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 cloud model is composed of five essential characteristics:

- **on-demand self-service**. A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider
- **broad network access**. Capabilities are available over the network and accessed through standard mechanisms
- resource polling. The provider computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand
- **rapid elasticity**. Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand
- measured service. Cloud systems automatically control and optimize resource use by leaveraging a metering capability at some level of abstraction appropriate to the type of service

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. *Services models.* 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. 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.

- Software as a Service (**SaaS**). The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure
- Platform as a Service (**PaaS**). The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications developed using programming languages, libraries, services, and tools supported by the provider
- Infrastructure as a Service (**IaaS**). The capability provided to the consumer is to provision processing, storage, network, and other foundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include OS and applications

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

- **private cloud**. The cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers
- **community cloud**. The cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns
- **public cloud**. The cloud infrastructure is provisioned for open use by the general public
- **hybrid cloud**. The cloud infrastructure is a composition of two or more distinct cloud infrastructures that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability

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

- **cloud consumer**. Person or organization that maintains a business relationship withm and uses services from, cloud service providers
- **cloud provider**. Person, organization or entity responsible for making a service available to service consumers
- **cloud carrier**. The intermediary that provides connectivity and transport of cloud services between cloud providers and cloud consumers
- **cloud broker**. An entity that manages the use, performance and delivery of cloud services, and negotiates relationships between cloud providers and cloud consumers
- **cloud auditor**. A party that can conduct independent assessment of cloud services, information system operations, performance and security of the cloud implementation

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

1.3.1. *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 privileged access to cloud consumer data. 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.

1.3.2. *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.

1.3.3. *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.

1.3.4. *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.

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.

2.2. **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.

3. Principles of distributed computing

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 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.

3.1. **The AKF scale cube.** Imagine a cube drawn with the aid of a three-dimensional axis for a guide. The initial point, with coordinates of $(0, 0, 0)$, is the point of least scalability within any system. It consists of a single monolithic solution deployed on a single server. It might scale up with larger and faster hardware, but it will not scale out. As such, it will limit your growth to that which can be served by a single unit. In other words, your system will be bound by how fast the server runs the application in question and how well the application is tuned to use the server.

Making modifications to your solution for the purposes of scaling moves you along one of the three axes. Equivalent effort applied to any axis will not always return equivalent results. Choosing one axis does not preclude you from making use of other axes later. When we make choices about what to implement, we should select the splits that have the highest return (in terms of scale) for our effort and that meet our needs in terms of delivering scalability in time for customer demand.

The x-axis of scale is very useful and easy to implement, you simply clone the activity among several participants. Scaling along the x-axis starts to fail, however, when you have a lot of different tasks requiring significantly different information

from many potential sources. Fast transactions start to run at the speed of slow transactions, and everything starts to work suboptimally.

The y-axis helps to solve that problem by isolating transaction type and speed to systems and people specializing in that area of data or service. Slower transactions are now bunched together, but because the data set has been reduced relative to the x-axis only example, they run faster than they had previously. Fast transactions also speed up because they are no longer competing for resources with the slower transactions and their data set has been reduced. Monolithic systems are reduced to components that operate more efficiently and can scale for data and transaction needs.

The z-axis not only helps us scale transactions and data, but may also help with monolithic system deconstruction. Furthermore, we can now move teams and systems around geographically and start to gain benefits from this geographic dispersion, such as disaster recovery.

3.2. **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.

3.2.1. *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. Systems developed according to this style are composed of one large main program that accomplishes its tasks by invoking subprograms or procedures. 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. 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

3.2.2. *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**. 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. 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**. 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. 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. 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

3.3. **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.

3.3.1. *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. 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.

3.3.2. *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.

3.4. **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.

3.4.1. *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.

3.4.2. *Distributed Object framework.* 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
(2) the client process, by using a given addressing scheme, obtains a reference to the active remote object
(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 references have to be tracked. Being more complex and resource demanding, marshaling by reference should be used only when duplication of parameters and return values lead to unexpected and inconsistent behavior of the system.

