocation of a resource, fetching a page from secondary storage, allocation of memory on the stack;
  - **Commit-point**.
  - **Post-commit**: Phase/commit step that are irreversible actions. It includes the alteration of the only copy of an object.

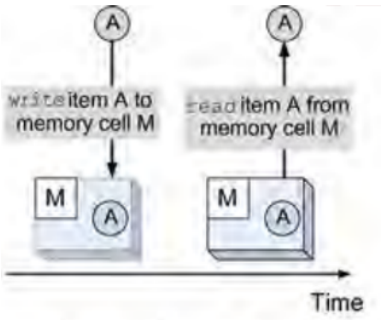


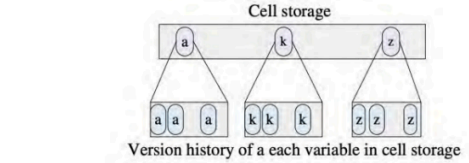
To manage failures and ensure consistency we need to maintain the history of all the activities so **logs are necessary**. Concurrent actions have the before-or-after property if their effect from the point of view of their invokers is as if the actions occurred either completely before or completely after one another.

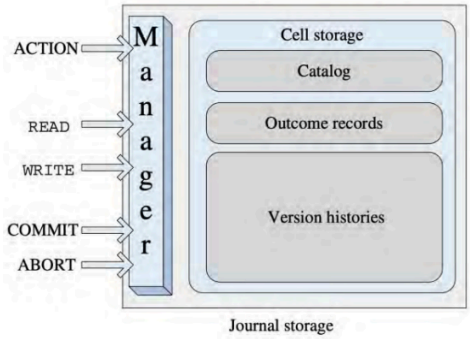
- **Before-or-after** atomicity means that, from the point of view of an external observer, the effect of multiple actions is as though these actions have occurred one after another, in some order. A stronger condition is to impose a sequential order among transitions. The result of every read or write is the same as if that read or write occurred either completely before or completely after any other read or write.

A **physical storage** is a local disk, a removable USB disk, a disk accessible via a network and it has a solid or magnetic state. A **storage model** describes the layout of a data structure in physical storage while a data model captures the most important logical aspects of a data structure in a database. The desired properties that we would like to have are read/write coherence (the result of a read of a memory cell M should be the same as the most recent write to that cell), Before-or-after atomicity and All-or-nothing atomicity.
Two abstract models of storage are commonly used: cell storage and journal storage:

- **Cell storage** assumes that the storage consists of **cells of the same size** and that **each object is exactly in one cell**. It Reflects the physical organization of common storage media in fact we have a primary memory (organized as an array of memory cells) and a secondary storage device (e.g. a disk) (organized in sectors or blocks). Once the content of a cell is changed by an action, there is no way to abort the action and restore the original content of the cell. To be able to restore a previous value we have to maintain a version history for each variable in the cell storage;
- **Journal storage** consists of a **manager** and **cell storage**, where the entire history of a variable is maintained in the cell storage, rather than just the current value. The user does not have direct access to the cell storage; instead the user can request the journal manager to (i) start a new action; (ii) read the value of a cell; (iii) write the value of a cell; (iv) commit an action; or (v) abort an action. An all-or-nothing action first records the action in a log in journal storage and then installs the change in the cell storage by overwriting the previous version of a data item. Many cloud applications must support online transaction processing and have to guarantee the correctness of the transactions. Transactions consist of multiple actions and the system may fail during or after each one of the actions, and steps to ensure correctness must be taken. Correctness of a transaction means that the result should be guaranteed to be the same as though the actions were applied one after another, regardless of the order. To guarantee correctness, a transaction-processing system supports all-or-nothing atomicity.
  Now, The log is always kept on nonvolatile storage, in particular, the considerably larger cell storage resides typically on nonvolatile memory, but can be held in memory for real-time access or using a write-through cache.







So, to sum up, the main differences between Cell storage and Journal storage are:

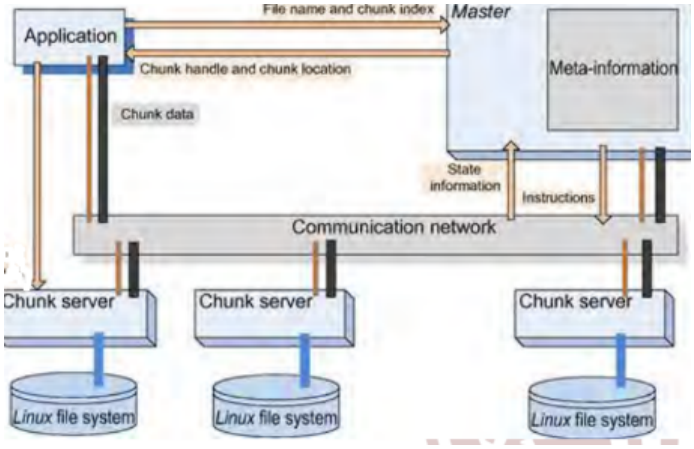
- Cell storage does not guarantee all-or-nothing atomicity so once the content of a cell is changed by an action, there is no way to abort the action and restore the original content of the cell. It guarantees read/write coherence and before-or-after atomicity;
- Journal storage guarantees all-or-nothing atomicity. When an action is performed first is modified the version history (log); then is modified the cell store (commit). If one of the two operations fail the action is discarded (abort).

5.2 Google File System

The GFS was designed after a careful analysis of the file characteristics and of the access models:

- Component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of commodity storage machines, accessed by a comparable number of clients. Therefore, scalability and reliability are critical features of the system.
- **File sizes range from a few GB to hundreds of TB**;
- **Random writes are extremely infrequent**, and the most common operation is to append to an existing file. Once written, files are **only read**, and often **only sequentially**.
- Users often process data in bulk and are less concerned with the response time, and the consistency model should be relaxed to simplify the system implementation.

For what concerns the architecture, a **GFS cluster** consists of a single master and multiple chunkServers, and is accessed by multiple clients. The master controls a large number of chunk servers and maintains metadata such as filenames, access control information, the location of all the replicas for every chunk of each file, and the state of individual chunk servers. The locations of the chunks are stored only in the control structure of the master's memory and are updated at system startup or when a new chunk server joins the cluster. This strategy allows the master to have up-to-date information about the location of the chunks.



Files are divided into **fixed-size 64-MB chunks** and stored on chunkServers' local disks as Linux files (so we have Linux File Systems). For reliability, each chunk is **replicated by default on three chunkServers**. A large chunk allows:

- To optimize performance for large files and to reduce the amount of metadata maintained by the system;
- To increase the likelihood that multiple operations will be directed to the same chunk;
- To reduces the number of requests to locate the chunk;
- To maintain a persistent network connection with the server where the chunk is located;
- To reduce disk fragmentation.

System reliability is a major concern. To **recover in case of failure**:

- The operation log maintains a historical record of metadata changes;
- Changes are atomic and are not made visible to the clients until they have been recorded on multiple replicas on persistent storage;
- The master replays the operation log.

To **minimize the recovery time**, the master periodically check points its state and at recovery time replays only the log records after the last checkpoint.

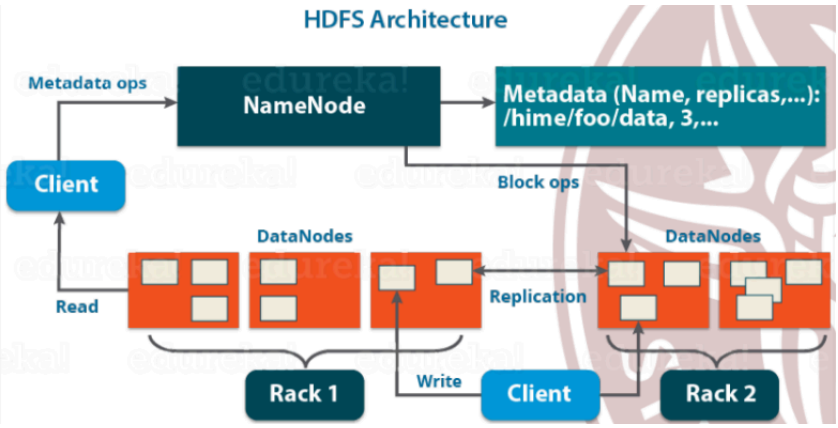
So, the master maintains all file system metadata such as the namespace, access control information, mapping from files to chunks, and current locations of chunks. Neither clients nor chunkServers cache file data, since most clients' applications stream through huge files or have datasets too large to be cached. However, clients do cache metadata. Having a single master vastly simplifies the overall system design, but its involvement must be minimized in order to avoid possible bottlenecks. Files creation is handled directly by the master, while clients never read and write file data through the master. Instead, clients ask the master which chunkServer it should contact and caches this information. The master does not keep a persistent record of which chunkServers have a replica of a given chunk. It simply polls chunkServers for that information at startup. Furthermore, the master keeps in stable memory an operational log of critical metadata changes. Given its nature, log changes are not visible to clients until metadata changes are made persistent. The master recovers its file system state by replaying the operation log, and checkpoints its state whenever the log grows beyond a certain size.

