CLOUD COMPUTING

1. Fundamental concepts and models

1.1. The cloud.

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

This definition describes cloud computing as aphenomenon 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 as **XaaS**. This new approach significantly influences not only the way that we build software but also the way we deploy it, make it accessible, and design our IT infrastructure, and even the way companies allocate the costs for IT needs. 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. 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. Even though many cloud computing services are freely available for single users, enterprise-class services are delivered according a specific pricing scheme. In this case users subscribe to the service and establish with the service provider a Service-Level Agreement (**SLA**), defining the quality-of-service parameters under which the service is delivered.

1.1.1. *IT resources.* A physical or virtual IT-related artifact that can be either software-based, such as a virtual server or a custom software program, or hardware-based, such as a physical server or a network device.

1.1.2. *Cloud services.* Any IT resource that is made remotely accessible via a cloud. The driving motivation behind cloud computing is to provide IT resources as **services** that encapsulate other IT resources, while offering functions for clients to use and leverage remotely. As a distinct and remotely accessible environment, a cloud represents an option for the deployment of IT resources. In contrast IT resources hosted in a conventional IT enterprise, within an organizational boundary, are considered **on-premise**. An IT resource that is on-premise cannot be cloud-based, and vice-versa.

- an on-premise IT resource can access and interact with a cloud-based IT resource
- an on-premise IT resource can be moved to a cloud, thereby changing it to a cloud-based IT resource
- redundant deployments of an IT resource can exist in both on-premise and cloud-based environments

1.1.3. *Scaling.* From an IT resource perspective, represents the ability of the IT resource to handle increased or decreased usage demands. The allocation or release of IT resources that are of the same type is referred to as **horizontal scaling**. The horizontal allocation of resources is referred to as scaling out and the horizontal releasing of resources is referred to as scaling in. Horizontal scaling is a common form of scaling within cloud environments. Instead, when an existing IT resource is replaced by another with higher or lower capacity, we refer to **vertical scaling**. Specifically, the replacing of an IT resource with another that has a higher capacity is referred to as scaling up and the replacing an IT resource with another that has a lower capacity is considered scaling down. Vertical scaling is less common in cloud environments due to the downtime required while the replacement is taking place.

1.2. **Business drivers.** The most common economic rationale for investing in cloud-based IT resources is in the reduction or outright elimination of up-front IT investments, namely hardware and software purchases and ownership costs. This elimination or minimization of up-front financial commitments allows enterprises to start small and accordingly increase IT resource allocation as required. The same rationale applies to operating systems, middleware or platform software, and application software. Pooled IT resources are made available to and shared by multiple cloud consumers, resulting in increased or even maximum possible utilization. By providing pools of IT resources, along with tools and technologies designed to leverage them collectively, clouds can instantly and dynamically allocate IT resources to cloud consumers, on-demand or via the cloud consumer's direct configuration.

- **on-demand** access to pay-as-you-go computing resources on a short-term basis (such as processors by the hour), and the ability to release these computing resources when they are no longer needed.
- zero capital expenditure necessary to get started
- the ability to add or remove IT resources at a fine-grained level
- service accessible through a web browser or a web API

A hallmark of the typical cloud environment is its intrinsic ability to provide extensive support for increasing the availability of a cloud-based IT resource to minimize or even eliminate outages, and for increasing its **reliability** so as to minimize the impact of runtime failure conditions.

1.2.1. *Capacity planning.* The process of determining and fulfilling future demands of an organization's IT resources, products, and services. Within this context, capacity represents the maximum amount of work that an IT resource is capable of delivering in a given period of time. A discrepancy between the capacity of an IT resource and its demand can result in a system becoming either inefficient (**over-provisioning**) or unable to fulfill user needs (**under-provisioning**). **Capacity planning** is focused on minimizing this discrepancy to achieve predictable efficiency and performance. Cloud services prefer **lag strategy**: adding capacity when the IT resource reaches its full capacity. Planning for capacity can be challenging because it requires estimating usage load fluctuations. There is a constant need to balance peak usage requirements without unnecessary over-expenditure on infrastructure.

1.2.2. *Cost reduction.* A direct alignment between IT costs and business performance can be difficult to maintain. The growth of IT environments often corresponds to the assessment of their maximum usage requirements. Two costs need to be accounted for: the cost of acquiring new infrastructure, and the cost of its ongoing ownership. Operational overhead represents a considerable share of IT budgets, often exceeding up-front investment costs.

1.2.3. *Organizational agility.* Businesses need the ability to adapt and evolve to successfully face change caused by both internal and external factors. Organizational agility is the measure of an organization's responsiveness to change. An IT enterprise often needs to respond to business change by scaling its IT resources beyond the scope of what was previously predicted or planned for. Due to a lack of reliability controls within the infrastructure, responsiveness to consumer or customer requirements may be reduced to a point whereby a business' overall continuity is threatened. On a broader scale, the up-front investments and infrastructure ownership costs that are required to enable new or expanded business automation solutions may themselves be prohibitive enough for a business to settle for IT infrastructure of less-than-ideal quality, thereby decreasing its ability to meet real-world requirements.

1.3. **Cloud characteristics.** An IT environment requires a specific set of characteristics to enable the remote provisioning of scalable and measured IT resources in an effective manner. These characteristics need to exist to a meaningful extent for the IT environment to be considered an effective cloud.

1.3.1. *On-demand self-service.* A cloud consumer can unilaterally access cloud-based IT resources giving the cloud consumer the freedom to self-provision these IT resources. Once configured, usage of the self-provisioned IT resources can be automated, requiring no further human involvement by the cloud consumer or cloud provider. This **on-demand** service usage relies on

- orchestration technologies, e.g. OpenStack and Kubernetes
- web interface, e.g. OpenStack, AWS, Google Cloud
- dedicated shell, e.g. AWS shell
- programming API, e.g. AWS SDK for Java, Python, .Net, Ruby, Go

1.3.2. *Broad network access.* Represents the ability for a cloud service to be widely accessible from the internet using different devices. Relies on broadband network access and internet technologies.

1.3.3. *Multitenancy and resource pooling.* The characteristic of a software program that enables an instance of the program to serve different consumers (**tenants**) whereby each is isolated from the other, is referred to as **multitenancy**. A cloud provider pools its IT resources to serve multiple cloud service consumers by using multitenancy models that frequently rely on the use of **virtualization** technologies. Through the use of multitenancy technology, IT resources can be dynamically assigned and reassigned, according to cloud service consumer demands. **Resource pooling** allows cloud providers to pool large-scale IT resources to serve multiple cloud consumers.

1.3.4. *Rapid elasticity.* The automated ability of a cloud to transparently scale IT resources, as required in response to runtime conditions or as pre-determined by the cloud consumer or cloud provider.

1.3.5. *Measured usage.* The ability of a cloud platform to keep track of the usage of its IT resources, primarily by cloud consumers. Based on what is measured, the cloud provider can charge a cloud consumer only for the IT resources actually used and/or for the timeframe during which access to the IT resources was granted. In this context, measured usage is closely related to the on-demand characteristic.

1.3.6. *Resiliency.* Computing is a form of failover that distributes redundant implementations of IT resources across physical locations. IT resources can be preconfigured so that if one becomes deficient, processing is automatically handed over to another redundant implementation. Within cloud computing, the characteristic of **resiliency** can refer to redundant IT resources within the same cloud (but in different physical locations) or across multiple clouds.

1.4. **Risks and challanges.** Several of the most critical cloud computing challenges pertaining mostly to cloud consumers that use IT resources located in public clouds.

1.4.1. *Increased security vulnerabilities.* The moving of business data to the cloud means that the responsibility over data security becomes shared with the cloud provider. There can be overlapping trust boundaries from different cloud consumers due to the fact that cloud-based IT resources are commonly shared. Furthermore, another consequence of overlapping trust boundaries relates to the cloud provider's privileged access to cloud consumer data. The extent to which the data is secure is now limited to the security controls and policies applied by both the cloud consumer and cloud provider. The overlapping of trust boundaries and the increased exposure of data can provide malicious cloud consumers (human and automated) with greater opportunities to attack IT resources and steal or damage business data.

1.4.2. *Reduced Operational Governance Control.* Cloud consumers are usually allotted a level of governance control that is lower than that over on-premise IT resources. This can introduce risks associated with how the cloud provider operates its cloud, as well as the external connections that are required for communication between the cloud and the cloud consumer. Legal contracts, when combined with SLAs, technology inspections, and monitoring, can mitigate governance risks and issues. A cloud governance system is established through SLAs, given the "as-a-service" nature of cloud computing. A cloud consumer must keep track of the actual service level being offered and the other warranties that are made by the cloud provider.

1.4.3. *Limited Portability Between Cloud Providers.* Due to a lack of established industry standards within the cloud computing industry, public clouds are commonly proprietary to various extents. For cloud consumers that have custom-built solutions with dependencies on these proprietary environments, it can be challenging to move from one cloud provider to another.

1.4.4. *Multi-Regional Compliance and Legal Issues.* Third-party cloud providers will frequently establish data centers in affordable or convenient geographical locations. Cloud consumers will often not be aware of the physical location of their IT resources and data when hosted by public clouds. For some organizations, this can pose serious legal concerns pertaining to industry or government regulations that specify data privacy and storage policies. Another potential legal issue pertains to the accessibility and disclosure of data. Countries have laws that require some types of data to be disclosed to certain government agencies or to the subject of the data.

1.5. **Main roles.** Organizations and humans can assume different types of pre-defined roles depending on how they relate to and/or interact with a cloud and its hosted IT resources.

1.5.1. *Cloud provider.* The organization that provides cloud-based IT resources is the cloud provider. When assuming the role of cloud provider, an organization is responsible for making cloud services available to cloud consumers, as per agreed upon SLA guarantees. The cloud provider is further tasked with any required management and administrative duties to ensure the on-going operation of the overall cloud infrastructure.

1.5.2. *Cloud consumer.* An organization (or a human) that has a formal contract or arrangement with a cloud provider to use IT resources made available by the cloud provider. Specifically, the cloud consumer uses a cloud service consumer to access a cloud service

1.5.3. *Cloud service owner.* The person or organization that legally owns a cloud service is called a cloud service owner. The cloud service owner can be the cloud consumer, or the cloud provider that owns the cloud within which the cloud service resides. Note that a cloud consumer that owns a cloud service hosted by a third-party cloud does not necessarily need to be the user (or consumer) of the cloud service.

1.5.4. *Cloud Resource Administrator.* The person or organization responsible for administering a cloud-based IT resource (including cloud services). The cloud resource administrator can be (or belong to) the cloud consumer or cloud provider of the cloud within which the cloud service resides. Alternatively, it can be (or belong to) a third-party organization contracted to administer the cloud-based IT resource.