3.4.3. *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, services are autonomous, services share schema and contracts (not class or interface definitions), services compatibility is based on policy.

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**).

3.5. **Service-Oriented Architecture. SOA** organizes a software system into a colletion 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.

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**, representing a single centralized executable business process (the orchestrator) that coordinates the interaction among different services. The orchestrator is responsible for invoking and combining the services. On the other hand, **service choreography** coordinates interaction of services in a distributed manner. This means that a choreography differs from an orchestration with respect to where the logic that controls the interactions between the services involved should reside.

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, recommended guiding principles are standardized service contracts, loose coupling, abstractions, reusability, autonomy, lack of state, discoverability, and composability.

3.5.1. *Web services.* 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 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.

3.6. **Microservices.** Initially, a monolithic architecture offers several benefits, since it is simple to developt and easy to make radical changes, straightforward to test and deploy, easy to scale. But over time, as complexity increases, development, testing, deployment, and scaling becomes much more difficult. Instead, a **microservice** is a cohesive, independent process interacting via messages, and a **microservice architecture** is a distributed application where all its modules are microservices. Microservices manage growing complexity by functionally decomposing large systems into a set of independent services. By making services completely independent in development and deployment, microservices emphasise loose coupling and high cohesion by taking modularity to the next level. This approach delivers all sorts of benefits in terms of maintainability, scalability and so on:

- services are small and easily maintained, and can be independently deployable and scalable
- continuos integration. It is possible to plan gradual transitions to new versions of a microservice
- microservices naturally lend themselves to containerisation
- the microservice architecture enables teams to be autonomous
- better fault isolation

Like SOA, microservices architectures are made up of loosely coupled, reusable, and specialized components that often work independently of one another. Microservices also use a high degree of cohesion, otherwise known as bounded context. Bounded context refers to the relationship between a component and its data as a standalone entity or unit with very few dependencies. However, we can point out some key differences:

- **communication** and interoperability. In a microservices architecture, each service is developed independently, with its own communication protocol, using lightweight and open source messaging protocols like HTTP/REST and gRPC. On the other hand, each service in SOA must share a common Enterprise Service Bus (ESB) middleware, which can be a single point of failure. Furthermore, SOA tends to use more heavyweight technologies such as SOAP and other WS standards
- **data handling**. SOA architecture typically includes a single data storage layer shared by all services within a given application, whereas microservices will dedicate a server or database for data storage for any service that needs it
- **service size**. Microservices architectures are made up of highly specialized services, each of which is designed to do one thing very well. The services that make up SOAs, on the other hand, can range from small, specialized services to enterprise-wide services

Microservices also comes with a bundle of problems that are inherited from distributed systems and from SOA:

- deciding when to adopt the microservice architecture, and finding the right set of services, can be challenging
- distributed systems are complex and so development, testing and deployment can be problematic
- deploying features that span multiple services requires careful coordination (among development teams)

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: underutilization of hardware and software resources, lack of space, greening initiatives, and rise of administrative costs.

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. 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:

- increased security. 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
- managed execution. Virtualization of the execution environment not only allows increased security, but a wider range of features also can be implemented, such as sharing, aggregation, emulation, and isolation

- portability. In the case of a hardware virtualization solution, the guest is packaged into a virtual image that, in most cases, can be safely moved and executed on top of different virtual machines. In the case of programming-level virtualization, as implemented by the JVM or the .NET runtime, the binary code representing application components can be run without any recompilation on any implementation of the corresponding virtual machine

4.1.1. *Virtualization and cloud computing.* Besides being an enabler for computation on demand, virtualization also gives the opportunity to design more efficient computing systems by means of consolidation, which is performed transparently to cloud computing service users. Since virtualization allows us to create isolated and controllable environments, it is possible to serve these environments with the same resource without them interfering with each other. If the underlying resources are capable enough, there will be no evidence of such sharing. This practice is also known as server consolidation, while the movement of virtual machine instances is called **virtual machine migration**. Because virtual machine instances are controllable environments, consolidation can be applied with a minimum impact, either by temporarily stopping its execution and moving its data to the new resources or by performing a finer control and moving the instance while it is running. This second techniques is known as **live migration** and in general is more complex to implement but more efficient since there is no disruption of the activity of the virtual machine instance.

