Cloud computing at a glance

In 1969, Leonard Kleinrock, one of the chief scientists of the original Advanced Research Projects Agency Network (ARPANET), which seeded the Internet, said:

As *of now, computer networks are still in their infancy, but as they grow up and become sophisticated, we will probably see the spread of* 'computer utilities' *which, like present electric and telephone utilities, will service individual homes and offices across the country.*

This vision of computing utilities based on a service-provisioning model anticipated the massive transformation of the entire computing industry in the 21st century, whereby computing services will be readily available on demand, just as other utility services such as water, electricity, telephone, and gas are available in today's society. Similarly, users (consumers) need to pay providers only when they access the computing services. In addition, consumers no longer need to invest heavily or encounter difficulties in building and maintaining complex IT infrastructure.

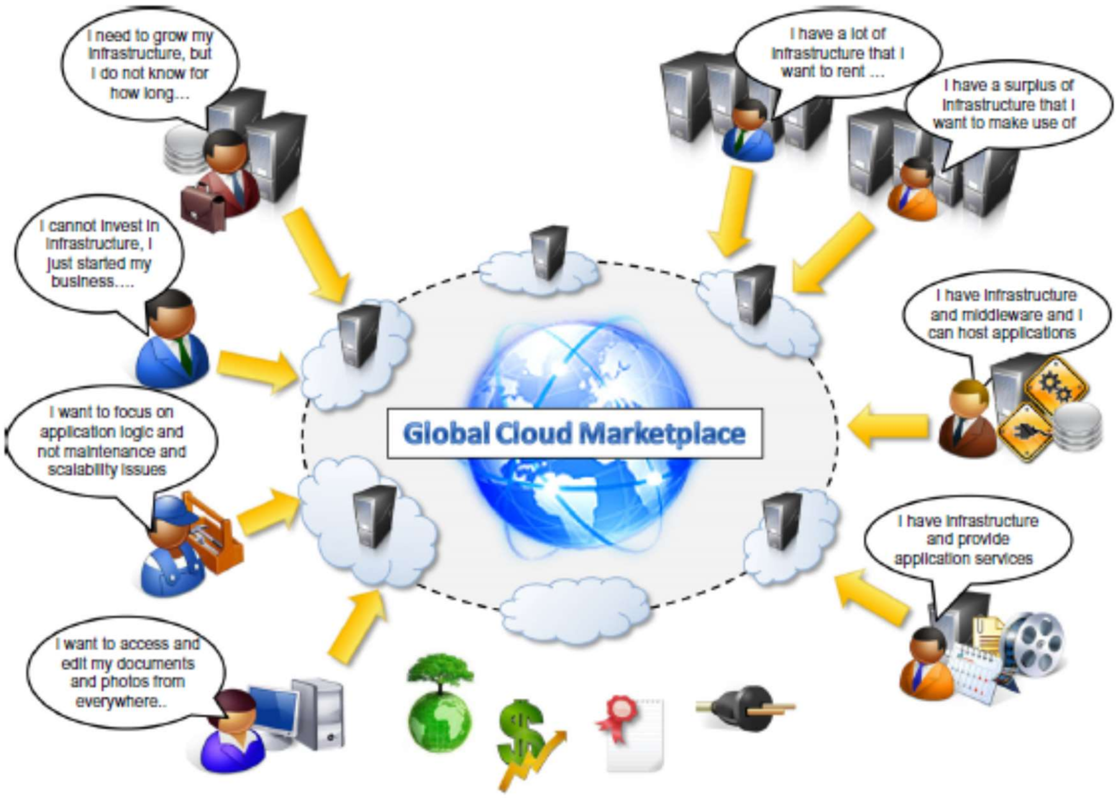
In such a model, users access services based on their requirements without regard to where the services are hosted. This model has been referred to as *utility computing* or, recently (since 2007), as *cloud computing.* The latter term often denotes the infrastructure as a "cloud" from which businesses and users can access applications as services from anywhere in the world and on demand. Hence, cloud computing can be classified as a new paradigm for the dynamic provisioning of computing services supported by state-of-the-art data centers employing virtualization technologies for consolidation and effective utilization of resources.
Cloud computing allows renting infrastructure, runtime environments, and services on a pay- per-use basis. This principle finds several practical applications and then gives different images of cloud computing to different people. Chief information and technology officers of large enterprises see opportunities for scaling their infrastructure on demand and sizing it according to their business needs. End users leveraging cloud computing services can access their documents and data anytime, anywhere, and from any device connected to the Internet. Many other points of view exist.

Besides being an extremely flexible environment for building new systems and applications, cloud computing also provides an opportunity for integrating additional capacity or new features into existing systems. The use of dynamically provisioned IT resources constitutes a more attractive opportunity than buying additional infrastructure and software, the sizing of which can be difficult to estimate and the needs of which are limited in time. This is one of the most important advantages of cloud computing, which has made it a popular phenomenon. With the wide deployment of cloud computing systems, the foundation technologies and systems enabling them are becoming consolidated and standardized. This is a fundamental step in the realization of the long-term vision for cloud computing, which provides an open environment where computing, storage, and other services are traded as computing utilities.

The vision of cloud computing

Cloud computing allows anyone with a credit card to provision virtual hardware, runtime environments, and services. These are used for as long as needed, with no up-front commitments required. The entire stack of a computing system is transformed into a collection of utilities, which can be provisioned and composed together to deploy systems in hours rather than days and with virtually no maintenance costs. This opportunity, initially met with skepticism, has now become a practice across several application domains and business sectors (see Figure). The demand has fast- tracked technical development and enriched the set of services offered, which have also become more sophisticated and cheaper.

Despite its evolution, the use of cloud computing is often limited to a single service at a time or, more commonly, a set of related services offered by the same vendor. Previously, the lack of effective standardization efforts made it difficult to move hosted services from one vendor to another. The long-term vision of cloud computing is that IT services are traded as utilities in an open market, without technological and legal barriers. In this cloud marketplace, cloud service providers and consumers, trading cloud services as utilities, play a central role.

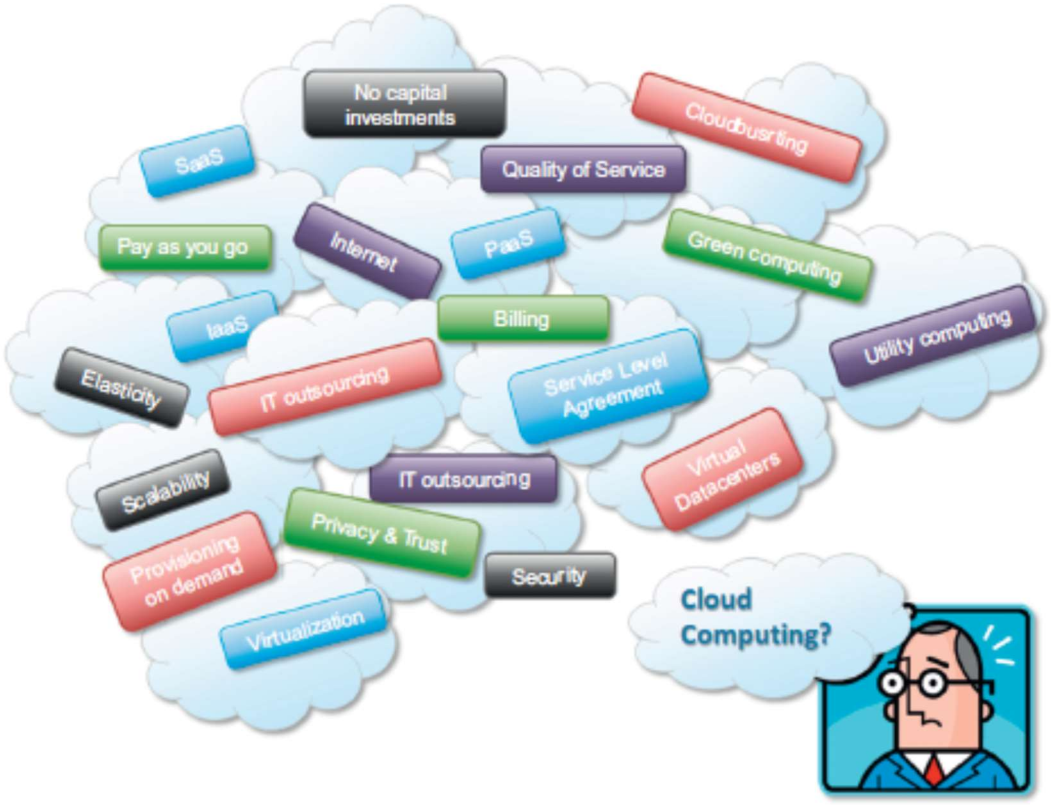

I need to grow my infrastructure, but I do not know for how long....
I cannot invest in infrastructure, I just started my business....
I want to focus on application logic and not maintenance and scalability issues
I want to access and edit my documents and photos from everywhere..
I have a lot of infrastructure that I want to rent ...
I have a surplus of infrastructure that I want to make use of
I have infrastructure and middleware and I can host applications
I have infrastructure and provide application services
Global Cloud Marketplace

Many of the technological elements contributing to this vision already exist. Different stakeholders leverage clouds for a variety of services. The need for ubiquitous storage and compute power on demand is the most common reason to consider cloud computing. A scalable runtime for applications is an attractive option for application and system developers that do not have infrastructure or cannot afford any further expansion of existing infrastructure. The capability for Web- based access to documents and their processing using sophisticated applications is one of the appealing factors for end users.

In all these cases, the discovery of such services is mostly done by human intervention: a person (or a team of people) looks over the Internet to identify offerings that meet his or her needs. We imagine that in the near future it will be possible to find the solution that matches our needs by simply entering our request in a global digital market that trades cloud computing services. The existence of such a market will enable the automation of the discovery process and its integration into existing software systems, thus allowing users to transparently leverage cloud resources in their applications and systems. The existence of a global platform for trading cloud services will also help service providers become more visible and therefore potentially increase their revenue. A global cloud market also reduces the barriers between service consumers and providers: it is no longer necessary to belong to only one of these two categories. For example, a cloud provider might become a consumer of a competitor service in order to fulfill its own promises to customers.

Defining a cloud

Cloud computing has become a popular buzzword; it has been widely used to refer to different technologies, services, and concepts. It is often associated with virtualized infrastructure or hardware on demand, utility computing, IT outsourcing, platform and software as a service, and many other things that now are the focus of the IT industry. Figure depicts the plethora of different notions included in current definitions of cloud computing.



The term *cloud* has historically been used in the telecommunications industry as an abstraction of the network in system diagrams. It then became the symbol of the most popular computer network: the Internet. This meaning also applies to *cloud computing,* which refers to an Internet-centric way of computing. The Internet plays a fundamental role in cloud computing, since it represents either the medium or the platform through which many cloud computing services are delivered and made accessible. This aspect is also reflected in the definition given by Armbrust et al.:

Cloud computing refers to both the applications delivered as services over the Internet and the hardware and system software in the datacenters that provide those services.