Now, the **file access** is handled by the App and Chunk server. The master grants a lease to a chunk server (primary) The primary chunk server is responsible to handle the update. The **file creation** is handled by the master. The **write on a file** works this way:

1. Client contacts the master which assigns a lease to one of the chunk servers (the primary) and reply with ID of primary and secondary chuck servers;
2. Client send data to all chunk servers:
    a. Chuck server store data in a LRU buffer;
    b. Chunk servers send ack to client;
3. Client send write req to primary:
    a. Primary apply mutation to file;
4. The primary send write to secondary;
5. Each secondary send ack to primary after mutation are applied;
6. The primary acks the client.

5.3 Hadoop Distributed File System (HDFS)

Hadoop provides a distributed le system and a framework for the analysis and transformation of very large datasets using the **MapReduce paradigm**. HDFS stores metadata on a dedicated server, called the **NameNode**. Application data is stored on other servers called **DataNodes**. File content is split into large blocks, typically 65/128 MB, and each block is **independently replicated at multiple DataNodes for reliability**. The NameNode maintains the namespace tree and the mapping of le blocks to DataNodes. So, we have a **master/slave architecture**:



- The NameNode (master): Manages the File System Namespace, stores metadata, controls access to files, records changes in a log and checks DataNode liveness;
- DataNode (slave) performs low level R/W operations.

For what concerns the **I/O operations**. An application adds data to HDFS by creating a new file and writing the data to it. After the file is closed, the bytes written cannot be altered or removed except that new data can be added to the file by reopening the file for append. HDFS implements a single-writer, multiple-reader model. The design of HDFS I/O is particularly optimized for batch processing systems, like MapReduce, which requires high throughput for sequential reads and writes.
An HDFS file consists of blocks. When there is a need for a new block, the NameNode allocates a block and determines a list of dataNodes to host replicas of the block. The dataNodes form a pipeline, the order of which minimizes the total network distance from the client to the last DataNode. Bytes are pushed to the pipeline as a sequence of packets. The client is then responsible for sending an acknowledgement message to the NameNode. When a client opens a file to **read**, it fetches the list of blocks and the locations of each block replica from the NameNode. The locations of each block are ordered by their distance from the reader. A read may fail if the target DataNode is unavailable, the node no longer hosts a replica of the block, or the replica is found to be corrupt when checksums are tested.

5.4 NoSQL datastore (Google BigTable and Amazon DynamoDB)

Relational DBMSs guarantee atomicity, consistency, isolation, durability (ACID), but usually, they are not scalable (in speed and size). Many cloud services are based on **online transaction processing (OLTP)** and operate under tight latency constraints. Moreover, these applications have to deal with extremely high data volumes and are expected to provide reliable services for very large communities of users.
A major concern for the designers of OLTP systems is to **reduce the response time**. **Scalability is the other major concern** for cloud OLTP applications and implicitly for datastores. There is a distinction between **vertical scaling**, where the data and the workload are distributed to systems that share resources such as cores and processors, disks, and possibly RAM, and **horizontal scaling**, where the systems do not share either primary or secondary storage (so basically there is an issue regarding consistency among multiple distributed copies/portion of the DB). Relational databases are not able to handle the massive amount of data and the real-time demands. The **soft-state approach in the design of NoSQL** allows **data to be inconsistent** and **ensures that data will be eventually consistent at some future point in time** instead of enforcing consistency at the time when a transaction is committed. Data partitioning among multiple storage servers and data replication are also tenets of the NoSQL philosophy in fact they increase availability, reduce response time, and enhance scalability. So, the main property of the NoSQL databases are:

- They are designed to scale well (horizontally);
- They do not exhibit a single point of failure;
- They have built-in support for consensus-based decisions (where decisions are taken on the majority of the votes);
- They support partitioning and replication as basic primitives.

Bigtable is built upon GFS to store log and data files, it employs the Google SSTable le format to store internal data, and relies on a highly-available and persistent distributed lock service called Chubby. It is based on a simple and flexible data model that:

- Support dynamic control over data layout and format;
- Clients can control the locality properties of the data represented in the underlying storage by using appropriate schema;
- Data is indexed using row and column names that can be arbitrary strings;
- Data is treated as uninterpreted strings;
- The client can control serving data from memory or disk.