(1) **pre-migration**. Determine the migrating VM and the destination host. This can be performed manually, but in most circumstances it is automatically started by strateges such as load balancing and server consolidation
(2) **reservation** and **iterative pre-copy**. The whole execution state of the VM is stored in memory and sent to the destination node, ensuring continuity of service
(3) **stop and copy**. The VM is suspended and its apps no longer run (downtime). Other non-memory data such as CPU and network states should be sent as well
(4) **commitment**. the VM reloads its state and recovers the execution of its programs. The services provided by this VM continues
(5) **activation**. The network connection is redirected to the new VM and the dependency to the source host is cleared, removing the original VM from the source host

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. 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.

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 or modify the privileged state.

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.

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[1] (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.

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

[1]Instruction Set Architecture.

- **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[2] 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

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, which 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
- **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 technique has been successfully used by Xen for providing virtualization solutions for Linux-based operating systems specifically ported to run on Xen hypervisors
- **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

[2]Application Binary Interface.

Multiple instances of a variety of operating systems may share the virtualized hardware resources. This contrasts with operating-system-level virtualization, where all instances must share a single kernel, though the guest operating systems can differ in user space.

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. This virtualization technique can be considered an evolution of the chroot mechanism in Unix systems. 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.

5. Scaling mechanisms

5.1. **Autoscaling.** 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 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**. 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

5.1.1. *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.

5.2.1. *AWS Elastic Load Balancer.* Elastic Load Balancing (**ECB**) automatically distributes incoming traffic across multiple targets, such as EC2 instances, containers, and IP addresses, in one or more Availability Zones. It monitors the health of its registered targets, and routes traffic only to the healthy targets. A listener checks for connection requests from clients, using the protocol and port that you configure. The rules that you define for a listener determine how the load balancer routes requests to its registered targets. Each rule consists of a priority, one or more actions, and one or more conditions. ECB supports the following load balancers: Application Load Balancers, Network Load Balancers, Gateway Load Balancers, and Classic Load Balancers.

An **Application Load Balancer**, after it receives a request, evaluates the listener rules in priority order to determine which rule to apply, and then selects a target from the target group for the rule action. You can configure listener rules to route requests to different target groups based on the content of the application traffic. You can configure the routing algorithm used at the target group level. The default routing algorithm is round robin.

A **Network Load Balancer** can handle millions of requests per second. After the load balancer receives a connection request, it selects a target from the target group for the default rule. It attempts to open a TCP connection to the selected target on the port specified in the listener configuration.

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

5.4. **MAPE-K.** The **MAPE-K** (Monitor-Analyze-Plan-Execute over a shared Knowledge) is the most influential reference control model for autonomic and self-adaptive systems. A common approach to realize a feedback loop - a system for improving a product, process, etc. by collecting and reacting to events - is by means of a MAPE-K loop.

A **component Knowledge** maintains data of the managed system and environment, adaptation goals, and other relevant states that are shared by the MAPE components. A **component Monitor** gathers particular data from the underlying managed system and the environment through probes (or sensors) of the managed system, and saves data in the Knowledge. A **component Analyze** performs data analysis to check whether an adaptation is required. If so, it triggers a **component Plan** that composes a workflow of adaptation actions necessary to achieve the system's goals. These actions are then carried out by a **component Execution** through effectors (or actuators) of the managed system.