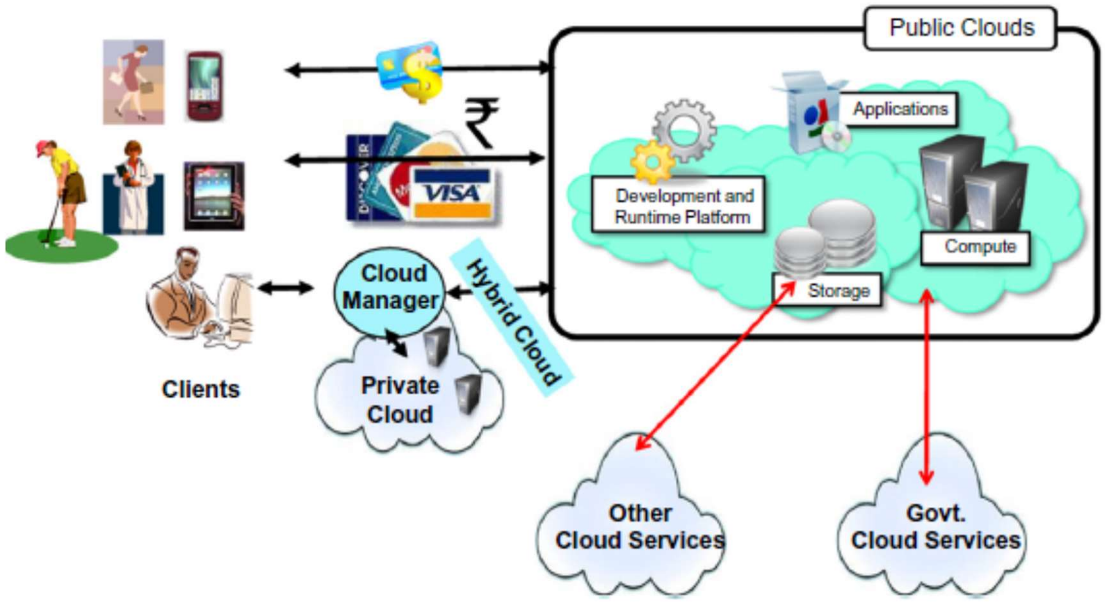
This definition describes cloud computing as a phenomenon touching on the entire stack: from the underlying hardware to the high-level software services and applications. It introduces the concept of *everything as a service,* mostly referred as *XaaS* where the different components of a system—IT infrastructure, development platforms, databases, and so on—can be delivered, measured, and consequently priced as a service. 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. The approach fostered by cloud computing is global: it covers both the needs of a single user hosting documents in the cloud and the ones of a CIO deciding to deploy part of or the entire corporate IT infrastructure in the public cloud. This notion of multiple parties using a shared cloud computing environment is highlighted in a definition proposed by the U.S. National Institute of Standards and Technology (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.*

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. All these operations can be performed and billed simply by entering the credit card details and accessing the exposed services through a Web browser.

The cloud computing reference model

A fundamental characteristic of cloud computing is the capability to deliver, on demand, a variety of IT services that are quite diverse from each other. This variety creates different perceptions of what cloud computing is among users. Despite this lack of uniformity, it is possible to classify cloud computing services offerings into three major categories: *Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS),* and *Software-as-a-Service (SaaS).* These categories are related to each other as described in Figure, which provides an organic view of cloud computing.



At the base of the stack, *Infrastructure-as-a-Service* solutions deliver infrastructure on demand in the form of virtual *hardware, storage,* and *networking.* Virtual hardware is utilized to provide compute on demand in the form of virtual machine instances. These are created at users' request on the provider's infrastructure, and users are given tools and interfaces to configure the software stack installed in the virtual machine. The pricing model is usually defined in terms of dollars per hour, where the hourly cost is influenced by the characteristics of the virtual hardware. Virtual storage is delivered in the form of raw disk space or object store. The former complements a virtual hardware offering that requires persistent storage. The latter is a more high-level abstraction for storing entities rather than files. Virtual networking identifies the collection of services that manage the networking among virtual instances and their connectivity to the Internet or private networks.

Platform-as-a-Service solutions are the next step in the stack. They deliver scalable and elastic runtime environments on demand and host the execution of applications. These services are backed by a core middleware platform that is responsible for creating the abstract environment where applications are deployed and executed. It is the responsibility of the service provider to provide scalability and to manage fault tolerance, while users are requested to focus on the logic of the application developed by leveraging the provider's APIs and libraries. This approach increases the level of abstraction at which cloud computing is leveraged but also constrains the user in a more controlled environment.

At the top of the stack, *Software-as-a-Service* solutions provide applications and services on demand. Most of the common functionalities of desktop applications—such as office automation, document management, photo editing, and customer relationship management (CRM) software—are replicated on the provider's infrastructure and made more scalable and accessible through a browser on demand. These applications are shared across multiple users whose interaction is isolated from the other users. The SaaS layer is also the area of social networking Websites, which leverage cloud-based infrastructures to sustain the load generated by their popularity.

Each layer provides a different service to users. IaaS solutions are sought by users who want to leverage cloud computing from building dynamically scalable computing systems requiring a specific software stack. IaaS services are therefore used to develop scalable Websites or for background processing. PaaS solutions provide scalable programming platforms for developing applications and are more appropriate when new systems have to be developed. SaaS solutions target mostly end users who want to benefit from the elastic scalability of the cloud without doing any software development, installation, configuration, and maintenance. This solution is appropriate when there are existing SaaS services that fit users needs (such as email, document management, CRM, etc.) and a minimum level of customization is needed.

Characteristics and benefits

Cloud computing has some interesting characteristics that bring benefits to both cloud service consumers (CSCs) and cloud service providers (CSPs). These characteristics are:

- No up-front commitments
- On-demand access
- Nice pricing
- Simplified application acceleration and scalability

- Efficient resource allocation
- Energy efficiency
- Seamless creation and use of third-party services

The most evident benefit from the use of cloud computing systems and technologies is the increased economical return due to the reduced maintenance costs and *operational costs* related to IT software and infrastructure. This is mainly because IT assets, namely software and infrastructure, are turned into *utility costs,* which are paid for as long as they are used, not paid for up front. Capital costs are costs associated with assets that need to be paid in advance to start a business activity. Before cloud computing, IT infrastructure and software generated capital costs, since they were paid up front so that business start-ups could afford a computing infrastructure, enabling the business activities of the organization. The revenue of the business is then utilized to compensate over time for these costs. Organizations always minimize capital costs, since they are often associated with depreciable values. This is the case of hardware: a server bought today for $1,000 will have a market value less than its original price when it is eventually replaced by new hardware. To make profit, organizations have to compensate for this depreciation created by time, thus reducing the net gain obtained from revenue. Minimizing capital costs, then, is fundamental. Cloud computing transforms IT infrastructure and software into utilities, thus significantly contributing to increasing a company's net gain. Moreover, cloud computing also provides an opportunity for small organizations and start-ups: these do not need large investments to start their business, but they can comfortably grow with it. Finally, maintenance costs are significantly reduced: by renting the infrastructure and the application services, organizations are no longer responsible for their maintenance. This task is the responsibility of the cloud service provider, who, thanks to economies of scale, can bear the maintenance costs.

End users can benefit from cloud computing by having their data and the capability of operating on it always available, from anywhere, at any time, and through multiple devices. Information and services stored in the cloud are exposed to users by Web-based interfaces that make them accessible from portable devices as well as desktops at home. Since the processing capabilities (that is, office automation features, photo editing, information management, and so on) also reside in the cloud, end users can perform the same tasks that previously were carried out through considerable software investments. The cost for such opportunities is generally very limited, since the cloud service provider shares its costs across all the tenants that he is servicing. Multitenancy allows for better utilization of the shared infrastructure that is kept operational and fully active. The concentration of IT infrastructure and services into large datacenters also provides opportunity for considerable optimization in terms of resource allocation and energy efficiency, which eventually can lead to a less impacting approach on the environment.

Finally, service orientation and on-demand access create new opportunities for composing systems and applications with a flexibility not possible before cloud computing. New service offerings can be created by aggregating together existing services and concentrating on added value. Since it is possible to provision on demand any component of the computing stack, it is easier to turn ideas into products with limited costs and by concentrating technical efforts on what matters: the added value.

Challenges ahead

As any new technology develops and becomes popular, new issues have to be faced. Cloud computing is not an exception. New, interesting problems and challenges are regularly being posed to the cloud community, including IT practitioners, managers, governments, and regulators.

Besides the practical aspects, which are related to configuration, networking, and sizing of cloud computing systems, a new set of challenges concerning the dynamic provisioning of cloud computing services and resources arises. For example, in the Infrastructure-as-a-Service domain, how many resources need to be provisioned, and for how long should they be used, in order to maximize the benefit? Technical challenges also arise for cloud service providers for the management of large computing infrastructures and the use of virtualization technologies on top of them. In addition, issues and challenges concerning the integration of real and virtual infrastructure need to be taken into account from different perspectives, such as security and legislation.

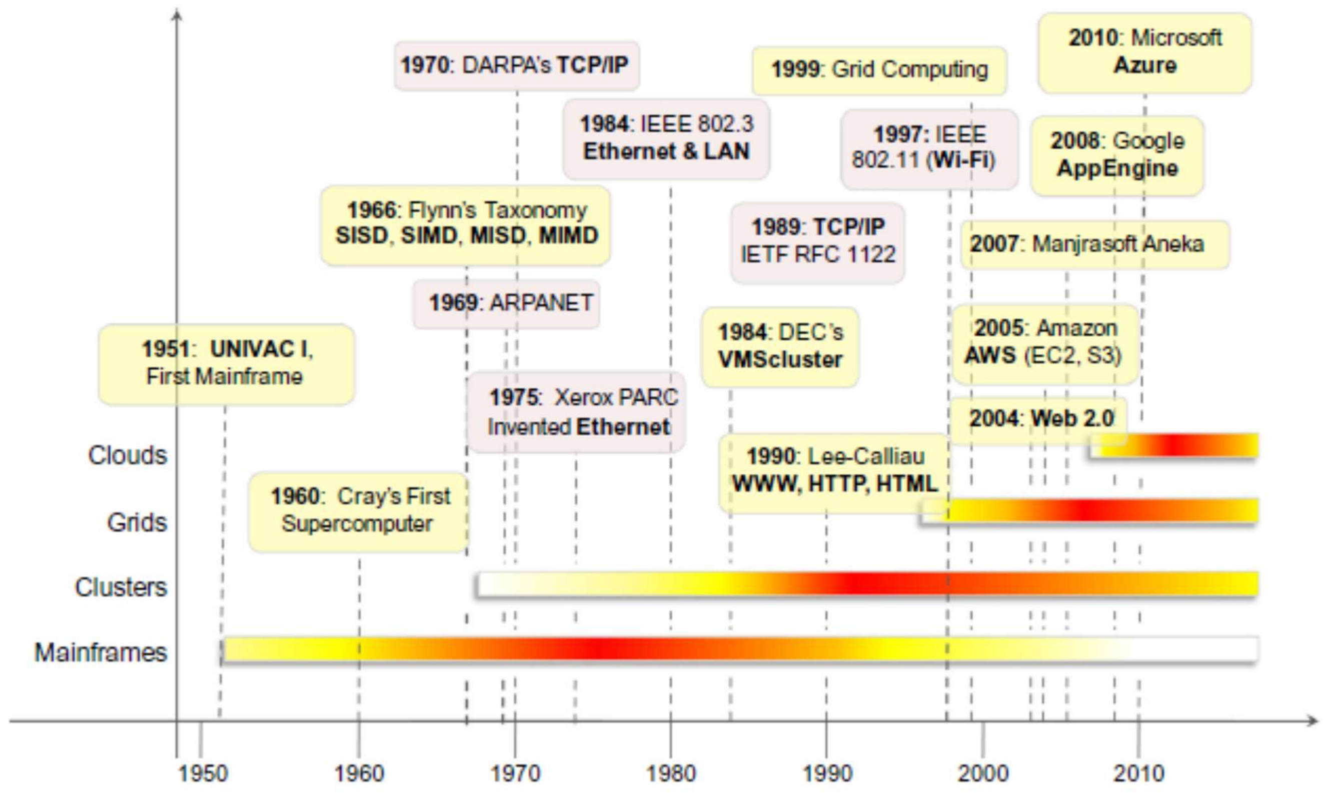
Security in terms of confidentiality, secrecy, and protection of data in a cloud environment is another important challenge. Organizations do not own the infrastructure they use to process data and store information. This condition poses challenges for confidential data, which organizations cannot afford to reveal. Therefore, assurance on the confidentiality of data and compliance to security standards, which give a minimum guarantee on the treatment of information on cloud computing systems, are sought. The problem is not as evident as it seems: even though cryptography can help secure the transit of data from the private premises to the cloud infrastructure, in order to be processed the information needs to be decrypted in memory. This is the weak point of the chain: since virtualization allows capturing almost transparently the memory pages of an instance, these data could easily be obtained by a malicious provider.