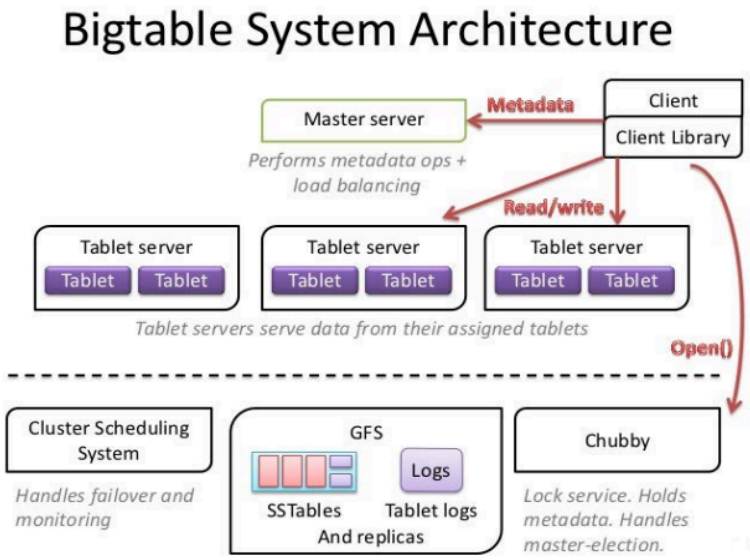
A Bigtable is a sparse, distributed, persistent multidimensional **sorted map**. Each value in the map is an uninterpreted array of bytes (string). The **map is indexed by a row key, column key, and a timestamp**:

$$(\text{row}{:}string, \text{column}{:}string, \text{time}{:}int64) \rightarrow string$$

How do indexes work in Big Data?
The **row keys** in a table are arbitrary strings and every read or write of data under a single row key is atomic, a design decision that makes it easier for clients to reason about the system's behavior in the presence of concurrent updates to the same row. The row range for a table is dynamically partitioned. Each row range is called a **tablet**, which is the unit of distribution and load balancing.
Column keys are grouped into sets called **column families**, which form the basic unit of access control. All data stored in a column family is usually of the same type and compressed together. A column family must be created before data can be stored under any column key in that family. The number of distinct column families in a table is kept small, and families rarely change during operation. In contrast, a table may have an unbounded number of columns.
Each cell in a Bigtable can contain multiple versions of the same data and these versions are indexed by **timestamp**. Different versions of a cell are stored in decreasing timestamp order, so that the **most recent versions can be read first**. To make the management of versioned data less onerous, Bigtable garbage-collects cell versions automatically. The client **can specify either that only the last n versions of a cell be kept**, or **that only new-enough versions be kept**.

How is Big Data implemented?
There are three BigTable building blocks: GFS, Google SSTable file format, and Chubby.
Google SSTable file format is used internally to store Bigtable data. SSTable provides a persistent, ordered immutable map from keys to values. Each SSTable contains a sequence of blocks and a block index to locate blocks.

For what concerns the architecture, Bigtable cluster **stores a number of tables**. Each table consists of a set of tablets, and each tablet contains **all data associated with a row range**. Initially, each table consists of just one tablet. As a table grows, it is automatically split into multiple tablets.
The **master** is responsible for **assigning tablets to TabletServers**, balancing tablet-server load, and garbage collection of files in Google File System. In addition, it handles schema changes such as table and column family creations. Each tabletServer manages a set of tablets, and can be dynamically added (or removed) from a cluster to accommodate changes in workloads. A **tabletServer handles read and write requests to the tablets that it has loaded**, and also **splits tablets that have grown too large**.
As with many single-master distributed storage systems, client data does not move through the master: clients communicate directly with tabletServers for reads and writes. Because Bigtable clients do not rely on the master for tablet location information, most clients never communicate with the master.



Chubby is a highly available and persistent distributed lock service. It provides a lock mechanism based on files and directories.

Amazon Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance.
There are many services on Amazon's platform that only need primary-key access to a data store. Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing, and consistency is facilitated by object versioning. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol.
The requirements are:

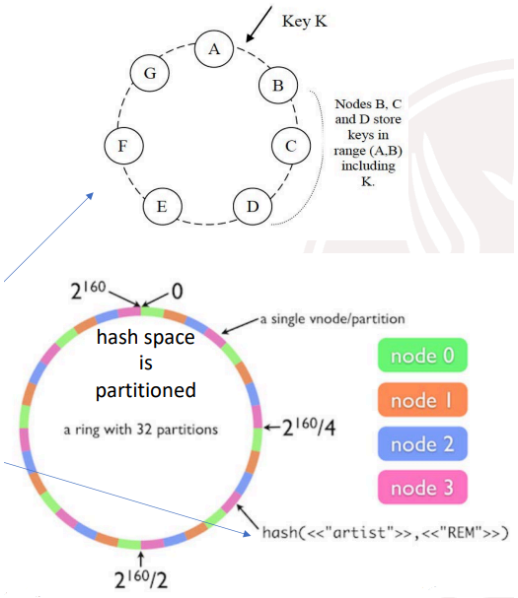
- Query model: Simple read and write operations to a data item that is uniquely identified by a key. No operations span multiple data items and there is no need for relational schema;
- ACID properties: Experience at Amazon has shown that data stores that provide ACID guarantees tend to have poor availability. Dynamo targets applications that operate with weaker consistency (the C in ACID) if this results in high availability
- Efficiency and other assumptions: The system needs to function on a commodity hardware infrastructure. The tradeos are in performance, cost efficiency, availability, and durability guarantees. Furthermore, the DynamoDB operation environment is assumed to be non-hostile and there are no security related requirements such as authentication and authorization.