Computations M, A, P, and E may be made by multiple components that coordinate with one another to adapt the system when needed, i.e., they may be decentralized through multiple MAPE-K loops. These MAPE components can communicate explicitly or indirectly by sharing information in the knowledge repository.

AWS implements these componets through CloudWatch (Monitor), alarms (Analyze), dynamic scaling (Plan), internal mechanisms or CLI commands (Execution), metrics (knowedge).

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. Unlike virtual machines, Docker is not a hardware virtualization technology. Instead, it helps you use the container technology already built into your operating system kernel. While virtual machines provide hardware abstractions so you can run operating systems, containers are an operating system feature.

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.

6.1. **Underlying technology.**

6.1.1. *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 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.

6.1.2. *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.

6.1.3. *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.

6.1.4. *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 can select the best fit for the system. The filesystem is used to create mount points on the host 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 the system. 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.

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

6.2. **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.

6.2.1. *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

6.2.2. *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

6.2.3. *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.

6.3. **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, removes 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

6.4. **Higher-level abstractions and orchestration.** Any processes, functionality, or data that must be discoverable and available over a network is called a **service**. 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.

6.4.1. *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).

6.4.2. *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. Kubernetes

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. Kubernetes provides you with a framework to run distributed systems resiliently. lt takes care of scaling and failover for your application, provides deployment patterns, and more. Kubernetes provides you with:

- service discovery and load balancing. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable
- storage orchestration. Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more
- automated rollouts and rollbacks. You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate.
- automatic bin packing. You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources
- self-healing. Kubernetes restarts containers that fail, replaces containers, kills containers that do not respond to your user-defined health check, and does not advertise them to clients until they are ready to serve
- secret and configuration management. Kubernetes lets you stare and manage sensitive information, such as passwords, OAuth tokens, and SSH keys

Kubernetes operates at the container level rather than at the hardware level, providing some generally applicable features common to PaaS offerings, such as deployment, scaling, load balancing, and lets users integrate their logging, monitoring, and alerting solutions. However, Kubernetes is not monolithic, and these default solutions are optional and pluggable. Kubernetes provides the building blocks far building developer platfarms, but preserves user choice and flexibility where it is important.

7.1. **Kubernetes components.** When you deploy Kubernetes, you get a cluster. A **Kubernetes cluster** consists of a set of worker machines, called **nodes**, that

run containerized applications. Every cluster has at least one worker node. The worker node(s) host the **pods**, i.e. the components of the application workload. The **control plane** manages the worker nodes and the pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

7.2. **Control plane components.** The control plane components make global decisions about the cluster, e.g. scheduling, as well as detecting and responding to cluster events. Control plane components can be run on any machine in the cluster. However, for simplicity, setup scripts typically start all control plane components on the same machine, and do not run user containers on this machine.

7.2.1. *Kube-apiserver.* The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane. The main implementation of a Kubernetes API server is kube-apiserver, designed to scale horizontally - that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.

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

7.2.3. *kube-controller-manager.* Control Plane component that runs controller processes. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process. Some types of these controllers are:

- node controller, responsible for noticing and responding when nodes go down
- job controller, watches for job objects that represent one-off tasks, then creates Pods to run those tasks to completion.
- endpoints controller, populates the endpoints object
- service account & token controllers, create default accounts and API access tokens for new namespaces

7.2.4. *cloud-controller-manager.* A Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platfarm from components that only interact with your cluster. As with the kube-controller-manager, the cloud-controller-manager combines several logically independent control loops into a single binary that you run as a single process. You can scale horizontally - run more than one copy - to improve performance or to help tolerate failures. The following controllers can have cloud provider dependencies:

- node controller, for checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
- route controller, for setting up routes in the underlying cloud infrastructure

- service controller, for creating, updating and deleting cloud provider load balancers

7.3. **Node Components.** Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

7.3.1. *Kube-proxy.* Kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept. Kube-proxy maintains network rules on nodes. These network rules allow network communication to your pods from network sessions inside or outside of your cluster. Kube-proxy uses the operating system packet filtering layer if there is one and it is available. Otherwise, kube-proxy forwards the traffic itself.