Legal issues may also arise. These are specifically tied to the ubiquitous nature of cloud computing, which spreads computing infrastructure across diverse geographical locations. Different legislation about privacy in different countries may potentially create disputes as to the rights that third parties (including government agencies) have to your data.

Historical developments

The idea of renting computing services by leveraging large distributed computing facilities has been around for long time. It dates back to the days of the mainframes in the early 1950s. From there on, technology has evolved and been refined. This process has created a series of favorable conditions for the realization of cloud computing.

Figure 1.6 provides an overview of the evolution of the distributed computing technologies that have influenced cloud computing. In tracking the historical evolution, we briefly review five core technologies that played an important role in the realization of cloud computing. These technologies are distributed systems, virtualization, Web 2.0, service orientation, and utility computing.



Distributed systems

Clouds are essentially large distributed computing facilities that make available their services to third parties on demand. As a reference, we consider the characterization of a distributed system proposed by Tanenbaum et al.:

A distributed system is a collection of independent computers that appears to its users as a single coherent system.

Distributed systems often exhibit other properties such as *heterogeneity, openness, scalability, transparency*, *concurrency, continuous availability,* and *independent failures*. To some extent these also characterize clouds, especially in the context of scalability, concurrency, and continuous availability.

Three major milestones have led to cloud computing: mainframe computing, cluster computing, and grid computing.

- ***Mainframes***. These were the first examples of large computational facilities leveraging multiple processing units. Mainframes were powerful, highly reliable computers specialized for large data movement and massive input/output (I/O) operations. They were mostly used by large organizations for bulk data processing tasks such as online transactions, enterprise resource planning, and other operations involving the processing of significant amounts of data. Even though mainframes cannot be considered distributed systems, they offered large computational power by using multiple processors, which were presented as a single entity to users. One of the most attractive features of mainframes was the ability to be highly reliable computers that were "always on" and capable of tolerating failures transparently. No system shutdown was required to replace failed components, and the system could work without interruption. Batch processing was the main application of mainframes.

- ***Clusters.*** Cluster computing started as a low-cost alternative to the use of mainframes and supercomputers. The technology advancement that created faster and more powerful mainframes and supercomputers eventually generated an increased availability of cheap commodity machines as a side effect. These machines could then be connected by a high-bandwidth network and controlled by specific software tools that manage them as a single system. Starting in the 1980s, clusters become the standard technology for parallel and high-performance computing. Built by commodity machines, they were cheaper than mainframes and made high-performance computing available to a large number of groups, including universities and small research labs. Cluster technology contributed considerably to the evolution of tools and frameworks for distributed computing, including Condor, Parallel Virtual Machine (PVM), and Message Passing Interface (MPI). One of the attractive features of clusters was that the computational power of commodity machines could be leveraged to solve problems that were previously manageable only on expensive supercomputers. Moreover, clusters could be easily extended if more computational power was required.

- ***Grids.*** Grid computing appeared in the early 1990s as an evolution of cluster computing. In an analogy to the power grid, grid computing proposed a new approach to access large computational power, huge storage facilities, and a variety of services. Users can “consume” resources in the same way as they use other utilities such as power, gas, and water. Grids initially developed as aggregations of geographically dispersed clusters by means of Internet connections. These clusters belonged to different organizations, and arrangements were made among them to share the computational power. Different from a “large cluster,” a computing grid was a dynamic aggregation of heterogeneous computing nodes, and its scale was nationwide or even worldwide. Several developments made possible the diffusion of computing grids: (a) clusters became quite common resources; (b) they were often underutilized; (c) new problems were requiring computational power that went beyond the capability of single clusters; and (d) the improvements in networking and the diffusion of the Internet made possible long-distance, high-bandwidth connectivity. All these elements led to the development of grids, which now serve a multitude of users across the world.

Cloud computing is often considered the successor of grid computing. In reality, it embodies aspects of all these three major technologies. Computing clouds are deployed in large datacenters hosted by a single organization that provides services to others. Clouds are characterized by the fact of having virtually infinite capacity, being tolerant to failures, and being always on, as in the case of mainframes. In many cases, the computing nodes that form the infrastructure of computing clouds are commodity machines, as in the case of clusters. The services made available by a cloud vendor are consumed on a pay-per-use basis, and clouds fully implement the utility vision introduced by grid computing.

Virtualization

Virtualization is 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, runtime environments, storage, and networking. Virtualization has been around for more than 40 years, but its application has always been limited by technologies that did not allow an efficient use of virtualization solutions. Today these limitations have been substantially overcome, and virtualization has become a fundamental element of cloud computing. This is particularly true for solutions that provide IT infrastructure on demand. Virtualization confers that degree of customization and control that makes cloud computing appealing for users and, at the same time, sustainable for cloud services providers.

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. The most common example of virtualization is *hardware virtualization.* This technology allows simulating the hardware interface expected by an operating system. 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 complete isolation from each other. High-performance servers can host several virtual machine instances, thus creating the opportunity to have a customized software stack on demand. This is the base technology that enables cloud computing solutions to deliver virtual servers on demand, such as Amazon EC2, RightScale, VMware vCloud, and others. Together with hardware virtualization, *storage* and *network virtualization* complete the range of technologies for the emulation of IT infrastructure.

Virtualization technologies are also used to replicate runtime environments for programs. Applications in the case of *process virtual machines* (which include the foundation of technologies such as Java or .NET), 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. This approach is used in cloud computing to provide a platform for scaling applications on demand, such as Google AppEngine and Windows Azure.

Web 2.0

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.* This term captures 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 by gaining Web-based access to all the functions that are normally found in desktop applications. These capabilities are obtained by integrating a collection of standards and technologies such as *XML, Asynchronous JavaScript and XML (AJAX), Web Services,* and others. These technologies allow us to build applications leveraging the contribution of users, who now become providers of content.

Furthermore, the capillary diffusion of the Internet opens new opportunities and markets for the Web, the services of which can now be accessed from a variety of devices: mobile phones, car dashboards, TV sets, and others. These new scenarios require an increased dynamism for applications, which is another key element of this technology. Web 2.0 applications are extremely dynamic: they improve continuously, and new updates and features are integrated at a constant rate by following the usage trend of the community. There is no need to deploy new software releases on the installed base at the client side. Users can take advantage of the new software features simply by interacting with cloud applications. Lightweight deployment and programming models are very important for effective support of such dynamism. Loose coupling is another fundamental property. New applications can be "synthesized" simply by composing existing services and integrating them, thus providing added value. This way it becomes easier to follow the interests of users. Finally, Web 2.0 applications aim to leverage the "long tail" of Internet users by making themselves available to everyone in terms of either media accessibility or affordability.

Examples of Web 2.0 applications are *Google Documents*, *Google Maps*, *Flickr*, *Facebook*, *Twitter, YouTube, de.li.cious, Blogger,* and *Wikipedia.* In particular, social networking Websites take the biggest advantage of Web 2.0. The level of interaction in Websites such as Facebook or Flickr would not have been possible without the support of AJAX, Really Simple Syndication (RSS), and other tools that make the user experience incredibly interactive. Moreover, community Websites harness the collective intelligence of the community, which provides content to the applications themselves: Flickr provides advanced services for storing digital pictures and videos, Facebook is a social networking site that leverages user activity to provide content, and Blogger, like any other blogging site, provides an online diary that is fed by users.

Today it is a mature platform for supporting the needs of cloud computing, which strongly leverages Web 2.0. Applications and frameworks for delivering *rich Internet applications (RIAs)* are fundamental for making cloud services accessible to the wider public. From a social perspective, Web 2.0 applications definitely contributed to making people more accustomed to the use of the Internet in their everyday lives and opened the path to the acceptance of cloud computing as a paradigm, whereby even the IT infrastructure is offered through a Web interface.

Service-oriented computing

Service orientation is the core reference model for cloud computing systems. This approach adopts the concept of services as the main building blocks of application and system development. *Service-oriented computing (SOC)* supports the development of rapid, low-cost, flexible, interoperable, and evolvable applications and systems.

A *service* is an abstraction representing a self-describing and platform-agnostic component that can perform any function—anything from a simple function to a complex business process. Virtually any piece of code that performs a task can be turned into a service and expose its functionalities through a network-accessible protocol. A service is supposed to be *loosely coupled*, *reusable*, *programming language independent*, and *location transparent*. Loose coupling allows services to serve different scenarios more easily and makes them reusable. Independence from a specific platform increases services accessibility. Thus, a wider range of clients, which can look up services in global registries and consume them in a location-transparent manner, can be served. Services are composed and aggregated into a *service-oriented architecture (SOA)*, which is a logical way of organizing software systems to provide end users or other entities distributed over the network with services through published and discoverable interfaces.

Service-oriented computing introduces and diffuses two important concepts, which are also fundamental to cloud computing: *quality of service (QoS)* and *Software-as-a-Service (SaaS).*

- Quality of service (QoS) identifies a set of functional and nonfunctional attributes that can be used to evaluate the behavior of a service from different perspectives. These could be performance metrics such as response time, or security attributes, transactional integrity, reliability, scalability, and availability. QoS requirements are established between the client and the provider via an SLA that identifies the minimum values (or an acceptable range) for the QoS attributes that need to be satisfied upon the service call.
- The concept of Software-as-a-Service introduces a new delivery model for applications. The term has been inherited from the world of application service providers (ASPs), which deliver software services-based solutions across the wide area network from a central datacenter and make them available on a subscription or rental basis. The ASP is responsible for maintaining the infrastructure and making available the application, and the client is freed from maintenance costs and difficult upgrades. This software delivery model is possible because economies of scale are reached by means of multitenancy. The SaaS approach reaches its full development with service-oriented computing (SOC), where loosely coupled software components can be exposed and priced singularly, rather than entire applications. This allows the delivery of complex business processes and transactions as a service while allowing applications to be composed on the fly and services to be reused from everywhere and by anybody.

One of the most popular expressions of service orientation is represented by Web Services (WS). These introduce the concepts of SOC into the World Wide Web, by making it consumable by applications and not only humans. Web services are software components that expose functionalities accessible using a method invocation pattern that goes over the HyperText Transfer Protocol (HTTP). The interface of a Web service can be programmatically inferred by metadata expressed through the *Web Service Description Language (WSDL)*; this is an XML language that defines the characteristics of the service and all the methods, together with parameters, descriptions, and return type, exposed by the service. The interaction with Web services happens through *Simple Object Access Protocol (SOAP)*. This is an XML language that defines how to invoke a Web service method and collect the result. Using SOAP and WSDL over HTTP, Web services become platform independent and accessible to the World Wide Web. The standards and specifications concerning Web services are controlled by the World Wide Web Consortium (W3C). Among the most popular architectures for developing Web services we can note ASP.NET and Axis.

Utility-oriented computing

Utility computing is a vision of computing that defines a service-provisioning model for compute services in which resources such as storage, compute power, applications, and infrastructure are packaged and offered on a pay-per-use basis. The idea of providing computing as a *utility* like natural gas, water, power, and telephone connection has a long history but has become a reality today with the advent of cloud computing. Among the earliest forerunners of this vision we can include the American scientist

John McCarthy, who, in a speech for the Massachusetts Institute of Technology (MIT) centennial in 1961, observed:

If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility, just as the telephone system is a public utility ... The computer utility could become the basis of a new and important industry.