1.5.5. *Cloud Auditor.* A third-party (often accredited) that conducts independent assessments of cloud environments assumes the role of the cloud auditor. The typical responsibilities associated with this role include the evaluation of security controls, privacy impacts, and performance.

1.5.6. *Cloud Broker.* This role is assumed by a party that assumes the responsibility of managing and negotiating the usage of cloud services between cloud consumers and cloud providers. Mediation services provided by cloud brokers include service intermediation, aggregation, and arbitrage.

1.5.7. *Cloud Carrier.* The party responsible for providing the wire-level connectivity between cloud consumers and cloud providers assumes the role of the cloud carrier. This role is often assumed by network and telecommunication providers.

1.6. **Cloud delivery models.** A cloud delivery model represents a specific, pre-packaged combination of IT resources offered by a cloud provider.

1.6.1. *Infrastructure-as-a-Service.* The **IaaS** delivery model represents a self-contained IT environment comprised of infrastructure-centric IT resources that can be accessed and managed via cloud service-based interfaces and tools. In contrast to traditional hosting or outsourcing environments, with IaaS, IT resources are typically virtualized and packaged into bundles that simplify up-front runtime scaling and customization of the infrastructure. The general purpose of an IaaS environment is to provide cloud consumers with a high level of control and responsibility over its configuration and utilization. Sometimes cloud providers will contract IaaS offerings from other cloud providers in order to scale their own cloud environments.

1.6.2. *Platform-as-a-Service.* The **PaaS** delivery model represents a pre-defined "ready-to-use" environment typically comprised of already deployed and configured IT resources. Specifically, PaaS relies on (and is primarily defined by) the usage of a ready-made environment that establishes a set of pre-packaged products and tools used to support the entire delivery lifecycle of custom applications. By working within a ready-made platform, the cloud consumer is spared the administrative burden of setting up and maintaining the bare infrastructure IT resources provided via the IaaS model. Conversely, the cloud consumer is granted a lower level of control over the underlying IT resources that host and provision the platform.

1.6.3. *Software-as-a-Service.* A software program positioned as a shared cloud service and made available as a "product" or generic utility represents the typical profile of a **SaaS** offering. The SaaS delivery model is typically used to make a reusable cloud service widely available (often commercially) to a range of cloud consumers. A cloud consumer is generally granted very limited administrative control over a SaaS implementation. It is most often provisioned by the cloud provider, but it can be legally owned by whichever entity assumes the cloud service owner role.

1.7. **Cloud deployment models.**

A cloud deployment model represents a specific type of cloud environment, primarily distinguished by ownership, size, and access.

1.7.1. *Public clouds.* A publicly accessible cloud environment owned by a third-party cloud provider. The IT resources on public clouds are usually provisioned via the previously described cloud delivery models and are generally offered to cloud consumers at a cost or are commercialized via other avenues.

1.7.2. *Community clouds.* Similar to a public cloud, except that its access is limited to a specific community of cloud consumers. The community cloud may be jointly owned by the community members or by a third-party cloud provider that provisions a public cloud with limited access. The member cloud consumers of the community typically share the responsibility for defining and evolving the community cloud

1.7.3. *Private clouds.* Enable an organization to use cloud computing technology as a means of centralizing access to IT resources by different parts, locations, or departments of the organization. When a private cloud exists as a controlled environment, the problems described in "Risks and challenges" section do not tend to apply. The same organization is technically both the cloud consumer and cloud provider.

- a separate organizational department typically assumes the responsibility for provisioning the cloud (and therefore assumes the cloud provider role)

- departments requiring access to the private cloud assume the cloud consumer role

1.7.4. *Hybrid Clouds.* An environment comprised of two or more different cloud deployment models. For example, a cloud consumer may choose to deploy cloud services processing sensitive data to a private cloud and other, less sensitive cloud services to a public cloud. Hybrid deployment architectures can be complex and challenging to create and maintain due to the potential disparity in cloud environments and the fact that management responsibilities are typically split between the private cloud provider organization and the public cloud provider.

2. Cloud-enabling technology

Modern-day clouds are underpinned by a set of primary technology components that collectively enable key features and characteristics associated with contemporary cloud computing.

2.1. **Data center technology.** Grouping IT resources in close proximity with one another, rather than having them geographically dispersed, allows for power sharing, higher efficiency in shared IT resource usage, and improved accessibility for IT personnel. These are the advantages that naturally popularized the data center concept. Modern data centers exist as specialized IT infrastructure used to house centralized IT resources, such as servers, databases, networking and telecommunication devices, and software systems.

2.1.1. *Virtualization.* Another core technology for cloud computing. It encompasses a collection of solutions allowing the abstraction of some of the fundamental elements for computing, such as hardware, storage, and networking. Virtualization is essentially a technology that allows creation of different computing environments. These environments are called virtual because they simulate the interface that is expected by a guest. **Hardware virtualization** allows the coexistence of different software stacks on top of the same hardware. These stacks are contained inside **virtual machine instances**, which operate in isolation from each other. Virtualization technologies are also used to replicate runtime environments for programs. Applications in the case of **process virtual machines**, instead of being executed by the operating system, are run by a specific program called a virtual machine. This technique allows isolating the execution of applications and providing a finer control on the resource they access. Process virtual machines offer a higher level of abstraction with respect to hardware virtualization, since the guest is only constituted by an application rather than a complete software stack.

2.1.2. *Automation.* Autonomic computing refers to the ability of a computer system to self-manage, which includes the following capabilities:

- **self-configuration**: ability to accommodate varying and possibly unpredictable conditions
- **self-healing**: ability to remain functioning when problems arise
- **self-protection**: ability to detect threats and take appropriate actions
- **self-optimization**: constant monitoring for optimal operation

Autonomic systems are commonly modeled as closed-loop control systems where sensors monitor the external conditions and feed the collected data back to the decision logic. The aim is to have systems that can self-run while adapting to increasing system complexity, without the need for any user input. These systems can have high levels of built-in artificial intelligence that remain hidden from the users. Autonomic computing supports several cloud computing characteristics, including:

- elasticity: autonomic systems can monitor usage conditions and leverage cloudbased IT resources to automatically acquire and free IT resources as needed for the purpose of maintaining required service levels
- resiliency: autonomic systems can automatically detect unavailable IT resources and self-respond to allocate alternative IT resources as required

2.1.3. *High availability.* Since any form of data center outage significantly impacts business continuity for the organizations that use their services, data centers are designed to operate with increasingly higher levels of redundancy to sustain availability. Data centers usually have redundant, uninterruptable power supplies, cabling, and environmental control subsystems in anticipation of system failure, along with communication links and clustered hardware for load balancing.

2.1.4. *Computing hardware.* Much of the heavy processing in data centers is often executed by standardized commodity servers that have substantial computing power and storage capacity. Several computing hardware technologies are integrated into these modular servers, such as:

- support for different hardware processing architectures, such as x86 and RISCs
- redundant and hot-swappable components, such as hard disks, power supplies, network interfaces, and storage controller cards

With a properly established management console, a single operator can oversee hundreds to thousands of physical servers, virtual servers, and other IT resources.

2.1.5. *Storage hardware.* Data centers have specialized storage systems that maintain enormous amounts of digital information in order to fulfill considerable storage capacity needs. These storage systems are containers housing numerous hard disks that are organized into arrays. Storage systems usually involve the following technologies:

- hard disk arrays, which inherently divide and replicate data among multiple physical drives, and increase performance and redundancy by including spare disks. This technology is often implemented using Redundant Arrays of Independent Disks (RAID) schemes
- I/O caching, generally performed through hard disk array controllers, which enhance disk access times and performance
- hot-swappable hard disks, which can be safely removed from arrays without requiring prior powering down
- storage virtualization, realized through the use of virtualized hard disks and storage sharing
- fast data replication mechanisms, including snapshotting, which is saving a virtual machine's memory into a hypervisor-readable file for future reloading, and volume cloning, which is copying virtual or physical hard disk volumes and partitions

Storage systems encompass tertiary redundancies, such as robotized tape libraries, which are used as backup and recovery systems that typically rely on removable media. This type of system can exist as a networked IT resource or direct-attached storage (DAS). In the former case, the storage system is connected to one or more IT resources through a network. Networked storage devices usually fall into one of the following categories:

- Storage Area Network (SAN). Physical data storage media are connected through a dedicated network and provide block-level data storage access using industry standard protocols
- Network-Attached Storage (NAS). Hard drive arrays are contained and managed by this dedicated device, which connects through a network and facilitates access to data using file-centric data access protocols like the Network File System (NFS) or Server Message Block (SMB).

NAS, SAN, and other more advanced storage system options provide fault tolerance in many components through controller redundancy, cooling redundancy, and hard disk arrays that use RAID storage technology.

2.2. **Web technology.** The web is the primary interface through which cloud computing delivers its services. At present, the web encompasses a set of technologies and services that facilitate interactive information sharing, collaboration, user-centered design, and application composition. This evolution has transformed the web into a rich platform for application development and is known as **web 2.0**, a new way in which developers architect applications and deliver services through the Internet and provides new experience for users of these applications and services. Web 2.0 brings interactivity and flexibility into web pages, providing enhanced user experience. Furthermore, applications can be "synthesized" simply by composing existing services and integrating them, thus providing added value.

2.3. **Service-Oriented Computing.** Service orientation is the core reference model for cloud computing systems. This approach adopts the concept of services as the main building blocks of application and system development. Service-Oriented Computing (**SOC**) supports the development of rapid, low-cost, flexible, interoperable, and evolvable applications and systems. A service is an abstraction representing a self-describing and platform-agnostic component that can perform any function. A service is supposed to be loosely-coupled, reusable, programming language independent, and location transparent. Loose coupling allows services to serve different scenarios more easily and makes them reusable. Independence from a specific platform increases services accessibility.

3. Principles of distributed computing

Distributed computing studies the models, architectures, and algorithms used for building and managing distributed systems. As a general definition of the term distributed system, we use the one proposed by Tanenbaum et. al: "A distributed system is a collection of independent computers that appears to its users as a single coherent system". Communication is another fundamental aspect of distributed computing.A distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages.

A distributed system is the result of the interaction of several components that traverse the entire computing stack from hardware to software. At the very bottom layer, computer and network hardware constitute the physical infrastructure; these components are directly managed by the operating system, which provides the basic services for interprocess communication (**IPC**), process scheduling and management, and resource management in terms of file system and local devices. Taken together these two layers become the platform on top of which specialized software is deployed to turn a set of networked computers into a distributed system. The use of well-known standards at the operating system level and even more at the hardware and network levels allows easy harnessing of heterogeneous components and their organization into a coherent and uniform 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 of the distributed system stack is represented by the applications and services designed and developed to use the middleware. These can serve several purposes and often expose their features in the form of graphical user interfaces accessible locally or through the Internet via a web browser.