Other considerations:

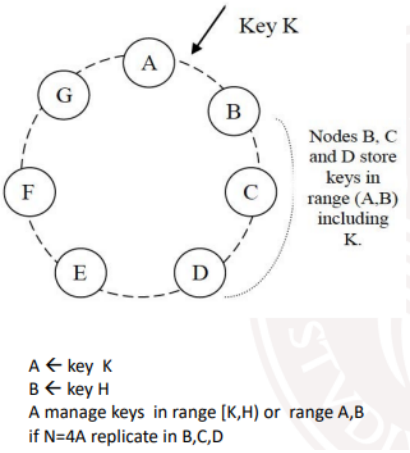
- Incremental scalability: Dynamo should be able to scale out one storage host at a time, with minimal impact on both operators of the system and the system itself;
- Symmetry. Every node in Dynamo should have the same set of responsibilities as its peers;
- Decentralization. The design should favor decentralized peer-to-peer techniques over centralized control;
- Heterogeneity: The system needs to be able to exploit heterogeneity in the infrastructure it runs on (e.g. the work distribution must be proportional to the capabilities of the individual servers).

For what concerns the **system interface**. Dynamo stores objects associated with a key through a simple interface; it exposes two operations: get( ) and put( ). The **get(key) operation** locates the object replicas associated with the key in the storage system and returns a single object or a list of objects with conflicting versions along with a context. The **put(key, context, object)** operation determines where the replicas of the object should be placed based on the associated key, and writes the replicas to disk.
The context encodes system metadata about the object that is opaque to the caller and includes information such as the version of the object. The context information is stored along with the object so that the system can verify the validity of the context object supplied in the put request.

For what concerns the **partitioning algorithm**. One of the key design requirements for Dynamo is that it must scale incrementally. This requires a mechanism to dynamically partition the data over the set of nodes in the system. Dynamo's partitioning scheme relies on consistent hashing to distribute the load across multiple storage hosts. In consistent hashing, the output range of a hash function is treated as a fixed circular space or ring. Each node in the system is assigned a random value within this space which represents its position on the ring. Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item's position. The basic consistent hashing algorithm presents some challenges. First, the random position assignment of each node on the ring leads to non-uniform data and load distribution. Second, the basic algorithm is oblivious to the heterogeneity in the performance of nodes. To address these issues, Dynamo uses a variant of consistent hashing: instead of mapping a node to a single point in the circle, each node gets assigned to multiple points in the ring. To this end, Dynamo uses the concept of virtual nodes. A virtual node looks like a single node in the system, but each node can be responsible for more than one virtual node. Effectively, when a new node is added to the system, it is assigned multiple positions (henceforth, tokens) in the ring.



For what concerns **replication**. To achieve high availability and durability, Dynamo replicates its data on multiple hosts. Each data item is replicated at N hosts, where N is a parameter configured per-instance. Each key, k, is assigned to a coordinator node. The coordinator is in charge of the replication of the data items that fall within its range. In addition to locally storing each key within its range, the coordinator replicates these keys at the N −1 clockwise successor nodes in the ring. This results in a system where each node is responsible for the region of the ring between it and its N_{th} predecessor.



For what concerns **data versioning**. Dynamo provides eventual consistency, which allows for updates to be propagated to all replicas asynchronously. A put( ) call may return to its caller before the update has been applied at all the replicas, which can result in scenarios where a subsequent get( ) operation may return an object that does not have the latest updates. There is a category of applications in Amazon's platform that can tolerate such inconsistencies and can be constructed to operate under these conditions.
In order to provide this kind of guarantee, Dynamo treats the result of each modification as a new and immutable version of the data. It allows for multiple versions of an object to be present in the system at the same time. Most of the time, new versions subsume the previous version(s), and the system itself can determine the authoritative version. Dynamo uses vector clocks in order to capture causality between different versions of the same object.