The first traces of this service-provisioning model can be found in the mainframe era. IBM and other mainframe providers offered mainframe power to organizations such as banks and government agencies throughout their datacenters. The business model introduced with utility computing brought new requirements and led to improvements in mainframe technology: additional features such as operating systems, process control, and user-metering facilities. The idea of computing as utility remained and extended from the business domain to academia with the advent of cluster computing.

worldwide scale and through simple interfaces. As already discussed, computing grids provided a planet-scale distributed computing infrastructure that was accessible on demand. Computing grids brought the concept of utility computing to a new level: market orientation. With utility computing accessible on a wider scale, it is easier to provide a trading infrastructure where grid products—storage, computation, and services—are bid for or sold. Moreover, e-commerce technologies provided the infrastructure support for utility computing. In the late 1990s a significant interest in buying any kind of good online spread to the wider public: food, clothes, multimedia products, and online services such as storage space and Web hosting. After the *dot-com bubble* burst, this interest reduced in size, but the phenomenon made the public keener to buy online services. As a result, infrastructures for online payment using credit cards become easily accessible and well proven.

From an application and system development perspective, service-oriented computing and *service-oriented architectures (SOAs)* introduced the idea of leveraging external services for performing a specific task within a software system. Applications were not only distributed, they started to be composed as a mesh of services provided by different entities. These services, accessible through the Internet, were made available by charging according to usage. SOC broadened the concept of what could have been accessed as a utility in a computer system: not only compute power and storage but also services and application components could be utilized and integrated on demand. Together with this trend, QoS became an important topic to investigate.

All these factors contributed to the development of the concept of utility computing and offered important steps in the realization of cloud computing, in which the vision of computing utilities comes to its full expression.

Building cloud computing environments

The creation of cloud computing environments encompasses both the development of applications and systems that leverage cloud computing solutions and the creation of frameworks, platforms, and infrastructures delivering cloud computing services.

1. Application development

Applications that leverage cloud computing benefit from its capability to dynamically scale on demand. One class of applications that takes the biggest advantage of this feature is that of *Web applications*. Their performance is mostly influenced by the workload generated by varying user demands. With the diffusion of Web 2.0 technologies, the Web has become a platform for developing rich and complex applications, including *enterprise applications* that now leverage the Internet as the preferred channel for service delivery and user interaction. These applications are characterized by complex processes that are triggered by the interaction with users and develop through the interaction between several tiers behind the Web front end. These are the applications that are mostly sensible to inappropriate sizing of infrastructure and service deployment or variability in workload.

Another class of applications that can potentially gain considerable advantage by leveraging cloud computing is represented by *resource-intensive applications.* These can be either data- intensive or compute-intensive applications. In both cases, considerable amounts of resources are required to complete execution in a reasonable timeframe. It is worth noticing that these large amounts of resources are not needed constantly or for a long duration. For example, *scientific applications* can require huge computing capacity to perform large-scale experiments once in a while, so it is not feasible to buy the infrastructure supporting them. In this case, cloud computing can be the solution. Resource-intensive applications are not interactive and they are mostly characterized by batch processing.

Cloud computing provides a solution for on-demand and dynamic scaling across the entire stack of computing. This is achieved by (a) providing methods for renting compute power, storage, and networking; (b) offering runtime environments designed for scalability and dynamic sizing; and (c) providing application services that mimic the behavior of desktop applications but that are completely hosted and managed on the provider side. All these capabilities leverage service orientation, which allows a simple and seamless integration into existing systems. Developers access such services via simple Web interfaces, often implemented through representational state transfer (REST) Web services. These have become well-known abstractions, making the development and management of cloud applications and systems practical and straightforward.

2. Infrastructure and system development

Distributed computing, virtualization, service orientation, and Web 2.0 form the core technologies enabling the provisioning of cloud services from anywhere on the globe. Developing applications and systems that leverage the cloud requires knowledge across all these technologies. Moreover, new challenges need to be addressed from design and development standpoints.

Distributed computing is a foundational model for cloud computing because cloud systems are distributed systems. Besides administrative tasks mostly connected to the accessibility of resources in the cloud, the extreme dynamism of cloud systems—where new nodes and services are provisioned on demand—constitutes the major challenge for engineers and developers. This characteristic is pretty peculiar to cloud computing solutions and is mostly addressed at the middleware layer of computing system. Infrastructure-as-a-Service solutions provide the capabilities to add and remove resources, but it is up to those who deploy systems on this scalable infrastructure to make use of such opportunities with wisdom and effectiveness. Platform-as-a-Service solutions embed into their core offering algorithms and rules that control the provisioning process and the lease of resources.

Web 2.0 technologies constitute the interface through which cloud computing services are delivered, managed, and provisioned. Besides the interaction with rich interfaces through the Web browser, Web services have become the primary access point to cloud computing systems from a programmatic standpoint. Therefore, service orientation is the underlying paradigm that defines the architecture of a cloud computing system. Cloud computing is often summarized with the acronym *XaaS—Everything-as-a-Service*—that clearly underlines the central role of service orientation. Despite the absence of a unique standard for accessing the resources serviced by different cloud providers, the commonality of technology smoothes the learning curve and simplifies the integration of cloud computing into existing systems.

Virtualization is another element that plays a fundamental role in cloud computing. This technology is a core feature of the infrastructure used by cloud providers. Developers of cloud applications need to be aware of the limitations of the selected virtualization technology and the implications on the volatility of some components of their systems.

3. Computing platforms and technologies

Development of a cloud computing application happens by leveraging platforms and frameworks that provide different types of services, from the bare-metal infrastructure to customizable applications serving specific purposes.

- ***Amazon web services (AWS)***

  AWS offers comprehensive cloud IaaS services ranging from virtual compute, storage, and networking to complete computing stacks. AWS is mostly known for its compute and storage-on-demand services, namely *Elastic Compute Cloud (EC2)* and *Simple Storage Service (S3).* EC2 provides users with customizable virtual hardware that can be used as the base infrastructure for deploying computing systems on the cloud. It is possible to choose from a large variety of virtual hardware configurations, including GPU and cluster instances. EC2 instances are deployed either by using the AWS console, which is a comprehensive Web portal for accessing AWS services, or by using the Web services API available for several programming languages. EC2 also provides the capability to save a specific running instance as an image, thus allowing users to create their own templates for deploying systems. These templates are stored into S3 that delivers persistent storage on demand. S3 is organized into buckets; these are containers of objects that are stored in binary form and can be enriched with attributes. Users can store objects of any size, from simple files to entire disk images, and have them accessible from everywhere.

  Besides EC2 and S3, a wide range of services can be leveraged to build virtual computing systems. including networking support, caching systems, DNS, database (relational and not) support, and others.

- ***Google AppEngine***

  Google AppEngine is a scalable runtime environment mostly devoted to executing Web applications. These take advantage of the large computing infrastructure of Google to dynamically scale as the demand varies over time. AppEngine provides both a secure execution environment and a collection of services that simplify the development of scalable and high-performance Web applications. These services include in-memory caching, scalable data store, job queues, messaging, and cron tasks. Developers can build and test applications on their own machines using the AppEngine software development kit (SDK), which replicates the production runtime environment and helps test and profile applications. Once development is complete, developers can easily migrate their application to AppEngine, set quotas to contain the costs generated, and make the application available to the world. The languages currently supported are Python, Java, and Go.

- ***Microsoft Azure***

Microsoft Azure is a cloud operating system and a platform for developing applications in the cloud. It provides a scalable runtime environment for Web applications and distributed applications in general. Applications in Azure are organized around the concept of roles, which identify a distribution unit for applications and embody the application's logic. Currently, there are three types of role: *Web role, worker role,* and *virtual machine role.* The Web role is designed to host a Web application, the worker role is a more generic container of applications and can be used to perform workload processing, and the virtual machine role provides a virtual environment in which the computing stack can be fully customized, including the operating systems. Besides roles, Azure provides a set of additional services that complement application execution, such as support for storage (relational data and blobs), networking, caching, content delivery, and others.

- ***Hadoop***

Apache Hadoop is an open-source framework that is suited for processing large data sets on commodity hardware. Hadoop is an implementation of MapReduce, an application programming model developed by Google, which provides two fundamental operations for data processing: *map* and *reduce.* The former transforms and synthesizes the input data provided by the user; the latter aggregates the output obtained by the map operations. Hadoop provides the runtime environment, and developers need only provide the input data and specify the map and reduce functions that need to be executed. Yahoo!, the sponsor of the Apache Hadoop project, has put considerable effort into transforming the project into an enterprise-ready cloud computing platform for data processing. Hadoop is an integral part of the Yahoo! cloud infrastructure and supports several business processes of the company. Currently, Yahoo! manages the largest Hadoop cluster in the world, which is also available to academic institutions.

- ***Force.com and Salesforce.com***

Force.com is a cloud computing platform for developing social enterprise applications. The platform is the basis for SalesForce.com, a Software-as-a-Service solution for customer relationship management. Force.com allows developers to create applications by composing ready-to-use blocks; a complete set of components supporting all the activities of an enterprise are available. It is also possible to develop your own components or integrate those available in *AppExchange* into your applications. The platform provides complete support for developing applications, from the design of the data layout to the definition of business rules and workflows and the definition of the user interface. The Force.com platform is completely hosted on the cloud and provides complete access to its functionalities and those implemented in the hosted applications through Web services technologies.

- ***Manjrasoft Aneka***

Manjrasoft Aneka is a cloud application platform for rapid creation of scalable applications and their deployment on various types of clouds in a seamless and elastic manner. It supports a collection of programming abstractions for developing applications and a distributed runtime environment that can be deployed on heterogeneous hardware (clusters, networked desktop computers, and cloud resources). Developers can choose different abstractions to design their application: *tasks, distributed threads,* and *map-reduce.* These applications are then executed on the distributed service-oriented runtime environment, which can dynamically integrate additional resource on demand. The service-oriented architecture of the runtime has a great degree of flexibility and simplifies the integration of new features, such as abstraction of a new programming model and associated execution management environment. Services manage most of the activities happening at runtime: scheduling, execution, accounting, billing, storage, and quality of service.

Eras of computing

The two fundamental and dominant models of computing are sequential and parallel. The sequential computing era began in the 1940s; the parallel (and distributed) computing era followed it within a decade (As given in below fig.). The four key elements of computing developed during these eras are architectures, compilers, applications, and problem-solving environments.

The computing era started with a development in hardware architectures, which actually enabled the creation of system software—particularly in the area of compilers and operating systems—which support the management of such systems and the development of applications. The development of applications and systems are the major element of interest to us, and it comes to consolidation when problem-solving environments were designed and introduced to facilitate and empower engineers. This is when the paradigm characterizing the computing achieved maturity and became main stream. Moreover, every aspect of this era underwent a three-phase process: research and development (R&D), commercialization, and commoditization.