3.1. **Software architectural styles.** Software architectural styles are based on the logical arrangement of software components. They are helpful because they provide an intuitive view of the whole system, despite its physical deployment. They also identify the main abstractions that are used to shape the components of the system and the expected interaction patterns between them. These models constitute the foundations on top of which distributed systems are designed from a logical point of view.

3.1.1. *Call-and-return architecture.* This category identifies all systems that are organised into components mostly connected together by method calls. The activity of systems modeled in this way is characterized by a chain of method calls whose overall execution and composition identify the execution of one or more operations. The internal organization of components and their connections may vary.

- **top-down** style. This architectural style is quite representative of systems developed with imperative programming, which leads to a divide-and-conquer approach to problem resolution. Systems developed according to this style are composed of one large main program that accomplishes its tasks by invoking subprograms or procedures. The components in this style are procedures and subprograms, and connections are method calls or invocation. The calling program passes information with parameters and receives data from return values or parameters. Method calls can also extend beyond the boundary of a single process by leveraging techniques for remote method invocation, such as remote procedure call (**RPC**) and all its descendants. The overall structure of the program execution at any point in time is characterized by a tree, the root of which constitutes the main function of the principal program. This architectural style is quite intuitive from a design point of view but hard to maintain and manage in large systems
- **layered** style. Each layer generally operates with at most two layers: the one that provides a lower abstraction level and the one that provides a higher abstraction layer. Specific protocols and interfaces define how adjacent layers interact. It is possible to model such systems as a stack of layers, one for each level of abstraction. Therefore, the components are the layers and the connectors are the interfaces and protocols used between

adjacent layers. A user or client generally interacts with the layer at the highest abstraction, which, in order to carry its activity, interacts and uses the services of the lower layer. This process is repeated (if necessary) until the lowest layer is reached. It is also possible to have the opposite behavior: events and callbacks from the lower layers can trigger the activity of the higher layer and propagate information up through the stack. The advantages of the layered style are that, as happens for the object-oriented style, it supports a modular design of systems and allows us to decompose the system according to different levels of abstractions by encapsulating together all the operations that belong to a specific level. Layers can be replaced as long as they are compliant with the expected protocols and interfaces, thus making the system flexible. The main disadvantage is constituted by the lack of extensibility, since it is not possible to add layers without changing the protocols and the interfaces between layers. This also makes it complex to add operations

3.1.2. *Independent components based.* This class of architectural style models systems in terms of independent components that have their own life cycles, which interact with each other to perform their activities.

- **communicating processes**. In this architectural style, components are represented by independent processes that leverage IPC facilities for coordination management. This is an abstraction that is quite suitable to modeling distributed systems that, being distributed over a network of computing nodes, are necessarily composed of several concurrent processes. Each of the processes provides other processes with services and can leverage the services exposed by the other 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 systems**. In this architectural style, the 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 register. In general, other components provide a callback that will be executed when the event is activated. During the activity of a component, a specific runtime condition can activate one of the exposed events, thus triggering the execution of the callbacks registered with it. Event activation may be accompanied by contextual information that can be used in the callback to handle the event. This information can be passed as an argument to the callback or by using some shared repository between components. The main advantage of such an architectural style is that it fosters the development of open systems: new modules can be added and easily integrated into the system as long as they have compliant interfaces for registering to the events. This architectural style solves some of the limitations observed for the top-down and object-oriented styles. First, the invocation pattern is implicit, and the connection between the caller and the callee is not hard-coded; this gives a lot of flexibility since addition or removal of a handler to events can be done without changes in the source code of applications. Second, the event source does not need to know the identity of the event handler in order to invoke the callback. The disadvantage of such a style is that it relinquishes

control over system computation. When a component triggers an event, it does not know how many event handlers will be invoked and whether there are any registered handlers. This information is available only at runtime and, from a static design point of view, becomes more complex to identify the connections among components and to reason about the correctness of the interactions

3.1.3. *Microservices.* **Microservice**s is an architectural style that structures an application as a collection of services that are

- highly maintainable and testable
- loosely coupled
- independently deployable
- organized around business capabilities
- owned by a small team

The microservice architecture enables rapid, frequent and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack. Keep in mind that the microservice architecture is not a silver bullet. It has several drawbacks. Moreover, when using this architecture there are numerous issues that you must address. Initially a monolithic architecture has benefits like:

- simple to develop, easy to make radical changes to the application
- straightforward to test and to deploy
- easy to scale

But over time, as complexity increases, development, testing, deployment, and scaling became much more difficult. Switching to a microservice achitecture has several benefits:

- it enables the continuous delivery and deployment of large, complex applications
- services are small and easily maintained
- services are independently deployable and independently scalable
- it enables different teams to be autonomous
- it allows easy experimenting and adoption of new technologies
- it has better fault isolation

However, no technology is a silver bullet, and the microservice architecture has a number of significant drawbacks and issues:

- finding the right set of services is challenging
- distributed systems are complex, which makes development, testing, and deployment difficult
- deploying features that span multiple services requires careful coordination
- deciding when to adopt the microservice architecture is difficult

3.2. **System architectural styles.** System architectural styles cover the physical organization of components and processes over a distributed infrastructure. They provide a set of reference models for the deployment of such systems and help engineers not only have a common vocabulary in describing the physical layout of systems but also quickly identify the major advantages and drawbacks of a given deployment and whether it is applicable for a specific class of applications.

3.2.1. *Client/server.* These two components interact with each other through a network connection using a given protocol. The communication is unidirectional: The client issues a request to the server, and after processing the request the server returns a response. There could be multiple client components issuing requests to a server that is passively waiting for them. Hence, the important operations in the client-server paradigm are request, accept (client side), and listen and response (server side). 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: the server. In general, multiple clients are interested in such services and the server must be appropriately designed to efficiently serve requests coming from different clients. This consideration has impli- cations on both client design and server design.

3.2.2. *Peer-to-peer.* Introduces a symmetric architecture in which all the components, called peers, play the same role and incorporate both client and server capabilities of the client/server model. 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. With respect to the client/server model that partitions the responsibilities of the IPC between server and clients, the peer-to-peer model attributes the same responsibilities to each component. Therefore, this model is quite suitable for highly decentralized architecture, which can scale better along the dimension of the number of peers. The disadvantage of this approach is that the management of the implementation of algorithms is more complex than in the client/server model.

3.3. **Models for interprocess communication - message passing.** Distributed systems are composed of a collection of concurrent processes interacting with each other by means of a network connection. Therefore, IPC is a fundamental aspect of distributed systems design and implementation. IPC is used to either exchange data and information or coordinate the activity of processes. IPC is what ties together the different components of a distributed system, thus making them act as a single system. There are several different models in which processes can interact with each other; these map to different abstractions for IPC. At a lower level, IPC is realized through the fundamental tools of network programming. Sockets are the most popular IPC primitive for implementing communication channels between distributed processes. They facilitate interaction patterns that, at the lower level, mimic the client/server abstraction and are based on a request-reply communication model.

The abstraction of message has played an important role in the evolution of the models and technologies enabling distributed computing. Couloris et al. define a distributed system as "one in which components located at networked computers communicate and coordinate their actions only by passing messages". The term message, in this case, identifies any discrete amount of information that is passed from one entity to another. Several distributed programming paradigms eventually use message-based communication despite the abstractions that are presented to developers for programming the interaction of distributed components.

3.3.1. *Remote Procedure Call.* This paradigm 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, underlying client/server architecture is implied. The server process maintains a 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. 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.

An important aspect of RPC is **marshaling**, which identifies the process of converting parameter 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. Marshaling and unmarshaling are performed by the RPC runtime infrastructure, and the client and server user code does not necessarily have to perform these tasks. The RPC runtime, on the other hand, is not only responsible for parameter packing and unpacking but also for handling the request-reply interaction that happens between the client and the server process in a completely transparent manner. Therefore, developing a system leveraging RPC for IPC consists of the following steps:

(1) design and implementation of the server procedures that will be exposed for remote invocation
(2) registration of remote procedures with the RPC server on the node where they will be made available
(3) design and implementation of the client code that invokes the remote procedure(s)

3.3.2. *Distributed Object framework.* Distributed object frameworks extend object-oriented programming systems by allowing objects to be distributed across a heterogeneous network and provide facilities so that they can coherently act as though they were in the same address space. Distributed object frameworks leverage the basic 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. With respect to the RPC model, the infrastructure manages instances that are exposed through well-known interfaces instead of procedures. Therefore, the common interaction pattern is the following:

(1) the server process maintains a registry of active objects that are made available to other processes. According to the specific implementation, active objects can be published using interface definitions or class definitions
(2) the client process, by using a given addressing scheme, obtains a reference to the active remote object. This reference is represented by a pointer to an instance that is of a shared type of interface and class definition
(3) the client process invokes the methods on the active object by calling them through the reference previously obtained. Parameters and return values are marshaled as happens in the case of RPC

Distributed object frameworks give the illusion of interaction with a local instance while invoking remote methods. 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. This is an important aspect to take into account in designing distributed object systems, because it might lead to inconsistencies. An alternative to this standard process, which is called marshaling by value, is **marshaling by reference**. In this second case the object instance is not duplicated and a proxy of it is created on the server side (for parameters) or the client side (for return values). Marshaling by reference is a more complex technique and generally puts more burden on the runtime infrastructure since remote refer- ences 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.

3.3.3. *Service-oriented computing.* **Service-oriented computing** organizes distributed systems in terms of services, which represent the major abstraction for building systems. Service orientation expresses applications and software systems as aggregations of services that are coordinated within a service-oriented architecture (**SOA**). Even though there is no designed technology for the development of service-oriented software systems, Web services are the de facto approach for developing SOA. Web services, the fundamental component enabling cloud computing systems, leverage the Internet as the main interaction channel between users and the system.