7.3.2. *Container runtime.* The container runtime is the software that is responsible for running containers. Kubernetes supports several container runtimes: Docker, Containerd, and any implementation of the Kubernetes CRI (Container Runtime lnterface).

7.4. **Addons: resource monitoring.** You can examine application performance in a Kubernetes cluster by examining the containers, pods, services, and the characteristics of the overall cluster. Kubernetes provides detailed information about an application resource usage at each of these levels.

8. Cloud storage

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**. The instruction sets of most processors support the **test-and-set** instruction, which writes to a memory location and returns the old content of that memory cell as non-interruptible operations. Other architectures support **compare-and-swap**, an atomic instruction that compares the contents of a memory location to a given value and, only if the two values are the same, modifies the contents of that memory location to a given new value.

Two flavors of atomicity can be distinguished: all-or-nothing 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. **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.

8.1.1. *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. 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. Once the content of a cell is changed by an action, there is no way to abort the action and restore the original content of the cell. To be able to restore a previous value we have to maintain a version history for each variable in the cell storage.

Journal storage consists of a manager and cell storage, where the entire history of a variable is maintained, rather than just the current value. The user does not have direct access to the cell storage; instead the user can request the journal manager to (i) start a new **action**; (ii) read the value of a cell; (iii) write the value of a cell; (iv) **commit** an action; or (v) **abort** an action. An all-or-nothing action first records the action in a log in journal storage and then installs the change in the cell storage by overwriting the previous version of a data item.

Many cloud applications must support online transaction processing and have to guarantee the correctness of the transactions. Transactions consist of multiple actions and the system may fail during or after each one of the actions, and steps to ensure correctness must be taken. Correctness of a transaction means that the result should be guaranteed to be the same as though the actions were applied one after another, regardless of the order. To guarantee correctness, a transaction-processing system supports all-or-nothing atomicity.

8.2. **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. 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.

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 the master 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. 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.

8.3. **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 which replicates data on multiple nodes. The default is three replicas; a large dataset is distributed over many nodes. The **nameNode**, running on the master node, manages the data distribution and data replication and communicates with **dataNodes** 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.

8.3.1. *File I/O operations.* An application adds data to HDFS by creating a new file and writing the data to it. After the file is closed, the bytes written cannot be altered or removed except that new data can be added to the file by reopening the file for append. HDFS implements a single-writer, multiple-reader model.

An HDFS file consists of blocks. When there is a need for a new block, the NameNode allocates a block and determines a list of dataNodes to host replicas of the block. The dataNodes form a **pipeline**, the order of which minimizes the total network distance from the client to the last DataNode. Bytes are pushed to the pipeline as a sequence of packets. After a packet buffer is filled (typically 64 KB), the data are pushed to the pipeline. The next packet can be pushed to the

pipeline before receiving the acknowledgement for the previous packets. After data are written to an HDFS file, HDFS does not pro- vide any guarantee that data are visible to a new reader until the file is closed. If a user application needs the visibility guarantee, it can explicitly call the hflush operation. Then the current packet is immediately pushed to the pipeline, and the hflush operation will wait until all dataNodes in the pipeline acknowledge the successful transmission of the packet.

When a client opens a file to read, it fetches the list of blocks and the locations of each block replica from the NameNode. The locations of each block are ordered by their distance from the reader. A read may fail if the target DataNode is unavailable, the node no longer hosts a replica of the block, or the replica is found to be corrupt when checksums are tested.

The design of HDFS I/O is particularly optimized for batch processing systems, like MapReduce, which require high throughput for sequential reads and writes.

8.3.2. *Block Placement and replicas.* For a large cluster, it may not be practical to connect all nodes in a flat topology. A common practice is to spread the nodes across multiple racks. In most cases, network bandwidth between nodes in the same rack is greater than network band- width between nodes in different racks.