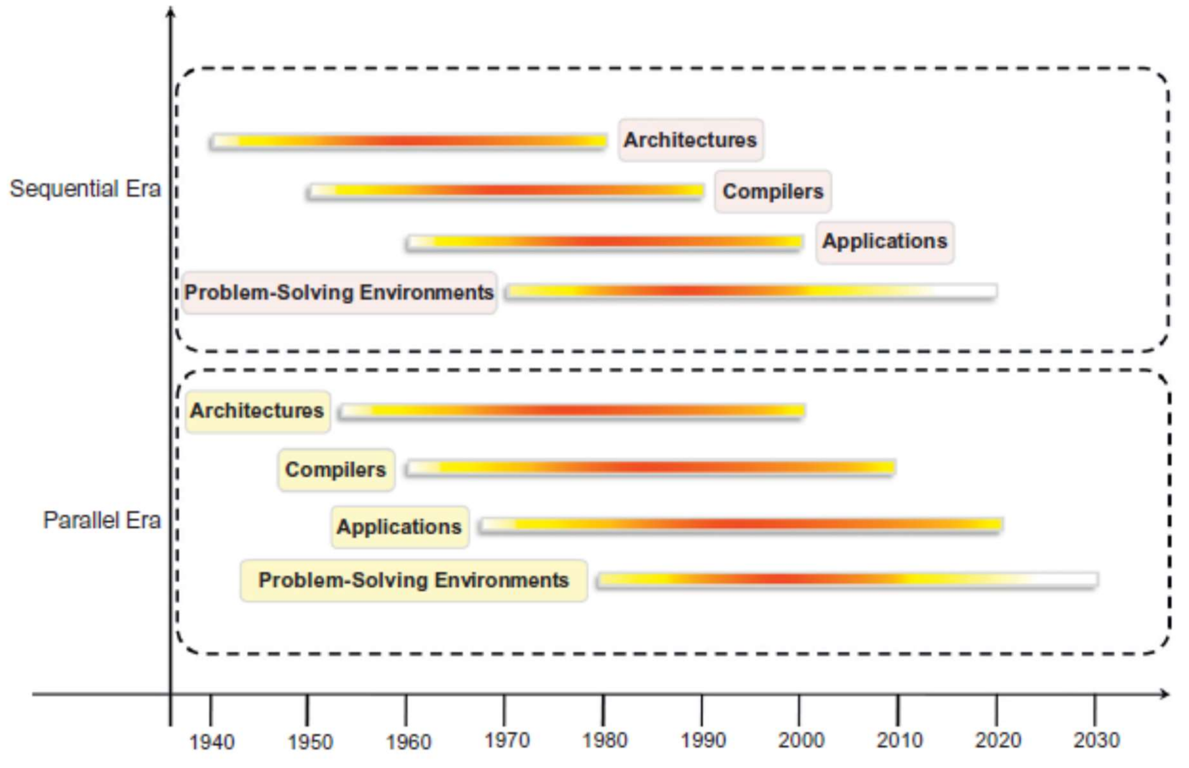


FIGURE 2.1

Eras of computing, 1940s–2030s.

Parallel vs distributed computing

The terms parallel computing and distributed computing are often used interchangeably, even though they means lightly different things. The term parallel implies a tightly coupled system, whereas distributed refers to a wider class of system, including those that are tightly coupled.

More precisely, the term parallel computing refers to a model in which the computation is divided among several processors sharing the same memory. The architecture of a parallel computing system is often characterized by the homogeneity of components: each processor is of the same type and it has the same capability as the others. The shared memory has a single address space, which is accessible to all the processors. Parallel programs are then broken down into several units of execution that can be allocated to different processors and can communicate with each other by means of the shared memory. Originally we considered parallel systems only those architectures that featured multiple processors sharing the same physical memory and that were considered a single computer. Overtime, these restrictions have been relaxed, and parallel systems now include all architectures that are based on the concept of shared memory, whether this is physically present or created with the support of libraries, specific hardware, and a highly efficient networking infrastructure. For example, a cluster of which the nodes are connected through an InfiniBand network and configured with a distributed shared memory system can be considered a parallel system.

The term *distributed computing* encompasses any architecture or system that allows the computation to be broken down into units and executed concurrently on different computing elements, whether these are processors on different nodes, processors on the same computer, or cores within the same processor. Therefore, distributed computing includes a wider range of systems and applications than parallel computing and is often considered a more general term. Even though it is not a rule, the term *distributed* often implies that the locations of the computing elements are not the same and such elements might be heterogeneous in terms of hardware and software features. Classic examples of distributed computing systems are computing grids or Internet computing systems, which combine together the biggest variety of architectures, systems, and applications in the world.

What is parallel processing?

Processing of multiple tasks simultaneously on multiple processors is called *parallel processing*. The parallel program consists of multiple active processes (tasks) simultaneously solving a given problem. A given task is divided into multiple subtasks using a divide-and-conquer technique, and each subtask is processed on a different central processing unit (CPU). Programming on a multiprocessor system using the divide-and-conquer technique is called *parallel programming*.

Many applications today require more computing power than a traditional sequential computer can offer. Parallel processing provides a cost-effective solution to this problem by increasing the number of CPUs in a computer and by adding an efficient communication system between them. The workload can then be shared between different processors. This setup results in higher computing power and performance than a single-processor system offers.

The development of parallel processing is being influenced by many factors. The prominent among them include the following:

- Computational requirements are ever increasing in the areas of both scientific and business computing. The technical computing problems, which require high-speed computational power, are related to life sciences, aerospace, geographical information systems, mechanical design and analysis, and the like.
- Sequential architectures are reaching physical limitations as they are constrained by the speed of light and thermodynamics laws. The speed at which sequential CPUs can operate is reaching saturation point (no more vertical growth), and hence an alternative way to get high computational speed is to connect multiple CPUs (opportunity for horizontal growth).
- Hardware improvements in pipelining, superscalar, and the like are nonscalable and require sophisticated compiler technology. Developing such compiler technology is a difficult task.
- Vector processing works well for certain kinds of problems. It is suitable mostly for scientific problems (involving lots of matrix operations) and graphical processing. It is not useful for other areas, such as databases.
- The technology of parallel processing is mature and can be exploited commercially; there is already significant R&D work on development tools and environments.
- Significant development in networking technology is paving the way for heterogeneous computing.

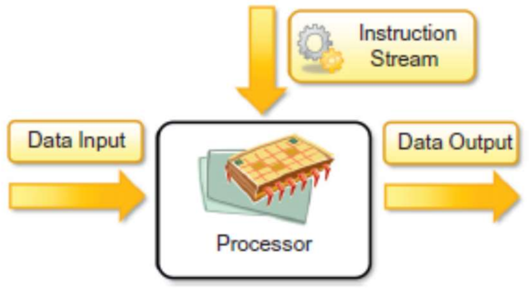
Hardware architectures for parallel processing

The core elements of parallel processing are CPUs. Based on the number of instruction and data streams that can be processed simultaneously, computing systems are classified into the following four categories:

- Single-instruction, single-data (SISD) systems
- Single-instruction, multiple-data (SIMD) systems
- Multiple-instruction, single-data (MISD) systems
- Multiple-instruction, multiple-data (MIMD) systems

- ***Single-instruction, single-data (SISD) systems***

An SISD computing system is a uniprocessor machine capable of executing a single instruction, which operates on a single data stream (see figure below). In SISD, machine instructions are processed sequentially; hence computers adopting this model are popularly called *sequential computers*. Most conventional computers are built using the SISD model. All the instructions and data to be processed have to be stored in primary memory. The speed of the processing element in the SISD model is limited by the rate at which the computer can transfer information internally. Dominant representative SISD systems are IBM PC, Macintosh, and workstations.

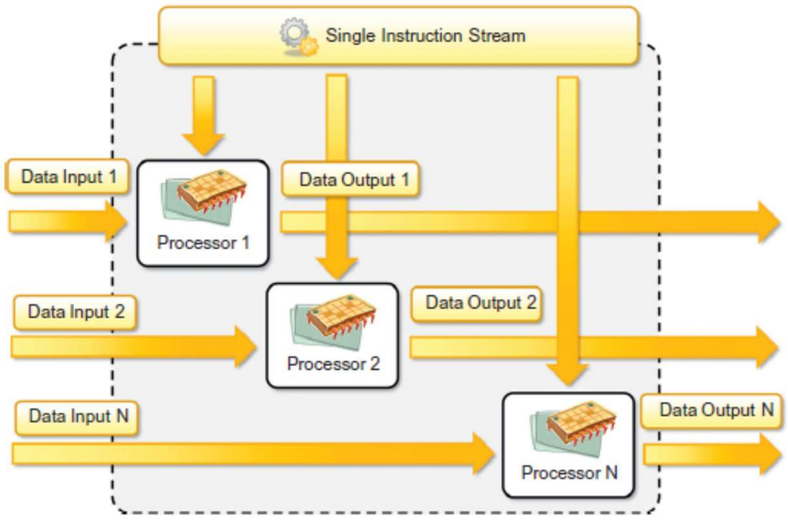


Single-instruction, multiple-data (SIMD) systems

An SIMD computing system is a multiprocessor machine capable of executing the same instruction on all the CPUs but operating on different data streams (see below figure). Machines based on an SIMD model are well suited to scientific computing since they involve lots of vector and matrix operations. For instance, statements such as

$$Ci = Ai * Bi$$

can be passed to all the processing elements (PEs); organized data elements of vectors A and B can be divided into multiple sets (N-sets for N PE systems); and each PE can process one data set. Dominant representative SIMD systems are Cray's vector processing machine and Thinking Machines cm*.

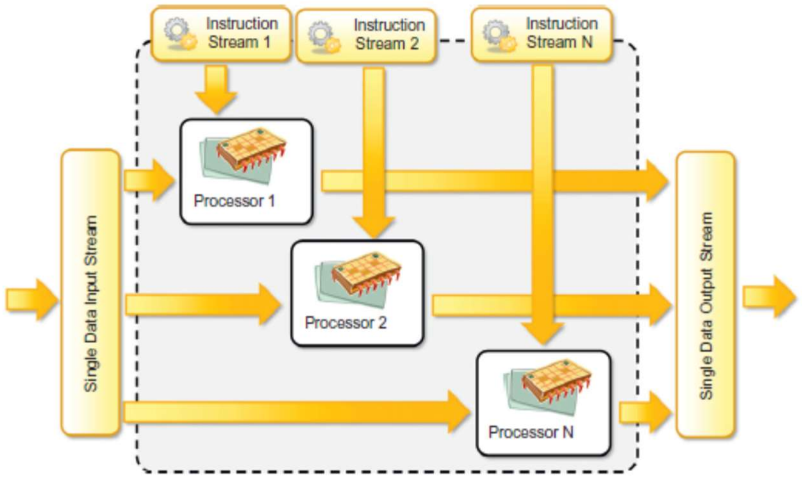


Multiple-instruction, single-data (MISD) systems

An MISD computing system is a multiprocessor machine capable of executing different instructions on different PEs but all of them operating on the same data set (see below figure). For instance, statements such as

$$y = \sin(x) + \cos(x) + \tan(x)$$

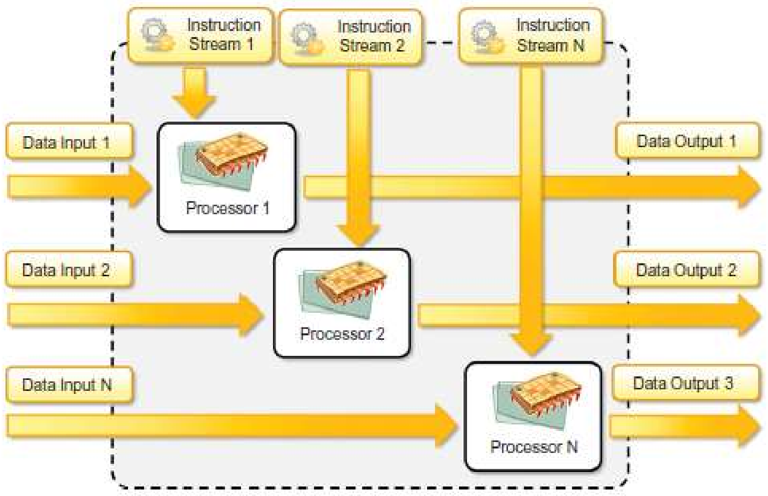
perform different operations on the same data set. Machines built using the MISD model are not useful in most of the applications; a few machines are built, but none of them are available commercially. They became more of an intellectual exercise than a practical configuration.