A **service** encapsulates a software component that provides a set of coherent and related functionalities that can be reused and integrated into bigger and more complex applications. The term service is a general abstraction that encompasses several different implementations using different technologies and protocols, usually identified by four major characteristics:

- boundaries are explicit. A service-oriented application is generally composed of services that are spread across different domains, trust authorities, and execution environments. Generally, crossing such boundaries is costly; therefore, service invocation is explicit by design and often leverages message passing. With respect to distributed object programming, whereby remote method invocation is transparent, in a service-oriented computing environment the interaction with a service is explicit and the interface of a service is kept minimal to foster its reuse and simplify the interaction
- services are autonomous. Services are components that exist to offer functionality and are aggregated and coordinated to build more complex system. They are not designed to be part of a specific system, but they can be integrated in several software systems, even at the same time. With respect to object orientation, which assumes that the deployment of applications is atomic, service orientation considers this case an exception rather than the rule and puts the focus on the design of the service as an autonomous component. The notion of autonomy also affects the way services handle failures. Services operate in an unknown environment and interact with third-party applications. Therefore, minimal assumptions can be made concerning such environments: applications may fail without notice, messages can be malformed, and clients can be unauthorized
- services share schema and contracts, not class or interface definitions. Services are not expressed in terms of classes or interfaces, as happens in

object-oriented systems, but they define themselves in terms of schemas and contracts. A service advertises a contract describing the structure of messages it can send and/or receive and additional constraint - if any - on their ordering. Because they are not expressed in terms of types and classes, services are more easily consumable in wider and heterogeneous environments. At the same time, a service orientation requires that contracts and schema remain stable over time, since it would be possible to propagate changes to all its possible clients. To address this issue, contracts and schema are defined in a way that allows services to evolve without breaking already deployed code
- services compatibility is determined based on policy. Service orientation separates structural compatibility from semantic compatibility. Structural compatibility is based on contracts and schema and can be validated or enforced by machine-based techniques. Semantic compatibility is expressed in the form of policies that define the capabilities and requirements for a service. Policies are organized in terms of expressions that must hold true to enable the normal operation of a service

Web service technology provides an implementation of the RPC concept over HTTP, thus allowing the interaction of components that are developed with different technologies. A Web service is exposed as a remote object hosted on a Web server, and method invocations are transformed in HTTP requests, opportunely packaged using specific protocols such as Simple Object Access Protocol (**SOAP**) or Representational State Transfer (**REST**).

3.4. **Service-oriented architecture.** SOA is an architectural style supporting service orientation. It organizes a software system into a collection of interacting services. SOA encompasses a set of design principles that structure system development and provide means for integrating components into a coherent and decentralized system. SOA-based computing packages functionalities into a set of interoperable services, which can be integrated into different software systems belonging to separate business domains. There are two major roles within SOA: the **service provider** and the **service consumer**. The service provider is the maintainer of the service and the organization that makes available one or more services for others to use. To advertise services, the provider can publish them in a registry, together with a service contract that specifies the nature of the service, how to use it, the requirements for the service, and the fees charged. The service consumer can locate the service metadata in the registry and develop the required client components to bind and use the service. Service providers and consumers can belong to different organization bodies or business domains. It is very common in SOA-based computing systems that components play the roles of both service provider and service consumer. Services might aggregate information and data retrieved from other services or create workflows of services to satisfy the request of a given service consumer. This practice is known as **service orchestration**, which more generally describes the automated arrangement, coor- dination, and management of complex computer systems, middleware, and services. Another important interaction pattern is **service choreography**, which is the coordinated interaction of services without a single point of control. SOA provides a reference model for architecting several software systems, especially enterprise business applications and systems. In this context, interoperability, standards, and service contracts play a fundamental role. In particular, the following guiding principles are recommended:

- standardized service contract. Services adhere to a given communication agreement, which is specified through one or more service description documents
- loose coupling. Services are designed as self-contained components, maintain relationships that minimize dependencies on other services, and only require being aware of each other. Service contracts will enforce the required interaction among services. This simplifies the flexible aggregation of services and enables a more agile design strategy that supports the evolution of the enterprise business
- abstraction. A service is completely defined by service contracts and description documents. They hide their logic, which is encapsulated within their implementation. The use of service description documents and contracts removes the need to consider the technical implementation details and provides a more intuitive framework to define software systems within a business context
- reusability. Designed as components, services can be reused more effectively, thus reducing development time and the associated costs. Reusability allows for a more agile design and cost-effective system implementation and deployment. Therefore, it is possible to leverage third-party services to deliver required functionality by paying an appropriate fee rather developing the same capability in-house.
- autonomy. Services have control over the logic they encapsulate and, from a service consumer point of view, there is no need to know about their implementation
- lack of state. By providing a stateless interaction pattern (at least in principle), services increase the chance of being reused and aggregated, especially in a scenario in which a single service is used by multiple consumers that belong to different administrative and business domains
- discoverability. Services are defined by description documents that constitute supplemental metadata through which they can be effectively discovered. Service discovery provides an effective means for utilizing third-party resources
- composability. Using services as building blocks, sophisticated and complex operations can be implemented. Service orchestration and choreography provide a solid support for composing services and achieving business goals

SOA can be realized through several technologies, but nowadays, SOA is mostly realized through Web services technology, which provides an interoperable platform for connecting systems and applications.

3.4.1. *Web services.* Web services are the prominent technology for implementing SOA systems and applications. They leverage Internet technologies and standards for building distributed systems. Several aspects make Web services the technology of choice for SOA. First, they allow for interoperability across different platforms and programming languages. Second, they are based on well-known and vendor-independent standards such as HTTP, XML and JSON. Third, they provide an intuitive and simple way to connect heterogeneous software systems, enabling the quick composition of services in a distributed environment. Finally, they provide the features required by enterprise business applications to be used in an industrial 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.

- SOAP structures the interaction in terms of messages that are XML documents, and leverages the transport level, most commonly HTTP, for IPC. These messages can be used for method invocation and result retrieval. SOAP has often been considered quite inefficient because of the excessive use of markup that XML imposes for organizing the information into a well-formed document
- REST provides a model for designing network-based software systems utilizing the client/server model and leverages the facilities provided by HTTP for IPC without additional burden. In a **RESTful** system, a client sends a request over HTTP using the standard HTTP methods (PUT, GET, POST, and DELETE), and the server issues a response that includes the representation of the resource. By relying on this minimal support, it is possible to provide whatever it needed to replace the basic and most important functionality provided by SOAP, which is method invocation. Together with an appropriate URI organization to identify resources, all the atomic operations required by a Web service are implemented

REST provides a lightweight alternative to SOAP: data can be transmitted using XML, but often JSON is preferable, as part of the HTTP content. Therefore the additional markup required by SOAP is removed.

4. Virtualization

Virtualization is a large umbrella of technologies and concepts that are 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 interested recently due to the confluence of several phenomena:

- increased performance and computing capacity, underutilized hardware and software resources. Nowadays, the average end-user desktop PC is powerful enough to meet almost all the needs of everyday computing, with extra capacity that is rarely used. Moreover, if we consider the IT infrastructure of an enterprise, many computers are only partially utilized whereas they could be used without interruption on a 24/7/365 basis
- lack of space. The continuous need for additional capacity, whether storage or compute power, makes data centers grow quickly. In most cases enterprises cannot afford to build another data center to accommodate additional resource capacity. This condition, along with hardware underutilization, has led to the diffusion of a technique called server consolidation, for which virtualization technologies are fundamental
- greening initiatives. Maintaining a data center operation not only involves keeping servers on, but a great deal of energy is also consumed in keeping

them cool. Hence, reducing the number of servers through server consolidation will definitely reduce the impact of cooling and power consumption of a data center. Virtualization technologies can provide an efficient way of consolidating servers

- rise of administrative costs. Power consumption and cooling costs have now become higher than the cost of IT equipment. Moreover, the increased demand for additional capacity, which translates into more servers in a data center, is also responsible for a significant increment in administrative costs. Virtualization can help reduce the number of required servers for a given workload, thus reducing the cost of the administrative personnel

4.1. **Characteristics of virtual environments.** In a virtualized environment there are three major components: guest, host, and virtualization layer. 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. In the case of virtual storage, the guest might be client applications or users that interact with the virtual storage management software deployed on top of the real storage system. The case of virtual networking is also similar: The guest - applications and users - interacts with a virtual network, such as a virtual private network (VPN), which is managed by specific software (VPN client) using the physical network available on the node. The main common characteristic of all these different implementations is the fact that the virtual environment is created by means of a software program. The technologies of today allow profitable use of virtualization and make it possible to fully exploit the advantages that come with it, such as:

4.1.1. *Increased security.* The virtual machine represents an emulated environment in which the guest is executed. All the operations of the guest are generally performed against the virtual machine, which then translates and applies them to the host. This level of indirection allows the virtual machine manager to control and filter 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 naturally hidden without the need to install complex security policies. Increased security is a requirement when dealing with untrusted code. By default, the file system exposed by the virtual computer is completely separated from the one of the host machine. This becomes the perfect environment for running applications without affecting other users in the environment.

4.1.2. *Managed execution.* Virtualization of the execution environment not only allows increased security, but a wider range of features also can be implemented. In particular, sharing, aggregation, emulation, and isolation are the most relevant features.

- **sharing**. Virtualization allows the creation of a separate computing environments within the same host. In this way it is possible to fully exploit the capabilities of a powerful guest, which would otherwise be underutilized. Sharing is a particularly important feature in virtualized data centers, where this basic feature is used to reduce the number of active servers and limit power consumption
- **aggregation**. Not only is it possible to share physical resource among several guests, but virtualization also allows aggregation, which is the opposite process. A group of separate hosts can be tied together and represented to guests as a single virtual host. This function is naturally implemented in middleware for distributed computing, with a classical example represented by cluster management software, which harnesses the physical resources of a homogeneous group of machines and represents them as a single resource
- **emulation**. Guest programs are executed within an environment that is controlled by the virtualization layer, which ultimately is a program. For instance, a completely different environment with respect to the host can be emulated, thus allowing the execution of guest programs requiring specific characteristics that are not present in the physical host. This feature becomes very useful for testing purposes, where a specific guest has to be validated against different platforms or architectures and the wide range of options is not easily accessible during development. Old and legacy software that does not meet the requirements of current systems can be run on emulated hardware without any need to change the code. This is possible either by emulating the required hardware architecture or within a specific operating system sandbox
- **isolation**. Virtualization allows providing guests—whether they are operating systems, applications, or other entities—with a completely separate environment, in which they are executed. The guest program performs its activity by interacting with an abstraction layer, which provides access to the underlying resources. Isolation brings several benefits; for example, it allows multiple guests to run on the same host without interfering with each other. Second, it provides a separation between the host and the guest. The virtual machine can filter the activity of the guest and prevent harmful operations against the host

4.1.3. *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. Virtual images are generally proprietary formats that require a specific virtual machine manager to be executed. In the case of programming-level virtualization, as implemented by the JVM or the .NET runtime, the binary code representing application components (jars or assemblies) can be run without any recompilation on any implementation of the corresponding virtual machine. This makes the application development cycle more flexible and application deployment very straightforward: One version of the application, in most cases, is able to run on different platforms with no changes. Finally, portability allows having your own system always with you and ready to use as long as the required virtual machine manager is available.

4.2. **Execution virtualization.** Execution virtualization includes all techniques that aim to emulate an execution environment that is separate from the one hosting the virtualization layer. All these techniques concentrate their interest on providing support for the execution of programs, whether these are the operating system, a binary specification of a program compiled against an abstract machine model, or an application. Therefore, execution virtualization can be implemented directly on top of the hardware by the operating system, an application, or libraries dynamically or statically linked to an application image.