The placement of replicas is critical to HDFS data reliability and read/write performance. A good replica placement policy should improve data reliability, availability, and network bandwidth utilization. The default HDFS block placement policy provides a tradeoff between minimizing the write cost, and maximizing data reliability, availability and aggregate read bandwidth. When a new block is created, HDFS places the first replica on the node where the writer is located, the second and the third replicas on two different nodes in a different rack, and the rest are placed on random nodes with some restrictions:

- no datanode contains more than one replica of any block
- no rack contains more than two replicas of the same block, provided there are sufficient racks on the cluster

This policy reduces the inter-rack and inter-node write traffic and generally improves write performance. Because the chance of a rack failure is far less than that of a node failure, this policy does not impact data reliability and availability guarantees.

8.4. **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. A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map $(\text{row} : string, \text{column} : string, \text{timestamp} : int64) \to string$.

The **master** is responsible for assigning tablets to tablet servers, detecting the addition and expiration of tablet servers, balancing tablet-server load, and garbage collection of files in GFS. In addition, it handles schema changes such as table and column family creations. As with many single-master distributed storage systems, client data does not move through the master: clients communicate directly with tablet servers for reads and writes.

A Bigtable cluster stores a number of **tables**. Each table consists of a set of **tablets**, and each tablet contains all data associated with a row range. Initially,

each table consists of just one tablet. As a table grows, it is automatically split into multiple tablets. Each **tablet server** manages a set of tablets (typically we have somewhere between ten to a thousand tablets per tablet server). The tablet server handles read and write requests to the tablets that it has loaded, and also splits tablets that have grown too large.

Each tablet is assigned to one **tablet server** at a time. The **master** keeps track of the set of live tablet servers, and the current assignment of tablets to tablet servers, including which tablets are unassigned. The master is responsible for detecting when a tablet server is no longer serving its tablets, and for reassigning those tablets as soon as possible. When a master is started by the cluster management system, it needs to discover the current tablet assignments before it can change them.

To make the management of versioned data less onerous, BigTable can garbage-collect cell versions automatically. The client can specify either that only the last n versions of a cell be kept, or that only new-enough versions be kept.

8.5. **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. 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 ensures that data will be "eventually consistent" at some future point in time instead of enforcing consistency at the time when a transaction is committed. Data partitioning among multiple storage servers and data replication are also tenets of the NoSQL philosophy; they increase availability, reduce response time, and enhance scalability.

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

8.6.1. *Query model.* Simple read and write operations to a data item that is uniquely identified by a key. State is stored as binary objects (i.e. blobs) identified by unique keys. No operations span multiple data items and there is no need for relational schema. This requirement is based on the observation that a significant portion

of Amazon's services can work with this simple query model and do not need any relational schema.

8.6.2. *ACID properties.* ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction. Experience at Amazon has shown that data stores that provide ACID guarantees tend to have poor availability. Dynamo targets applications that operate with weaker consistency (the "C" in ACID) if this results in high availability. Dynamo does not provide any isolation guarantees and permits only single key updates.

8.6.3. *Efficiency and other Assumptions.* The system needs to function on a commodity hardware infrastructure. Services must be able to configure Dynamo such that they consistently achieve their latency and throughput requirements. The tradeoffs are in performance, cost efficiency, availability, and durability guarantees. Furthermore, Dynamo operation environment is assumed to be non-hostile and there are no security related requirements such as authentication and authorization.

8.6.4. *Design considerations.* Dynamo is designed to be an eventually consistent data store; that is all updates reach all replicas eventually. An important design consideration is to decide when to perform the process of resolving update conflicts, i.e., whether conflicts should be resolved during reads or writes. Dynamo targets the design space of an "always writeable" data store, forcing the complexity of conflict resolution to the reads in order to ensure that writes are never rejected.