Multiple-instruction, multiple-data (MIMD) systems

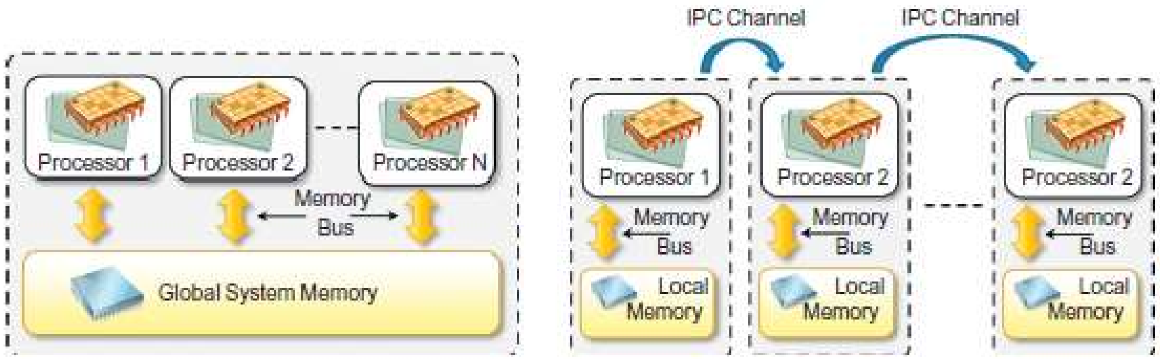
An MIMD computing system is a multiprocessor machine capable of executing multiple instructions on multiple data sets (see below figure). Each PE in the MIMD model has separate instruction and data streams; hence machines built using this model are well suited to any kind of application. Unlike SIMD and MISD machines, PEs in MIMD machines work asynchronously.

MIMD machines are broadly categorized into shared-memory MIMD and distributed-memory MIMD based on the way PEs are coupled to the main memory.



- **Shared memory MIMD machines**

  In the *shared memory MIMD model,* all the PEs are connected to a single global memory and they all have access to it (see Figure 2.6). Systems based on this model are also called *tightly coupled multiprocessor systems.* The communication between PEs in this model takes place through the shared memory; modification of the data stored in the global memory by one PE is visible to all other PEs. Dominant representative shared memory MIMD systems are Silicon Graphics machines and Sun/IBM's SMP (Symmetric Multi-Processing).



- **Distributed memory MIMD machines**

  In the *distributed memory MIMD model,* all PEs have a local memory. Systems based on this model are also called *loosely coupled multiprocessor systems.* The communication between PEs in this model takes place through the interconnection network (the interprocess communication channel, or IPC). The network connecting PEs can be configured to tree, mesh, cube, and so on. Each PE operates asynchronously, and if communication/synchronization among tasks is necessary, they can do so by exchanging messages between them.

Approaches to parallel programming

A sequential program is one that runs on a single processor and has a single line of control. To make many processors collectively work on a single program, the program must be divided into smaller independent chunks so that each processor can work on separate chunks of the problem. The program decomposed in this way is a parallel program.

A wide variety of parallel programming approaches are available. The most prominent among them are the following:

- Data parallelism
- Process parallelism
- Farmer-and-worker model

These three models are all suitable for task-level parallelism. In the case of data parallelism, the divide-and-conquer technique is used to split data into multiple sets, and each data set is processed on different PEs using the same instruction. This approach is highly suitable to processing on machines based on the SIMD model. In the case of process parallelism, a given operation has multiple (but distinct) activities that can be processed on multiple processors. In the case of the farmer- and-worker model, a job distribution approach is used: one processor is configured as master and all other remaining PEs are designated as slaves; the master assigns jobs to slave PEs and, on completion, they inform the master, which in turn collects results. These approaches can be utilized in different levels of parallelism.

Levels of parallelism

Levels of parallelism are decided based on the lumps of code (grain size) that can be a potential candidate for parallelism. Table lists categories of code granularity for parallelism. All these approaches have a common goal: to boost processor efficiency by hiding latency. To conceal latency, there must be another thread ready to run whenever a lengthy operation occurs. The idea is to execute concurrently two or more single-threaded applications, such as compiling, text formatting, database searching, and device simulation.

As shown in the table and depicted in Figure , parallelism within an application can be detected at several levels:

- Large grain (or task level)
- Medium grain (or control level)
- Fine grain (data level)
- Very fine grain (multiple-instruction issue)

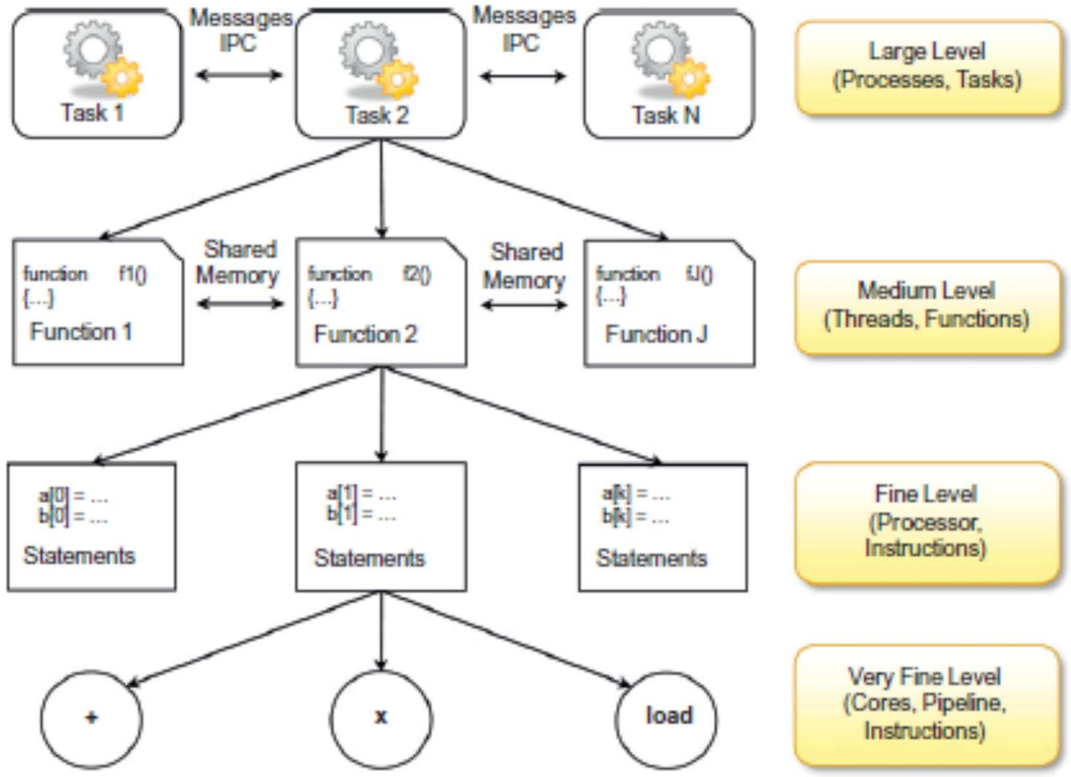


Table 2.1 Levels of Parallelism

Grain Size	Code Item	Parallelized By
Large	Separate and heavyweight process	Programmer
Medium	Function or procedure	Programmer
Fine	Loop or instruction block	Parallelizing compiler
Very fine	Instruction	Processor

Laws of caution

Now that we have introduced some general aspects of parallel computing in terms of architectures and models, we can make some considerations that have been drawn from experience designing and implementing such systems. These considerations are guidelines that can help us understand how much benefit an application or a software system can gain from parallelism. In particular, what we need to keep in mind is that parallelism is used to perform multiple activities together so that the system can increase its throughput or its speed. But the relations that control the increment of speed are not linear. For example, for a given n processors, the user expects speed to be increased by n times. This is an ideal situation, but it rarely happens because of the communication overhead.

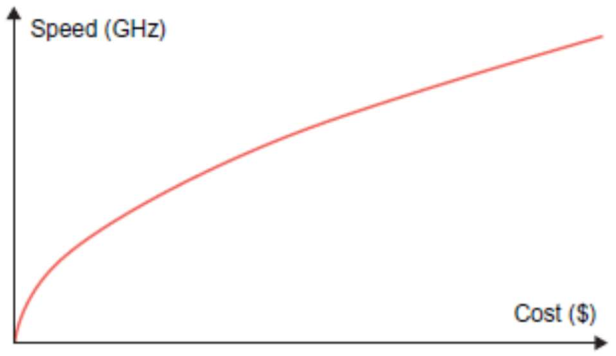


FIGURE 2.8

Cost versus speed.

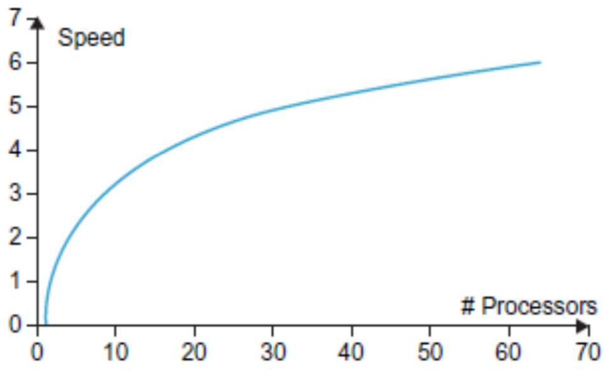


FIGURE 2.9

Number processors versus speed.

Here are two important guidelines to take into account:

- Speed of computation is proportional to the square root of system cost; they never increase linearly. Therefore, the faster a system becomes, the more expensive it is to increase its speed.
- Speed by a parallel computer increases as the logarithm of the number of processors (i.e., $y = k*log(N)$). This concept is shown in figure.

The very fast development in parallel processing and related areas has blurred conceptual boundaries, causing a lot of terminological confusion. Even well-defined distinctions such as shared memory and distributed memory are merging due to new advances in technology. There are no strict delimiters for contributors to the area of parallel processing. Hence, computer architects, OS designers, language designers, and computer network designers all have a role to play.

General concepts and definitions of a distributed system

Distributed computing studies the models, architectures, and algorithms used for building and managing distributed systems. As a general definition of the term *distributed system,* we use the one proposed by Tanenbaum et. al [1]:

A distributed system is a collection of independent computers that appears to its users as a single coherent system.

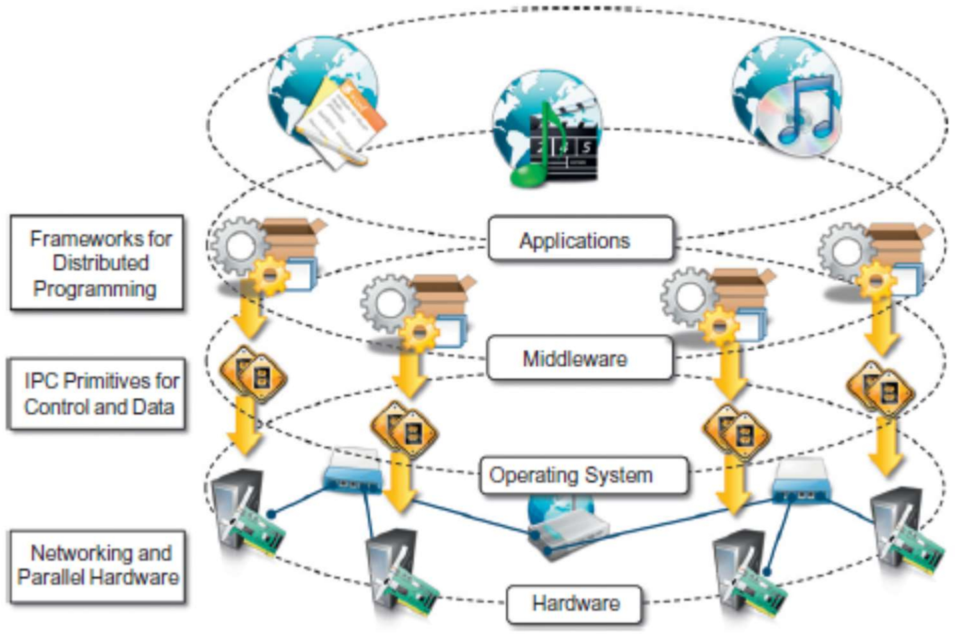
This definition is general enough to include various types of distributed computing systems that are especially focused on unified usage and aggregation of distributed resources. In this chapter, we focus on the architectural models that are used to harness independent computers and present them as a whole coherent system. Communication is another fundamental aspect of distributed computing. Since distributed systems are composed of more than one computer that collaborate together, it is necessary to provide some sort of data and information exchange between them, which generally occurs through the network:

A distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages.