4.2.1. *Machine reference model.* 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. From this perspective, virtualization techniques actually replace one of the layers and intercept the calls that are directed toward it. Modern computing systems can be expressed in terms of ISA, ABI and API layers. At the bottom layer, the model for the hardware is expressed in terms of the Instruction Set Architecture (**ISA**), which defines the instruction set for the processor, registers, memory, and interrupt management. ISA is the interface between hardware and software. The application binary interface (**ABI**) separates the operating system layer from the applications and libraries, which are managed by the OS. ABI covers details such as low-level data types, alignment, and call conventions and defines a format for executable programs. System calls are defined at this level. This interface allows portability of applications and libraries across operating systems that implement the same ABI. The highest level of abstraction is represented by the application programming interface (**API**), which interfaces applications to libraries and/or the underlying operating system.

Furthermore, the instruction set exposed by the hardware has been divided into different security classes that define who can operate with them. The first distinction can be made between privileged and nonprivileged instructions. **Nonprivileged instructions** are those instructions that can be used without interfering with other tasks because they do not access shared resources. This category contains e.g. all the floating, fixed-point, and arithmetic instructions. **Privileged instructions** are those that are executed under specific restrictions and are mostly used for sensitive operations, which expose (**behavior-sensitive**) or modify (**control-sensitive**) the privileged state. For instance, behavior-sensitive instructions are those that operate on the I/O, whereas control-sensitive instructions alter the state of the CPU registers. Some types of architecture feature more than one class of privileged instructions and implement a finer control of how these instructions can be accessed. For instance, a possible implementation features a hierarchy of privileges in the form of ring-based security: Ring 0, Ring 1, Ring 2, and Ring 3; **Ring 0** is in the most privileged level and **Ring 3** in the least privileged level. Ring 0 is used by the kernel of the OS, rings 1 and 2 are used by the OS-level services, and Ring 3 is used by the user. Recent systems support only two levels, with Ring 0 for supervisor mode and Ring 3 for user mode.

All the current systems support at least two different execution modes: **supervisor mode** and **user mode**. The first mode denotes an execution mode in which all the instructions (privileged and nonprivileged) 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.

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 nonprivileged instructions under other conditions. Conceptually, the **hypervisor** runs above the supervisor mode; in reality, hypervisors are run in supervisor mode, and the division between privileged and nonprivileged 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 emulate and manage the status of the CPU for guest operating systems. More recent implementations of ISA (Intel-VTx and AMD-V) have solved this problem by redesigning such sensitive instructions as privileged ones.

4.3. **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. Hardware-level virtualization is also called system virtualization, since it provides ISA to virtual machines, which is the representation of the hardware interface of a system. This is to differentiate it from process virtual machines, which expose ABI to virtual machines.

4.3.1. *Hypervisors.* A fundamental element of hardware virtualization is the hypervisor, or virtual machine manager (**VMM**). It recreates a hardware environment in which guest operating systems are installed. There are two major types of hypervisor:

- **type I** hypervisors run directly on top of the hardware. Therefore, they take the place of the operating systems and interact directly with the ISA interface exposed by the underlying hardware, and they emulate this interface in order to allow the management of guest operating systems. This type of hypervisor is also called a native virtual machine since it runs natively on hardware
- **type II** hypervisors require 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. This type of hypervisor is also called a hosted virtual machine since it is hosted within an operating system

Conceptually, a virtual machine manager is internally organized as three main modules, dispatcher, allocator, and interpreter. The **dispatcher** constitutes the entry point of the monitor and reroutes the instructions issued by the virtual machine instance to one of the two other modules. The **allocator** is responsible for deciding the system resources to be provided to the VM: whenever a virtual machine tries to execute an instruction that results in changing the machine resources associated with that VM, the allocator is invoked by the dispatcher. The **interpreter** module consists of interpreter routines. These are executed whenever a virtual machine executes a privileged instruction: a trap is triggered and the corresponding routine is executed. The design and architecture of a virtual machine manager, together with the underlying hardware design of the host machine, determine the full realization of hardware virtualization, where a guest operating system can be transparently executed on top of a VMM as though it were run on the underlying hardware. Three properties have to be met by a virtual machine manager to efficiently support virtualization:

- **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 virtual machine manager

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.

- **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. These extensions, which differ between the two vendors, are meant to reduce the performance penalties experienced by emulating x86 hardware with hypervisors. Today there exist many hardware-assisted solutions like Kernel-based Virtual Machine (KVM), VirtualBox, Xen, VMware and Hyper-V
- **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. 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. Whereas it is a desired goal for many virtualization solutions, full virtualization poses important concerns related to performance and technical implementation. A key challenge is the interception of privileged instructions such as I/O instructions: Since they change the state of the resources exposed by the host, they have to be contained within the virtual machine manager
- **paravirtualization**. This is a not-transparent virtualization solution that allows implementing thin virtual machine managers. 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. 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 allows a simpler implementation of virtual machine managers that have to simply transfer the execution of these operations, which were hard to virtualize, directly to the host. To take advantage of such an opportunity, guest operating systems need to be modified and explicitly ported by remapping the performance-critical operations through the virtual machine software interface. This technique has been successfully used by Xen for providing virtualization solutions for Linux-based operating systems specifically ported to run on Xen hypervisors. Operating systems that cannot be ported can still take advantage of paravirtualization by using adhoc device drivers that remap the execution of critical instructions to the paravirtualization APIs exposed by the hypervisor

- **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. An example of partial virtualization is address space virtualization used in time-sharing systems; this allows multiple applications and users to run concurrently in a separate memory space, but they still share the same hardware resources (disk, processor, and network)

4.3.2. *Operating system-level virtualization.* This offers the opportunity to create different and separated execution environments for applications that are managed concurrently. Differently from hardware virtualization, there is no virtual machine manager 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 and for limiting the impact of instances on each other. A user space instance in general contains a proper view of the file system, which is completely isolated, and separate IP addresses, software configurations, and access to devices. Operating systems supporting this type of virtualization are general-purpose, time-shared operating systems with the capability to provide stronger namespace and resource isolation. This virtualization technique can be considered an evolution of the chroot mechanism in Unix systems. The chroot operation changes the file system root directory for a process and its children to a specific directory. As a result, the process and its children cannot have access to other portions of the file system than those accessible under the new root directory. Because Unix systems also expose devices as parts of the file system, by using this method it is possible to completely isolate a set of processes. Following the same principle, operating system-level virtualization aims to provide separated and multiple execution containers for running applications. 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.

4.4. **Application-level virtualization.** Application-level virtualization is a technique allowing applications to be run in runtime environments that do not natively support all the features required by such applications. In this scenario, applications are not installed in the expected runtime environment but are run as though they were. In general, these techniques are mostly concerned with partial file systems, libraries, and operating system component emulation. Such emulation is performed by a thin layer - a program or an operating system component - that is in charge of executing the application. Emulation can also be used to execute program binaries compiled for different hardware architectures. In this case, one of the following strategies can be implemented:

- **interpretation**. In this technique every source instruction is interpreted by an emulator for executing native ISA instructions, leading to poor performance. Interpretation has a minimal startup cost but a huge overhead, since each instruction is emulated
- **binary translation**. In this technique every source instruction is converted to native instructions with equivalent functions. After a block of instructions is translated, it is cached and reused. Binary translation has a large initial overhead cost, but over time it is subject to better performance, since previously translated instruction blocks are directly executed

Emulation, as described, is different from hardware-level virtualization. The former simply allows the execution of a program compiled against a different hardware, whereas the latter emulates a complete hardware environment where an entire operating system can be installed. Application virtualization is a good solution in the case of missing libraries in the host operating system; in this case a replacement library can be linked with the application, or library calls can be remapped to existing functions available in the host system. Another advantage is that in this case the virtual machine manager is much lighter since it provides a partial emulation of the runtime environment compared to hardware virtualization. Moreover, this technique allows incompatible applications to run together.

5. Scaling mechanisms

5.1. **Autoscaling.**

5.1.1. *Dynamic scalability architecture.* An architectural model based on a system of predefined scaling conditions that trigger the dynamic allocation of IT resources from resource pools. Dynamic allocation enables variable utilization as dictated by usage demand fluctuations, since unnecessary IT resources are efficiently reclaimed without requiring manual interaction. The automated scaling listener is configured with workload thresholds that dictate when new IT resources need to be added to the workload processing. This mechanism can be provided with logic that determines how many additional IT resources can be dynamically provided, based on the terms of a given cloud consumer's provisioning contract. The following types of dynamic scaling are commonly used:

- **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**. the IT resource is relocated to a host with more capacity

The dynamic scalability architecture can be applied to a range of IT resources, including virtual servers and cloud storage devices. Besides the core automated scaling listener and resource replication mechanisms, the following mechanisms can also be used in this form of cloud architecture:

- **cloud usage monitor**. Specialized cloud usage monitors can track runtime usage in response to dynamic fluctuations caused by this architecture
- hypervisor. The hypervisor is invoked by a dynamic scalability system to create or remove virtual server instances, or to be scaled itself
- **pay-per-use monitor**. The pay-per-use monitor is engaged to collect usage cost information in response to the scaling of IT resources

5.1.2. *Automated scaling listener.* The automated scaling listener mechanism is a service agent that monitors and tracks communications between cloud service consumers and cloud services for dynamic scaling purposes. Workloads can be determined by the volume of cloud consumer-generated requests or via backend processing demands triggered by certain types of requests. Automated scaling listeners can provide different types of responses to workload fluctuation conditions, such as:

- automatically scaling IT resources out or in based on parameters previously defined by the cloud consumer (commonly referred to as **autoscaling**)
- **automatic notification** of the cloud consumer when workloads exceed current thresholds or fall below allocated resources. This way, the cloud consumer can choose to adjust its current IT resource allocation

Different cloud provider vendors have different names for service agents that act as automated scaling listeners.

5.2. **Load Balancer.** A common approach to horizontal scaling is to balance a workload across two or more IT resources to increase performance and capacity beyond what a single IT resource can provide. The **load balancer mechanism** is a runtime agent with logic fundamentally based on this premise. Beyond simple division of labor algorithms, load balancers can perform a range of specialized runtime workload distribution functions that include:

- **asymmetric distribution**. Larger workloads are issued to IT resources with higher processing capacities
- **workload prioritization**. Workloads are scheduled, queued, discarded, and distributed workloads according to their priority levels
- **content-aware distribution**. Requests are distributed to different IT resources as dictated by the request content

A load balancer is programmed or configured with a set of performance and QoS rules and parameters with the general objectives of optimizing IT resource usage, avoiding overloads, and maximizing throughput. The load balancer is typically located on the communication path between the IT resources generating the workload and the IT resources performing the workload processing. This mechanism can be designed as a transparent agent that remains hidden from the cloud service consumers, or as a proxy component that abstracts the IT resources performing their workload.