The next design choice is who performs the process of conflict resolution. This can be done by the data store or the application. If conflict resolution is done by the data store, its choices are rather limited. In such cases, the data store can only use simple policies, such as "last write wins", to resolve conflicting updates. On the other hand, since the application is aware of the data schema it can decide on the conflict resolution method that is best suited for its client's experience. Other considerations:

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

Dynamo stores objects associated with a key through a simple interface; it exposes two operations: $get(\,)$ and $put(\,)$. The $get(key)$ operation locates the object replicas associated with the key in the storage system and returns a single object or a list of objects with conflicting versions along with a context. The $put(key, context, object)$ operation determines where the replicas of the object should be placed based on the associated key, and writes the replicas to disk. The context encodes system

metadata about the object that is opaque to the caller and includes information such as the version of the object. The context information is stored along with the object so that the system can verify the validity of the context object supplied in the put request.

8.6.5. *Partitioning algorithm.* One of the key design requirements for Dynamo is that it must scale incrementally. This requires a mechanism to dynamically partition the data over the set of nodes in the system. Dynamo's partitioning scheme relies on consistent hashing to distribute the load across multiple storage hosts. In consistent hashing, the output range of a hash function is treated as a fixed circular space or "ring". Each node in the system is assigned a random value within this space which represents its "position" on the ring. Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item's position.

The basic consistent hashing algorithm presents some challenges. First, the random position assignment of each node on the ring leads to non-uniform data and load distribution. Second, the basic algorithm is oblivious to the heterogeneity in the performance of nodes. To address these issues, Dynamo uses a variant of consistent hashing: instead of mapping a node to a single point in the circle, each node gets assigned to multiple points in the ring. To this end, Dynamo uses the concept of "virtual nodes". A virtual node looks like a single node in the system, but each node can be responsible for more than one virtual node. Effectively, when a new node is added to the system, it is assigned multiple positions (henceforth, "tokens") in the ring.

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

8.6.7. *Data versioning.* Dynamo provides eventual consistency, which allows for updates to be propagated to all replicas asynchronously. A *put*( ) call may return to its caller before the update has been applied at all the replicas, which can result in scenarios where a subsequent *get*( ) operation may return an object that does not have the latest updates. There is a category of applications in Amazon's platform that can tolerate such inconsistencies and can be constructed to operate under these conditions.

In order to provide this kind of guarantee, Dynamo treats the result of each modification as a new and immutable version of the data. It allows for multiple versions of an object to be present in the system at the same time. Most of the time, new versions subsume the previous version(s), and the system itself can determine the authoritative version. Dynamo uses vector clocks in order to capture causality between different versions of the same object.

9. Big data

9.1. **Google 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.

MapReduce is based on a very simple idea for parallel processing of data-intensive applications. 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 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

9.1.1. *Fault tolerance.* Any map tasks completed by failed worker are reset back to their initial idle state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to idle and becomes eligible for rescheduling. Completed map tasks are re-executed on a failure because their output is stored on the local disks of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be re-executed since their output is stored in a global file system.

Given that there is only a single master, its failure is unlikely; therefore standard implementations abort the MapReduce computation if the master fails. Clients can check for this condition and retry the MapReduce operation if they desire.

9.2. **Apache 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. *Shuffle and sort.* MapReduce makes the guarantee that the input to every reducer is sorted by key. The process by which the system performs the sort - and transfers the map outputs to the reducers as inputs - is known as the **shuffle**. When the map function starts producing output, before it writes to disk, the thread first divides the data into partitions corresponding to the reducers that they will ultimately be sent to. Within each partition, the background thread performs an in-memory sort by key.

The map tasks may finish at different times, so the reduce task starts copying their outputsas soon as each completes. This is known as the copy phase of the reduce task. When all the map outputs have been copied, the reduce task moves into the **sort** (or merge) phase, which merges the map outputs, maintaining their sort ordering. During the reduce phase, the reduce function is invoked for each key in the sorted output. The output of this phase is written directly to the output filesystem, typically HDFS.