As specified in this definition, the components of a distributed system communicate with some sort of *message passing*. This is a term that encompasses several communication models.

Components of a distributed system

A distributed system is the result of the interaction of several components that traverse the entire computing stack from hardware to software. It emerges from the collaboration of several elements that—by working together—give users the illusion of a single coherent system. Figure provides an overview of the different layers that are involved in providing the services of a distributed system.


Frameworks for Distributed Programming
IPC Primitives for Control and Data
Networking and Parallel Hardware
Applications
Middleware
Operating System
Hardware

At the very bottom layer, computer and network hardware constitute the physical infrastructure; these components are directly managed by the operating system, which provides the basic services for interprocess communication (IPC), process scheduling and management, and resource management in terms of file system and local devices. Taken together these two layers become the platform on top of which specialized software is deployed to turn a set of networked computers into a distributed system.

The use of well-known standards at the operating system level and even more at the hardware and network levels allows easy harnessing of heterogeneous components and their organization into a coherent and uniform system. For example, network connectivity between different devices is controlled by standards, which allow them to interact seamlessly. At the operating system level, IPC services are implemented on top of standardized communication protocols such Transmission Control Protocol/Internet Protocol (TCP/IP), User Datagram Protocol (UDP) or others.

The middleware layer leverages such services to build a uniform environment for the development and deployment of distributed applications. This layer supports the programming paradigms for distributed systems. By relying on the services offered by the operating system, the middleware develops its own protocols, data formats, and programming language or frameworks for the development of distributed applications. All of them constitute a uniform interface to distributed application developers that is completely independent from the underlying operating system and hides all the heterogeneities of the bottom layers.

The top of the distributed system stack is represented by the applications and services designed and developed to use the middleware. These can serve several purposes and often expose their features in the form of graphical user interfaces (GUIs) accessible locally or through the Internet via a Web browser. For example, in the case of a cloud computing system, the use of Web technologies is strongly preferred, not only to interface distributed applications with the end user but also to provide platform services aimed at building distributed systems. A very good example is constituted by Infrastructure-as-a-Service (IaaS) providers such as Amazon Web Services (AWS), which provide facilities for creating virtual machines, organizing them together into a cluster, and deploying applications and systems on top.

Note that hardware and operating system layers make up the bare-bone infrastructure of one or more datacenters, where racks of servers are deployed and connected together through high-speed connectivity. This infrastructure is managed by the operating system, which provides the basic capability of machine and network management. The core logic is then implemented in the middleware that manages the virtualization layer, which is deployed on the physical infrastructure in order to maximize its utilization and provide a customizable runtime environment for applications. The middleware provides different facilities to application developers according to the type of services sold to customers. These facilities, offered through Web 2.0-compliant interfaces, range from virtual infrastructure building and deployment to application development and runtime environments.

Architectural styles for distributed computing

Although a distributed system comprises the interaction of several layers, the middleware layer is the one that enables distributed computing, because it provides a coherent and uniform runtime environment for applications. There are many different ways to organize the components that, taken together, constitute such an environment. The interactions among these components and their responsibilities give structure to the middleware and characterize its type or, in other words, define its architecture. Architectural styles aid in understanding and classifying the organization of software systems in general and distributed computing in particular.

Architectural styles are mainly used to determine the vocabulary of components and connectors that are used as instances of the style together with a set of constraints on how they can be combined .

Design patterns help in creating a common knowledge within the community of software engineers and developers as to how to structure the relations of components within an application and understand the internal organization of software applications. Architectural styles do the same for the overall architecture of software systems. In this section, we introduce the most relevant architectural styles for distributed computing and focus on the components and connectors that make each style peculiar. Architectural styles for distributed systems are helpful in understanding the different roles of components in the system and how they are distributed across multiple machines. We organize the architectural styles into two major classes:

- Software architectural styles
- System architectural styles

The first class relates to the logical organization of the software; the second class includes all those styles that describe the physical organization of distributed software systems in terms of their major components.

Component and connectors

Before we discuss the architectural styles in detail, it is important to build an appropriate vocabulary on the subject. Therefore, we clarify what we intend for *components* and *connectors,* since these are the basic building blocks with which architectural styles are defined. A *component* represents a unit of software that encapsulates a function or a feature of the system. Examples of components can be programs, objects, processes, pipes, and filters. A *connector* is a communication mechanism that allows cooperation and coordination among components. Differently from components, connectors are not encapsulated in a single entity, but they are implemented in a distributed manner over many system components.

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. According to Garlan and Shaw, architectural styles are classified as shown in Table 2.2.

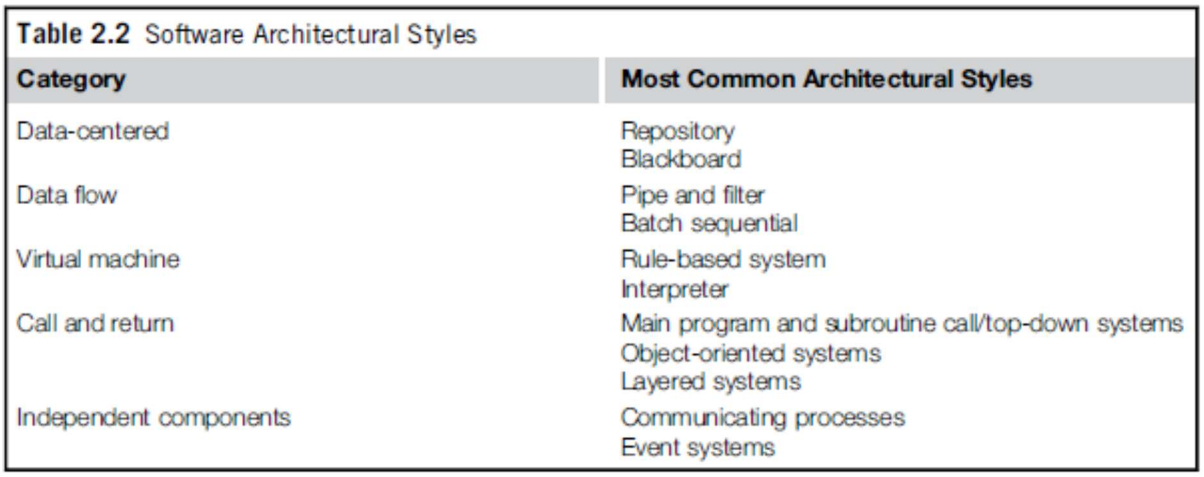


Table 2.2 Software Architectural Styles

Category	Most Common Architectural Styles
Data-centered	Repository Blackboard
Data flow	Pipe and filter Batch sequential
Virtual machine	Rule-based system Interpreter
Call and return	Main program and subroutine call/top-down systems Object-oriented systems Layered systems
Independent components	Communicating processes Event systems

These models constitute the foundations on top of which distributed systems are designed from a logical point of view.

Data centered architectures

These architectures identify the data as the fundamental element of the software system, and access to shared data is the core characteristic of the data-centered architectures. Therefore, especially within the context of distributed and parallel computing systems, integrity of data is the overall goal for such systems.

The *repository* architectural style is the most relevant reference model in this category. It is characterized by two main components: the central data structure, which represents the current state of the system, and a collection of independent components, which operate on the central data. The ways in which the independent components interact with the central data structure can be very heterogeneous. In particular, repository-based architectures differentiate and specialize further into subcategories according to the choice of control discipline to apply for the shared data structure. Of particular interest are *databases* and *blackboard systems.* In the former group the dynamic of the system is controlled by the independent components, which, by issuing an operation on the central repository, trigger the selection of specific processes that operate on data. In blackboard systems, the central data structure is the main trigger for selecting the processes to execute.

The *blackboard* architectural style is characterized by three main components:

- *Knowledge sources.* These are the entities that update the knowledge base that is maintained in the blackboard.
- *Blackboard.* This represents the data structure that is shared among the knowledge sources and stores the knowledge base of the application.
- *Control.* The control is the collection of triggers and procedures that govern the interaction with the blackboard and update the status of the knowledge base.

Data-flow architectures

In the case of *data-flow* architectures, it is the availability of data that controls the computation. With respect to the data-centered styles, in which the access to data is the core feature, data-flow styles explicitly incorporate the pattern of *data flow,* since their design is determined by an orderly motion of data from component to component, which is the form of communication between them. Styles within this category differ in one of the following ways: how the control is exerted, the degree of concurrency among components, and the topology that describes the flow of data.

- ***Batch Sequential Style.*** The batch sequential style is characterized by an ordered sequence of separate programs executing one after the other. These programs are chained together by providing as input for the next program the output generated by the last program after its completion, which is most likely in the form of a file. This design was very popular in the mainframe era of computing and still finds applications today. For example, many distributed applications for scientific computing are defined by jobs expressed as sequences of programs that, for example, pre-filter, analyze, and post-process data. It is very common to compose these phases using the batch- sequential style.

- ***Pipe-and-Filter Style.*** The *pipe-and-filter style* is a variation of the previous style for expressing the activity of a software system as sequence of data transformations. Each component of the processing chain is called a *filter,* and the connection between one filter and the next is represented by a data stream. With respect to the batch sequential style, data is processed incrementally and each filter processes the data as soon as it is available on the input stream. As soon as one filter produces a consumable amount of data, the next filter can start its processing. Filters generally do not have state, know the identity of neither the previous nor the next filter, and they are connected with in-memory data structures such as first-in/first-out (FIFO) buffers or other structures. This particular sequencing is called *pipelining* and introduces concurrency in the execution of the filters. A classic example of this architecture is the microprocessor pipeline, whereby multiple instructions are executed at the same time by completing a different phase of each of them. We can identify the phases of the instructions as the filters, whereas the data streams are represented by the registries that are shared within the processors.

Data-flow architectures are optimal when the system to be designed embodies a multistage process, which can be clearly identified into a collection of separate components that need to be orchestrated together. Within this reference scenario, components have well-defined interfaces exposing input and output ports, and the connectors are represented by the datastreams between these ports. The main differences between the two subcategories are reported in Table .