5.3. **Cloud usage monitor.** The cloud **usage monitor** mechanism is a lightweight and autonomous software program responsible for collecting and processing IT resource usage data. Depending on the type of usage metrics they are designed to collect and the manner in which usage data needs to be collected, cloud usage monitors can exist in different formats. Each can be designed to forward collected usage data to a log database for post-processing and reporting purposes.

5.3.1. *Monitoring agent.* An intermediary, **event-driven** program that exists as a service agent and resides along existing communication paths to transparently monitor and analyze dataflows. This type of cloud usage monitor is commonly used to measure network traffic and message metrics.

5.3.2. *Resource agent.* A processing module that collects usage data by having **event-driven** interactions with specialized resource software. This module is used to monitor usage metrics based on predefined, observable events at the resource software level, such as initiating, suspending, resuming, and vertical scaling.

5.3.3. *Polling agent.* A processing module that collects cloud service usage data by **polling** IT resources. This type of cloud service monitor is commonly used to periodically monitor IT resource status, such as uptime and downtime.

5.3.4. *Pay-per-use monitor.* This mechanism measures cloud-based IT resource usage in accordance with predefined pricing parameters and generates usage logs for fee calculations and billing purposes. Some typical monitoring variables are:

- request/response message quantity
- transmitted data volume
- bandwidth consumption

The data collected by the **pay-per-use monitor** is processed by a billing management system that calculates the payment fees.

6. Process virtualization with Docker

Docker is an open source project for building, shipping, and running programs. It is a command-line program, a background process, and a set of remote services to solve common software problems and simplifying your experience in installing, running, publishing, and removing software. It accomplishes this by using an operating system technology called **containers**. Using containers has been a best practice for a long time. But manually building containers can be challenging and easy to do incorrectly. This was a problem begging to be solved, and Docker helps. Any soft- ware run with Docker is run inside a container. Docker uses existing container engines to provide consistent containers built according to best practices.

6.1. Containers are not virtualization.

Unlike virtual machines, Docker containers don't use any hardware virtualization. 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 is not a hardware virtualization technology. Instead, it helps you use the container technology already built into your operating system kernel.

Virtual machines provide hardware abstractions so you can run operating systems. Containers are an operating system feature. So you can always run Docker in a virtual machine if that machine is running a modern Linux kernel. Docker for Mac and Windows users, and almost all cloud computing users, will run Docker inside virtual machines. Docker doesn't provide the container technology, but it specifically makes it simpler to use, by leveraging 10 major system features:

- **PID namespace**. Process identifiers and capabilities
- UTS namespace. Host and domain name
- MNT namespace. Filesystem access and structure
- IPC namespace. Process communication over shared memory
- **NET namespace**. Network access and structure
- USR namespace. User names and identifiers
- **chroot** syscall. Controls the location of the filesystem root
- **cgroups**. Resource protection
- CAP drop. Operating system feature restrictions
- Security modules. Mandatory access controls

Docker uses those to build containers at runtime, but it uses another set of technologies to package and ship containers.

6.2. Docker architecture.

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. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. Another Docker client is **Docker Compose**, that lets you work with applications consisting of a set of containers.

6.3. Underlying technology.

6.3.1. *Namespaces.*

Every running program - or process - on a Linux machine has a unique number called a process identifier (PID). A PID namespace is a set of unique numbers that identify processes. Linux provides tools to create multiple PID namespaces. Each namespace has a complete set of possible PIDs. Most programs will not need access to other running processes or be able to list the other running processes on the system. And so Docker creates a new PID namespace for each container by default. A container's PID namespace isolates processes in that container from processes in other containers.

Without a PID namespace, the processes running inside a container would share the same ID space as those in other containers or on the host. A process in a container would be able to determine what other processes were running on the host machine. Worse, processes in one container might be able to control processes in other containers. A process that cannot reference any processes outside its namespace is limited in its ability to perform targeted attacks. Like most Docker isolation features, you can optionally create containers without their own PID namespace.

6.3.2. *Control groups.* Physical system resources such as memory and time on the CPU are scarce. If the resource consumption of processes on a computer exceeds the available physical resources, the processes will experience performance issues and may stop running. Part of building a system that creates strong isolation includes providing resource allowances on individual containers. If you want to make sure that a program won't overwhelm other programs on your computer, the easiest thing to do is set limits on the resources that it can use. Docker Engine on Linux relies on control groups (**cgroups**) to organize processes into hierarchical groups, whose usage of various types of resources can then be limited and monitored.

6.3.3. *Images and layers.* Most of the time what we have been calling an image is actually a collection of image layers. A layer is set of files and file metadata that is packaged and distributed as an atomic unit. Internally, Docker treats each layer like an image, and layers are often called intermediate images. Images maintain parent/child relationships. In these relationships, they build from their parents and form layers. The files available to a container are the union of all lay- ers in the lineage of the image that the container was created from. Images can have relationships with any other image, including images in different repositories with dif- ferent owners.

6.3.4. *Union Filesystem.* Programs running inside containers know nothing about image layers. From inside a container, the filesystem operates as though it's not running in a container or operating on an image. From the perspective of the container, it has exclusive copies of the files provided by the image. This is made possible with something called a union filesystem (**UFS**).

Docker uses a variety of union filesystems and will select the best fit for your system. The filesystem is used to create mount points on your host's filesystem that abstract the use of layers. The layers created are bundled into Docker image layers. Likewise, when a Docker image is installed, its layers are unpacked and appropriately configured for use by the specific filesystem provider chosen for your system. The Linux kernel provides a namespace for the MNT system. When Docker creates a container, that new container will have its own MNT namespace, and a new mount point will be created for the container to the image. Lastly, chroot is used to make the root of the image filesystem the root in the container's context. This prevents anything running inside the container from referencing any other part of the host filesystem.

Different filesystems have different rules about file attributes, sizes, names, and characters. Union filesystems are in a position where they often need to translate between the rules of different filesystems. In the best cases, they're able to provide acceptable translations. In the worst cases, features are omitted. Union filesystems use a pattern called **copy-on-write**, and that makes implementing memory-mapped files. Some union filesystems provide implementations that work under the right conditions, but it may be a better idea to avoid memory-mapping files from an image.

6.3.5. *Container format.* Docker Engine combines the namespaces, control groups, and UnionFS into a wrapper called a **container format**. The default container format is libcontainer.

6.4. **Storage.** 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 filesystem, 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: volumes, and bind mounts. If running Docker on Linux you can also use a tmpfs mount. If running Docker on Windows you can also use a named pipe.

6.4.1. *Volumes.* Volumes are the preferred way to persist data in Docker containers and services. Some use cases for volumes include:

- sharing data among multiple running containers. Volumes persist after a container is removed/stopped, and multiple containers can mount the same volume simultaneously
- when the Docker host is not guaranteed to have a given directory or file structure. Volumes help you decouple the configuration of the Docker host from the container runtime
- store your container data on a remote host or a cloud provider
- back up, restore, or migrate data from one Docker host to another
- when an app requires high-performance I/O. Volumes are stored in the Linux VM rather than the host, which means that the reads and writes have much lower latency and higher throughput
- when an app requires fully native file system behavior, e.g. DBA requires precise control over disk flushing to guarantee transaction durability

6.4.2. *Bind mounts.* In general, you should use volumes where possible. Bind mounts are appropriate for the following types of use case:

- sharing configuration files from the host machine to containers. This is how Docker provides DNS resolution to containers by default
- sharing source code or build artifacts between a development environment on the Docker host and a container
- if you use Docker for development this way, your production Dockerfile would copy the production-ready artifacts directly into the image, rather than relying on a bind mount
- when the file or directory structure of the Docker host is guaranteed to be consistent with the bind mounts the containers require

6.4.3. *Tmpfs mounts.* tmpfs mounts are best used for cases when you do not want the data to persist either on the host machine or within the container. This may be for security reasons or to protect the performance of the container when your application needs to write a large volume of non-persistent state data.

6.5. **Networking.** Docker networking subsystem is pluggable, using drivers. Several drivers exist by default, and provide core networking functionality:

- bridge its the default network driver. Bridge networks are useful when you need multiple containers to communicate on the same Docker host
- host is for standalone containers, remove network isolation between the container and the Docker host, and use the host networking directly
- overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other
- macvlan networks allow 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

6.6. **Higher-level abstractions and orchestration.** Any processes, functionality, or data that must be discoverable and available over a network is called a **service**. That name, service, is an abstraction. By encoding those goals into an abstract term, we simplify how we talk about the things that use this pattern. We can reflect those same benefits in our tooling. Docker already does this for containers. Containers are described as processes that start using specific Linux namespaces, with specific filesystem views, and resource allotments. We do not have to describe those specifics each time we talk about a container, nor do we have to do the work of creating those namespaces ourselves. Docker does this for us. Docker provides tooling for other abstractions as well, including service.

6.6.1. *Declarative service environments with Compose.* Docker services are declarative abstractions: when we create a service, we declare that we want a certain number of replicas of that service, and Docker takes care of the individual commands required to maintain them. Declarative tools enable users to describe the new state of a system, rather than the steps required to change from the current state to the new state. The Swarm orchestration system is a state reconciliation loop that continuously compares the declared state of the system that the user desires with the current state of the system. When it detects a difference, it uses a simple set of rules to change the system so that it matches the desired state. Orchestrators automate service replication, resurrection, deployments, health checking, and rollback. Compose files use Yet Another Markup Language (YAML).

6.6.2. *Orchestrating services on a cluster.* Application developers and operators frequently deploy services onto multiple hosts to achieve greater availability and scalability. When an application is deployed across multiple hosts, the redundancy in the application's deployment provides capacity that can serve requests when a host fails or is removed from service. Deploying across multiple hosts also permits the application to use more compute resources than any single host can provide.

Docker Swarm is a clustering technology that connects a set of hosts running Docker and lets you run applications built using Docker services across those machines. Swarm orchestrates the deployment and operation of Docker services across this collection of machines. Swarm schedules tasks according to the application's resource requirements and machine capabilities. When you join a Docker Engine to a Swarm cluster, you specify whether that machine should be a manager or a worker. **Managers** listen for instructions to create, change, or remove definitions for entities such as Docker services, configuration, and secrets. Managers instruct **worker** nodes to create containers and volumes that implement Docker service instances. Managers continuously converge the cluster to the state you have declared it should be in.

7. Kubernets

8. Cloud storage

A variety of sources feed a continuous stream of data to cloud applications. An ever increasing number of cloud-based services collect detailed data about their services and information about the users of these services. Then the service providers use the clouds to analyze that data. Storage and processing on the cloud are intimately tied to one another; indeed, sophisticated strategies to reduce the access time and to support real-time multimedia access are necessary to satisfy the requirements of content delivery. On the other hand, most cloud applications process very large amounts of data; effective data replication and storage management strategies are critical to the computations performed on the cloud.