9.2.2. *Grid computing.* The approach in high-performance computing is to distribute the work across a cluster of machines, which access a shared filesystem. This works well for predominantly compute-intensive jobs, but it becomes a problem when nodes need to access larger data volumes, since the network bandwidth is the bottleneck and compute nodes become idle. Hadoop tries to co-locate the data with the compute nodes, so data access is fast because it is local.

This feature, known as **data locality**, is at the heart of data processing in Hadoop and is the reason for its good performance. Recognizing that network bandwidth is the most precious resource in a data center environment (it is easy to saturate network links by copying data around), Hadoop goes to great lengths to conserve it by explicitly modeling network topology. If for some reason data locality is not possible, Hadoop mapReduce fallback to, in order, rack-local or off-rack deployment/execution.

9.3. **Apache 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.

9.4. **Apache Spark.** Models of computing like MapReduce allow to execute data-parallel computations on clusters of unreliable machines, by systems that automatically provide locality-aware scheduling, fault tolerance, and load balancing. While this data flow programming model is useful for a large class of applications, others cannot be expressed efficiently, i.e. those that reuse a working set of data across multiple parallel operations. This includes two use cases such as iterative jobs - common in ML algorithms, where a function is applied repeatedly on the same dataset - and interactive analytics - run ad-hoc exploratory queries on large datasets, through SQL interfaces.

Spark has a programming model similar to MapReduce but extends it with a data-sharing abstraction called Resilient Distributed Datasets (**RDD**), and support for parallel operations on these datasets. To use Spark, developers write a driver program that implements the high-level control flow of their application and launches various operations in parallel. Spark is designed to be used with multiple external systems for persistent storage.

9.4.1. *Architecture.*

9.4.2. *Resilient Distributed Datasets.* A RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Spark lets programmers construct RDDs in different ways:

- from a file in a shared file system, such as HDFS
- by "parallelizing" a collection in the driver program, which means dividing it into a number of slices that will be sent to multiple nodes
- by transforming an existing RDD

Spark evaluates RDDs lazily, allowing it to find an efficient plan for the user's computation. When an action is called, Spark looks at the whole graph of transformations used to create an execution plan. Finally, RDDs provide explicit support for data sharing among computations, by avoiding replication of intermediate data.

RDDs are "ephemeral" by default, in that they get recomputed each time they are used in an action. However, users can also persist selected RDDs in storage memory or cache them in main memory.

RDDs also automatically recover from failures. Traditionally, distributed computing systems have provided fault tolerance through data replication or checkpointing. Spark uses a different approach called **lineage**. Each RDD tracks the graph of transformations that was used to build it and reruns these operations on base data to reconstruct any lost partitions. Lineage-based recovery is significantly more efficient than replication in data-intensive workloads.

RDDs are best suited for batch applications that apply the same operation to all elements of a dataset. In these cases, RDDs can efficiently remember each transformation as one step in a lineage graph and can recover lost partitions without having to log large amounts of data. RDDs would be less suitable for applications that make asynchronous finegrained updates to shared state, such as a storage system for a web application or an incremental web crawler. For these applications, it is more efficient to use systems that perform traditional update logging and data checkpointing.

9.4.3. *Higher-level libraries.* The RDD programming model provides only distributed collections of objects and functions to run on them. Using RDDs, Spark developers have built a variety of higher-level libraries, targeting many of the use cases of specialized computing engines.

One of the most common data processing paradigms is relational queries. **Spark SQL** implement such queries on Spark, using techniques similar to analytical databases (compressed columnar storage). Furthermore, Spark provides a higher-level abstraction for basic data transformations called **DataFrames**, which are RDDs of records with a known schema. **Spark Streaming** implements incremental stream processing using a model called "discretized streams". To implement streaming over Spark, input data is split into small batches, which is regularly combine with state stored inside RDDs to produce new results. **GraphX** provides a graph computation interface. **MLlib** for distributed machine learning model training.