8.1. **Atomic actions.**

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. An important observation is that such atomic actions should not expose the state of the system until the action is completed. Hiding the internal state of an atomic action reduces the number of states a system can be in; thus, it simplifies the design and maintenance of the system. An atomic action is composed of several steps, each of which may fail; therefore, we have to take additional precautions to avoid exposing the internal state of the system in case of such a failure.

Atomicity cannot be implemented without some hardware support; indeed, 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 noninterruptible 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 and before-or-after atomicity. **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. To guarantee the all-or-nothing property of an action we have to distinguish preparatory actions that can be undone from irreversible ones, such as the alteration of the only copy of an object. Such preparatory actions are as follows:

allocation of a resource, fetching a page from secondary storage, allocation of memory on the stack, and so on. One of the golden rules of data management is never to change the only copy; maintaining the history of changes and a log of all activities allows us to deal with system failures and to ensure consistency. An all-or-nothing action consists of a **pre-commit** and a **post-commit** phase; during the former it should be possible to back up from it without leaving any trace, whereas the latter phase should be able to run to completion. The transition from the first to the second phase is called a commit point. During the pre-commit phase all steps necessary to prepare the post-commit phase – for example, check permissions, swap in main memory all pages that may be needed, mount removable media, and allocate stack space – must be carried out; during this phase no results should be exposed and no irreversible actions should be carried out. Shared resources allocated during the pre-commit phase cannot be released until after the commit point. The commit step should be the last step of an all-or-nothing action.

The common storage model implemented by hardware is the so-called **cell storage**, a collection of cells each capable of holding an object. Cell storage does not support all-or-nothing actions. 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. The storage model that supports all-or-nothing actions is called **journal storage**. Now the cell storage is no longer accessible to the action because the access is mitigated by a **storage manager**. In addition to the basic primitives to read an existing value and to write a new value in cell storage, the storage manager uniquely identifies an action that changes the value in cell storage and, when the action is aborted, is able to retrieve the version of the variable before the action and restore it. When the action is committed, then the new value should be written to the cell. For a journal storage, in addition to the version histories of all variables affected by the action, we have to implement a catalog of variables and maintain a record to identify each new action. A new action first invokes the **Action** primitive; at that time an outcome record uniquely identifying the action is created. Then, every time the action accesses a variable, the version history is modified. Finally, the action invokes either a **Commit** or an **Abort** primitive. In the journal storage model the action is atomic.

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. Atomicity is a critical concept in our efforts to build reliable systems from unreliable components and, at the same time, to support as much parallelism as possible for better performance. Atomicity allows us to deal with unforseen events and to support coordination of concurrent activities. The unforseen event could be a system crash, a request to share a control structure, the need to suspend an activity, and so on; in all these cases we have to save the state of the process or of the entire system to be able to restart it at a later time. Because atomicity is required in many contexts, it is desirable to have a systematic approach rather than an ad hoc one. A systematic approach to atomicity must address several delicate questions:

- how to guarantee that only one atomic action has access to a shared resource at any given time

- how to return to the original state of the system when an atomic action fails to complete
- how to ensure that the order of several atomic actions leads to consistent results

Answers to these questions increase the complexity of the system and often generate additional problems.

8.2. **Storage models, filesystems, and databases.** A **storage model** describes the layout of a data structure in physical storage; a data model captures the most important logical aspects of a data structure in a database. The physical storage can be a local disk, a removable media, or storage accessible via a network. 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 fits exactly in one cell. This model reflects the physical organization of several storage media; the primary memory of a computer is organized as an array of memory cells, and a secondary storage device (e.g., a disk) is organized in sectors or blocks read and written as a unit. read/write coherence and before-or-after atomicity are two highly desirable properties of any storage model and in particular of cell storage.

Journal storage is a fairly elaborate organization for storing composite objects such as records consisting of multiple fields. Journal storage consists of a manager and cell storage, where the entire history of a variable is maintained, 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. The journal manager translates user requests to commands sent to the cell storage: (i) read a cell; (ii) write a cell; (iii) allocate a cell; or (iv) deallocate a cell.

In the context of storage systems, a log contains a history of all variables in cell storage. The infor- mation about the updates of each data item forms a record appended at the end of the log. A log provides authoritative information about the outcome of an action involving cell storage; the cell storage can be reconstructed using the log, which can be easily accessed – we only need a pointer to the last record. 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 The log is always kept on nonvolatile storage (e.g., disk) and 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.

Many cloud applications must support online transaction processing and have to guarantee the cor- rectness 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. More stringent conditions must sometimes be observed; for example, banking transactions must be processed in the order in which they are issued, the so-called external time consistency. To guarantee correctness, a transaction-processing system supports all-or-nothing atomicity.

8.2.1. *Network filesystem.* The **NFS** is very popular and has been used for some time, but it does not scale well and has reliability problems; an NFS server could be a single point of failure. NFS was the first widely used distributed file system; the development of this application based on the client-server model was motivated by the need to share a file system among a number of clients interconnected by a local area network. A majority of workstations were running under Unix; thus, many design decisions for the NFS were influenced by the design philosophy of the Unix File System (UFS). It is not surprising that the NFS designers aimed to:

- provide the same semantics as a local UFS to ensure compatibility with existing applications
- facilitate easy integration into existing UFS
- ensure that the system would be widely used and thus support clients running on different operating systems
- accept a modest performance degradation due to remote access over a network with a bandwidth of several Mbps

The NFS is based on the client-server paradigm. The client runs on the local host while the server is at the site of the remote file system, and they interact by means of RPCs. The API interface of the local file system distinguishes file operations on a local file from the ones on a remote file and, in the latter case, invokes the RPC client.

8.2.2. *General parallel filesystem.* Parallel I/O implies execution of multiple input/output operations concurrently. Support for parallel I/O is essential to the performance of many applications. Therefore, once distributed file systems became ubiquitous, the natural next step in the evolution of the file system was to support parallel access. Parallel file systems allow multiple clients to read and write concurrently from the same file. Concurrency control is a critical issue for parallel file systems. Several semantics for handling the shared access are possible. The GPFS was developed at IBM in the early 2000s as a parallel file system that emulates closely the behavior of a general-purpose POSIX system running on a single system. GPFS was designed for optimal performance of large clusters.

Reliability is a major concern in a system with many physical components. To recover from system failures, GPFS records all metadata updates in a **write-ahead** log file. Write-ahead means that updates are written to persistent storage only after the log records have been written. For example, when a new file is created, a directory block must be updated and an inode for the file must be created. These records are transferred from cache to disk after the log records have been written. When the directory block is written and then the I/O node fails before writing the inode, then the system ends up in an inconsistent state and the log file allows the system to recreate the inode record. The log files are maintained by each I/O node for each file system it mounts; thus, any I/O node is able to initiate recovery on behalf of a failed node. Disk parallelism is used to reduce access time. Multiple I/O read requests are issued in parallel and data is prefetched in a buffer pool.

Data striping allows concurrent access and improves performance but can have unpleasant side-effects. Indeed, when a single disk fails, a large number of files are affected. To reduce the impact of such undesirable events, the system attempts to mask a single disk failure or the failure of the access path to a disk. The system uses **RAID** devices with the stripes equal to the block size and dual-attached RAID controllers. To further improve the fault tolerance of the system, GPFS data files as well as metadata are replicated on two different physical disks.

Consistency and performance, critical to any distributed file system, are difficult to balance. Support for concurrent access improves performance but faces serious challenges in maintaining consistency. In GPFS, consistency and synchronization are ensured by a **distributed locking mechanism**; a **central lock manager** grants lock tokens to **local lock managers** running in each I/O node. Lock tokens are also used by the cache management system.

8.2.3. *Google filesystem.* The GFS uses thousands of storage systems built from inexpensive commodity components to provide petabytes of storage to a large user community with diverse needs. The system was designed after a careful analysis of the file characteristics and of the access models. Some of the most important aspects of this analysis reflected in the GFS design are:

- scalability and reliability are critical features of the system; they must be considered from the beginning rather than at some stage of the design
- the vast majority of files range in size from a few GB to hundreds of TB
- the most common operation is to append to an existing file; random write operations to a file are extremely infrequent
- sequential read operations are the norm
- the users process the data in bulk and are less concerned with the response time
- the consistency model should be relaxed to simplify the system implementation, but without placing an additional burden on the application developers

GFS files are collections of fixed-size segments called chunks; at the time of file creation each chunk is assigned a unique chunk handle. Chunks are stored on Linux filesystems and are replicated on multiple sites; a user may change the number of the replicas from the standard value of three to any desired value. The chunk size is 64 MB; this choice is motivated by the desire to optimize performance for large files and to reduce the amount of metadata maintained by the system. A large chunk size increases the likelihood that multiple operations will be directed to the same chunk; thus it reduces the number of requests to locate the chunk and, at the same time, it allows the application to maintain a persistent network connection with the server where the chunk is located. Space fragmentation occurs infrequently because the chunk for a small file and the last chunk of a large file are only partially filled.

A master controls a large number of chunk servers; it 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.

System reliability is a major concern, and the operation log maintains a historical record of metadata changes, enabling the master to recover in case of a failure. As a result, such changes are atomic and are not made visible to the clients until they have been recorded on multiple replicas on persistent storage. To recover from a failure, the master replays the operation log. To minimize the recovery time, the master periodically checkpoints its state and at recovery time replays only the log records after the last checkpoint.

Each chunk server is a commodity Linux system; it receives instructions from the master and responds with status information. To access a file, an application sends to the master the filename and the chunk index, the offset in the file for the read or write operation; the master responds with the chunk handle and the location of the chunk. Then the application communicates directly with the chunk server to carry out the desired file operation.

The consistency model is very effective and scalable. Operations, such as file creation, are atomic and are handled by the master. To ensure scalability, the master has minimal involvement in file mutations and operations such as write or append that occur frequently. In such cases the master grants a lease for a particular chunk to one of the chunk servers, called the primary; then, the primary creates a serial order for the updates of that chunk.

When data for a write straddles the chunk boundary, two operations are carried out, one for each chunk. The steps for a write request illustrate a process that buffers data and decouples the control flow from the data flow for efficiency:

(1) the client contacts the master, which assigns a lease to one of the chunk servers for a particular chunk if no lease for that chunk exists; then the master replies with the ID of the primary as well as secondary chunk servers holding replicas of the chunk. The client caches this information
(2) the client sends the data to all chunk servers holding replicas of the chunk; each one of the chunk servers stores the data in an internal LRU buffer and then sends an acknowledgment to the client
(3) the client sends a write request to the primary once it has received the acknowledgments from all chunk servers holding replicas of the chunk. The primary identifies mutations by consecutive sequence numbers
(4) the primary sends the write requests to all secondaries
(5) each secondary applies the mutations in the order of the sequence numbers and then sends an acknowledgment to the primary
(6) finally, after receiving the acknowledgments from all secondaries, the primary informs the client

8.2.4. *Apache Hadoop filesystem.* **Hadoop** is an open-source, Java-based software system, which supports distributed applications handling extremely large volumes of data. A Hadoop system has two components, a MapReduce engine and a database. The database could be the Hadoop File System (HDFS), Amazon S3, or CloudStore, an implementation of the Google File System.

HDFS is a distributed file system written in Java; it is portable, but it cannot be directly mounted on an existing operating system. HDFS is not fully POSIX compliant, but it is highly performant. HDFS replicates data on multiple nodes. The default is three replicas; a large dataset is distributed over many nodes. The **nameNode** (master) running on the master manages the data distribution and data replication and communicates with data nodes running on all cluster nodes; it shares with the job tracker information about data placement to minimize communication between the nodes on which data is located and the ones where it is needed. Although HDFS can be used for applications other than those based on the **MapReduce** model, its performance for such applications is not at par with the ones for which it was originally designed.

8.2.5. *Transaction processing and NoSQL databases.* 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. The search for alternate models with which to store the data on a cloud is motivated by the need to decrease the latency by caching frequently used data in memory on dedicated servers, rather than fetching it repeatedly. In addition, distributing the data on a large number of servers allows multiple transactions to occur at the same time and decreases the response time. The relational schema are of little use for such applications in which conversion to key-value databases seems a much better approach. Of course, such systems do not store meaningful metadata information, unless they use extensions that cannot be exported easily.

A major concern for the designers of OLTP systems is to reduce the **response time**. The term **memcaching** refers to a general-purpose distributed memory system that caches objects in main memory. The memcached system is based on a client-server architecture and runs under several operating systems. **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

The "soft-state" approach in the design of **NoSQL** allows data to be inconsistent and transfers the task of implementing only the subset of the ACID properties required by a specific application to the application developer. The NoSQL systems ensure 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; they increase availability, reduce response time, and enhance scalability.

8.2.6. *Chubby: A locking service.* Locks support the implementation of reliable storage for loosely coupled distributed systems; they enable controlled access to shared storage and ensure atomicity of read and write operations. Furthermore, critically important to the design of reliable distributed storage systems are distributed consensus problems, such as the election of a master from a group of data servers. A master has an important role in system management; for example, in GFS the master maintains state information about all system components.

8.2.7. *BigTable.* BigTable is a distributed storage system developed by Google to store massive amounts of data and to scale up to thousands of storage servers. The system uses the GFS to store user data as well as system information. To guarantee atomic read and write operations, it uses the Chubby distributed lock service.

The system is based on a simple and flexible data model. It allows an application developer to exercise control over the data format and layout and reveals data locality information to the application clients. A row key is an arbitrary string of up to 64 KB, and a row range is partitioned into tablets serving as units for load balancing. The timestamps used to index various versions of the data in a cell are 64-bit integers. A column key consists of a string defining the family name, a set of printable characters, and an arbitrary string as qualifier.

8.2.8. *DynamoDB.*

9. Big data

9.1. **MapReduce.** A main advantage of cloud computing is elasticity - the ability to use as many servers as necessary to optimally respond to the cost and the timing constraints of an application. In the case of transaction processing systems, typically a front-end system distributes the incoming transactions to a number of back-end systems and attempts to balance the load among them. As the workload increases, new back-end systems are added to the pool. For data-intensive batch applications, partitioning the workload is not always trivial. Only in some cases can the data be partitioned into blocks of arbitrary size and processed in parallel by servers in the cloud. We distinguish two types of divisible workloads:

- modularly divisible. The workload partitioning is defined a priori
- arbitrarily divisible. The workload can be partitioned into an arbitrarily large number of smaller workloads of equal or very close size

MapReduce is based on a very simple idea for parallel processing of data-intensive applications supporting arbitrarily divisible load sharing. First, split the data into blocks, assign each block to an instance or process, and run these instances in parallel. Once all the instances have finished, the computations assigned to them start the second phase: Merge the partial results produced by individual instances. The so-called same program, multiple data (**SPMD**) paradigm, used since the early days of parallel computing, is based on the same idea but assumes that a master instance partitions the data and gathers the partial results.

The MapReduce is a programming model conceived for processing and generating large data sets on computing clusters. As a result of the computation, a set of input $\langle key, value\rangle$ pairs is transformed into a set of output $\langle key', value'\rangle$ pairs.

Call M and R the number of Map and Reduce tasks, respectively, and N the number of systems used by the MapReduce. When a user program invokes the MapReduce function, the following sequence of actions take place:

(1) the run-time library splits the input files into M splits of 16 to 64 MB each, identifies a number N of systems to run, and starts multiple copies of the program, one of the system being a master and the others workers. The master assigns to each idle system either a Map or a Reduce task
(2) a worker being assigned a Map task reads the corresponding input split, parses $\langle key, value\rangle$ pairs, and passes each pair to a user-defined Map function. The intermediate $\langle key, value\rangle$ pairs produced by the Map function are buffered in memory before being written to a local disk and partitioned into R regions by the partitioning function
(3) the locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the Reduce workers. A Reduce worker uses remote procedure calls to read the buffered data from the local disks of the Map workers; after reading all the intermediate data, it sorts it by the intermediate keys. For each unique intermediate key, the key and the corresponding set of intermediate values are passed to a user-defined Reduce function. The output of the Reduce function is appended to a final output file
(4) when all Map and Reduce tasks have been completed, the master wakes up the user program

The system is fault tolerant. For each Map and Reduce task, the master stores the state (idle, inprogress, or completed) and the identity of the worker machine. The master pings every worker periodically and marks the worker as failed if it does not respond. A task in progress on a failed worker is reset to idle and becomes eligible for rescheduling. The master writes periodic checkpoints of its control data structures and, if the task fails, it can be restarted from the last checkpoint. The data is stored using GFS.

9.2. **Hadoop MapReduce.** MapReduce constitutes a simplified model for processing large quantities of data and imposes constraints on the way distributed algorithms should be organized to run over a MapReduce infrastructure. Although the model can be applied to several different problem scenarios, it still exhibits limitations, mostly due to the fact that the abstractions provided to process data are very simple, and complex problems might require considerable effort to be represented in terms of map and reduce functions only. Therefore, a series of extensions to and variations of the original MapReduce model have been proposed.

Apache Hadoop is a collection of software projects for reliable and scalable distributed computing. Taken together, the entire collection is an open-source implementation of the MapReduce framework supported by a GFS-like distributed filesystem. The initiative consists of mostly two projects: Hadoop Distributed File System (HDFS) and Hadoop MapReduce. Besides the core projects of Hadoop, a collection of other projects related to it provides services for distributed computing.

9.2.1. *Map-Reduce-Merge.* It is an extension of the MapReduce model, introducing a third phase to the standard MapReduce pipeline - the Merge phase - that allows efficiently merging data already partitioned and sorted (or hashed) by map and reduce modules. The MapReduce-Merge framework simplifies the management of heterogeneous related datasets and pro vides an abstraction able to express the common relational algebra operators as well as several join algorithms.

9.3. **Hadoop YARN.** Yet Another Resource Negotiator is the next generation of Hadoop compute platform, which departs from its familiar, monolithic architecture. By separating resource management functions from the programming model, YARN delegates many scheduling-related functions to per-job components. In this new context, MapReduce is just one of the applications running on top of YARN. This separation provides a great deal of flexibility in the choice of programming framework. Programming frameworks running on YARN coordinate intra-application communication, execution flow, and dynamic optimizations as they see fit.

9.3.1. *Architecture.* YARN lifts some functions into a platform layer responsible for resource management, leaving coordination of logical execution plans to a host of framework implementations. Specifically, a per-cluster **ResourceManager** (RM) tracks resource usage and node liveness, enforces allocation invariants, and arbitrates contention among tenants. By separating these duties in the JobTracker's charter, the central allocator can use an abstract description of tenants' requirements, but remain ignorant of the semantics of each allocation. That responsibility is delegated to an **ApplicationMaster** (AM), which coordinates the logical plan of a single job by requesting resources from the RM, generating a physical plan from the resources it receives, and coordinating the execution of that plan around faults.

9.3.2. *ResourceManager.* The ResourceManager exposes two public interfaces towards clients submitting applications, ApplicationMaster(s) dynamically negotiating access to resources, and one internal interface towards NodeManagers for cluster monitoring and resource access management. The RM matches a global model of the cluster state against the digest of resource requirements reported by running applications. This makes it possible to tightly enforce global scheduling propertiesfairness), but it requires the scheduler to obtain an accurate understanding of applications' resource requirements. ApplicationMasters codify their need for resources in terms of one or more ResourceRequests, each of which tracks:

(1) number of containers
(2) resources per container
(3) locality preferences
(4) priority of requests within the application

The scheduler tracks, updates, and satisfies these requests with available resources, as advertised on NM heartbeats. In response to AM requests, the RM generates containers together with tokens that grant access to resources. The RM forwards the exit status of finished containers, as reported by the NMs, to the responsible AMs. AMs are also notified when a new NM joins the cluster so that they can start requesting resources on the new nodes.

The RM is not responsible for coordinating application execution or task fault-tolerance, but neither is is charged with providing status or metrics for running applications (now part of the ApplicationMaster), nor serving framework specific reports of completed jobs (now delegated to a per-framework daemon). This is consistent with the view that the ResourceManager should only handle live resource scheduling.

9.3.3. *ApplicationMaster.* An application may be a static set of processes, a logical description of work, or even a long-running service. The ApplicationMaster is the process that coordinates the application's execution in the cluster, but it itself is run in the cluster just like any other container. A component of the RM negotiates for the container to spawn this bootstrap process.

The AM periodically heartbeats to the RM to affirm its liveness and to update the record of its demand. After building a model of its requirements, the AM encodes its preferences and constraints in a heartbeat message to the RM. In response to subsequent heartbeats, the AM will receive a container lease on bundles of resources bound to a particular node in the cluster.

Since the RM does not interpret the container status, the AM determines the semantics of the success or failure of the container exit status reported by NMs through the RM. Being a container running in a cluster of unreliable hardware, the AM itself should be resilient to failure. YARN provides some support for recovery, but because fault tolerance and application semantics are so closely intertwined, much of the burden falls on the AM.

9.3.4. *NodeManager.* The NodeManager is the "worker" daemon in YARN. It authenticates container leases, manages containers' dependencies, monitors their execution, and provides a set of services to containers. Operators configure it to report memory, CPU, and other resources available at this node and allocated for YARN. After registering with the RM, the NM heartbeats its status and receives instructions.