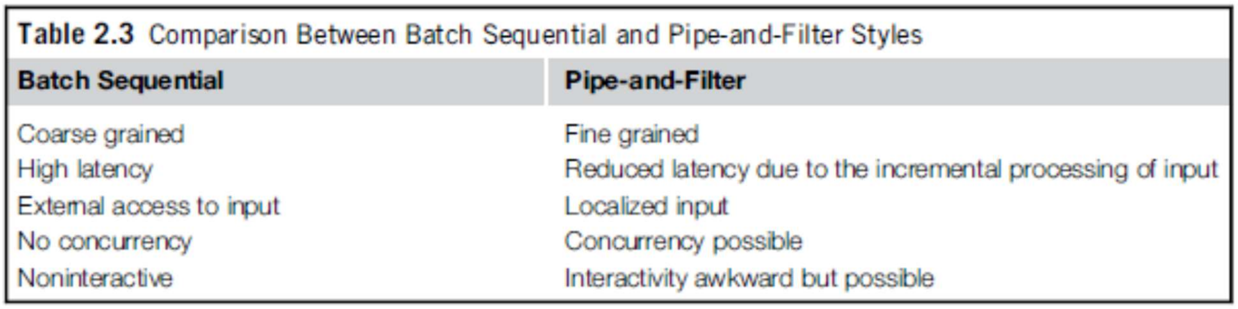


Table 2.3 Comparison Between Batch Sequential and Pipe-and-Filter Styles

Batch Sequential	Pipe-and-Filter
Coarse grained	Fine grained
High latency	Reduced latency due to the incremental processing of input
External access to input	Localized input
No concurrency	Concurrency possible
Noninteractive	Interactivity awkward but possible

Virtual machine architectures

The virtual machine class of architectural styles is characterized by the presence of an abstract execution environment (generally referred as a *virtual machine)* that simulates features that are not available in the hardware or software. Applications and systems are implemented on top of this layer and become portable over different hardware and software environments as long as there is an implementation of the virtual machine they interface with. The general interaction flow for systems implementing this pattern is the following: the program (or the application) defines its operations and state in an abstract format, which is interpreted by the virtual machine engine. The interpretation of a program constitutes its execution. It is quite common in this scenario that the engine maintains an internal representation of the program state. Very popular examples within this category are rule-based systems, interpreters, and command-language processors.

- ***Rule-Based Style.*** This architecture is characterized by representing the abstract execution environment as an *inference engine.* Programs are expressed in the form of rules or predicates that hold true. The input data for applications is generally represented by a set of assertions or facts that the inference engine uses to activate rules or to apply predicates, thus transforming data. The output can either be the product of the rule activation or a set of assertions that holds true for the given input data. The set of rules or predicates identifies the knowledge base that can be queried to infer properties about the system. This approach is quite peculiar, since it allows expressing a system or a domain in terms of its behavior rather than in terms of the components. Rule-based systems are very popular in the field of artificial intelligence. Practical applications can be found in the field of process control, where rule-based systems are used to monitor the status of physical devices by being fed from the sensory data collected and processed by PLCs[1] and by activating alarms when specific conditions on the sensory data apply. Another interesting use of rule-based systems can be found in the networking domain: *network intrusion detection systems (NIDS)* often rely on a set of rules to identify abnormal behaviors connected to possible intrusions in computing systems.

- ***Interpreter Style.*** The core feature of the interpreter style is the presence of an engine that is used to interpret a pseudo-program expressed in a format acceptable for the interpreter. The interpretation of the pseudo-program constitutes the execution of the program itself. Systems modeled according to this style exhibit four main components: the interpretation engine that executes the core activity of this style, an internal memory that contains the pseudo-code to be interpreted, a representation of the current state of the engine, and a representation of the current state of the program being executed. This model is quite useful in designing virtual machines for high-level programming (Java, C#) and scripting languages (Awk, PERL, and so on). Within this scenario, the virtual machine closes the gap between the end-user abstractions and the software/hardware environment in which such abstractions are executed.

Virtual machine architectural styles are characterized by an indirection layer between application and the hosting environment. This design has the major advantage of decoupling applications from the underlying hardware and software environment, but at the same time it introduces some disadvantages, such as a slowdown in performance. Other issues might be related to the fact that, by providing a virtual execution environment, specific features of the underlying system might not be accessible.

Call & return architectures

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. Nonetheless, it is possible to identify three major subcategories, which differentiate by the way the system is structured and how methods are invoked: top-down style, object-oriented style, and layered style.

- ***Top-Down Style.*** This architectural style is quite representative of systems developed with imperative programming, which leads to a divide-and-conquer approach to problem resolution. Systems developed according to this style are composed of one large main program that accomplishes its tasks by invoking subprograms or procedures. The components in this style are procedures and subprograms, and connections are method calls or invocation. The calling program passes information with parameters and receives data from return values or parameters. Method calls can also extend beyond the boundary of a single process by leveraging techniques for remote method invocation, such as remote procedure call (RPC) and all its descendants. The overall structure of the program execution at any point in time is characterized by a tree, the root of which constitutes the main function of the principal program. This architectural style is quite intuitive from a design point of view but hard to maintain and manage in large systems.

- ***Object-Oriented Style.*** This architectural style encompasses a wide range of systems that have been designed and implemented by leveraging the abstractions of object-oriented programming (OOP). Systems are specified in terms of classes and implemented in terms of objects. Classes define the type of components by specifying the data that represent their state and the operations that can be done over these data. One of the main advantages over the top-down style is that there is a coupling between data and operations used to manipulate them. Object instances become responsible for hiding their internal state representation and for protecting its integrity while providing operations to other components. This leads to a better decomposition process and more manageable systems. Disadvantages of this style are mainly two: each object needs to know the identity of an object if it wants to invoke operations on it, and shared objects need to be carefully designed in order to ensure the consistency of their state.
- ***Layered Style.*** The layered system style allows the design and implementation of software systems in terms of layers, which provide a different level of abstraction of the system. Each layer generally operates with at most two layers: the one that provides a lower abstraction level and the one that provides a higher abstraction layer. Specific protocols and interfaces define how adjacent layers interact. It is possible to model such systems as a stack of layers, one for each level of abstraction. Therefore, the components are the layers and the connectors are the interfaces and protocols used between adjacent layers. A user or client generally interacts with the layer at the highest abstraction, which, in order to carry its activity, interacts and uses the services of the lower layer. This process is repeated (if necessary) until the lowest layer is reached. It is also possible to have the opposite behavior: events and callbacks from the lower layers can trigger the activity of the higher layer and propagate information up through the stack. The advantages of the layered style are that, as happens for the object-oriented style, it supports a modular design of systems and allows us to decompose the system according to different levels of abstractions by encapsulating together all the operations that belong to a specific level. Examples of layered architectures are the modern operating system kernels and the International Standards Organization/Open Systems Interconnection (ISO/OSI) or the TCP/IP stack.

Architectural styles based on independent components

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. There are two major categories within this class—communicating processes and event systems—which differentiate in the way the interaction among components is managed.

- ***Communicating Processes.*** In this architectural style, components are represented by independent processes that leverage IPC facilities for coordination management. This is an abstraction that is quite suitable to modeling distributed systems that, being distributed over a network of computing nodes, are necessarily composed of several concurrent processes. Each of the processes provides other processes with services and can leverage the services exposed by the other processes. The conceptual organization of these processes and the way in which the communication happens vary according to the specific model used, either peer-to-peer or client/server. Connectors are identified by IPC facilities used by these processes to communicate.

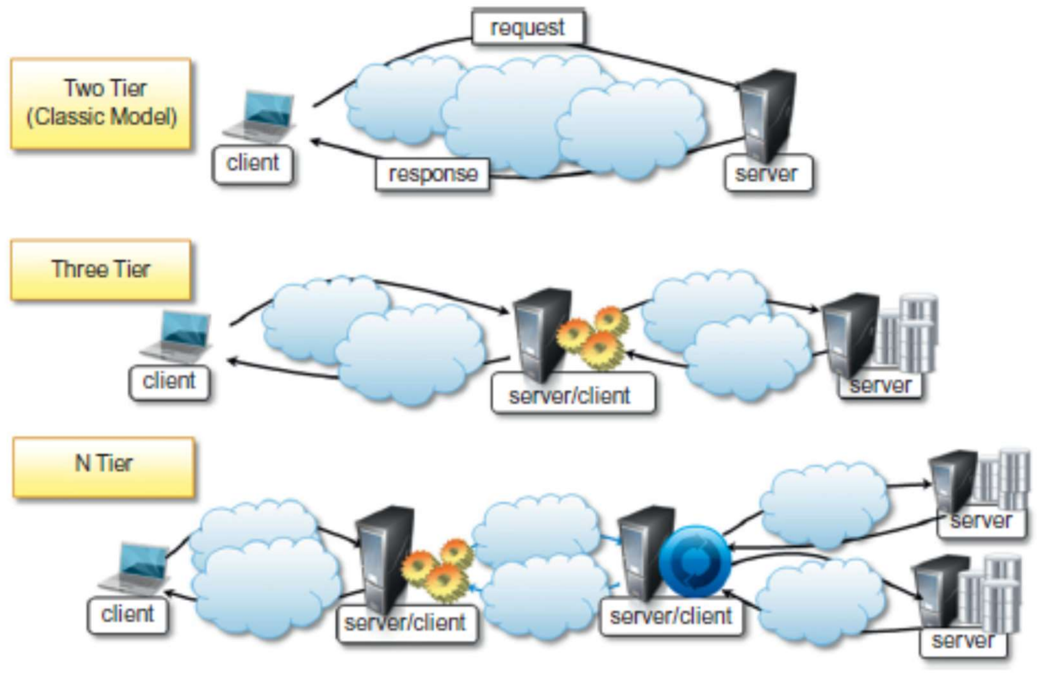
- ***Event Systems.*** In this architectural style, the components of the system are loosely coupled and connected. In addition to exposing operations for data and state manipulation, each component also publishes (or announces) a collection of events with which other components can register. In general, other components provide a callback that will be executed when the event is activated. During the activity of a component, a specific runtime condition can activate one of the exposed events, thus triggering the execution of the callbacks registered with it. Event activation may be accompanied by contextual information that can be used in the callback to handle the event. This information can be passed as an argument to the callback or by using some shared repository between components. Event-based systems have become quite popular, and support for their implementation is provided either at the API level or the programming language level. The main
  advantage of such an architectural style is that it fosters the development of open systems: new modules can be added and easily integrated into the system as long as they have compliant interfaces for registering to the events. This architectural style solves some of the limitations observed for the top-down and object-oriented styles. First, the invocation pattern is implicit, and the connection between the caller and the callee is not hard-coded; this gives a lot of flexibility since addition or removal of a handler to events can be done without changes in the source code of applications. Second, the event source does not need to know the identity of the event handler in order to invoke the callback. The disadvantage of such a style is that it relinquishes control over system computation. When a component triggers an event, it does not know how many event handlers will be invoked and whether there are any registered handlers. This information is available only at runtime and, from a static design point of view, becomes more complex to identify the connections among components and to reason about the correctness of the interactions.

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. In this section, we introduce two fundamental reference styles: *client/server* and *peer-to-peer.*

Client/server

This architecture is very popular in distributed computing and is suitable for a wide variety of applications. As depicted in below figure, the client/server model features two major components: a *server* and a *client*. These two components interact with each other through a network connection using a given protocol. The communication is unidirectional: The client issues a request to the server, and after processing the request the server returns a response. There could be multiple client components issuing requests to a server that is passively waiting for them. Hence, the important operations in the client-server paradigm are *request, accept* (client side), and *listen* and *response* (server side).



The client/server model is suitable in many-to-one scenarios, where the information and the services of interest can be centralized and accessed through a single access point: the server. In general, multiple clients are interested in such services and the server must be appropriately designed to efficiently serve requests coming from different clients. This consideration has implications on both client design and server design. For the client design, we identify two major models:

- ***Thin-client model.*** In this model, the load of data processing and transformation is put on the server side, and the client has a light implementation that is mostly concerned with retrieving and returning the data it is being asked for, with no considerable further processing.
- ***Fat-client model.*** In this model, the client component is also responsible for processing and transforming the data before returning it to the user, whereas the server features a relatively light implementation that is mostly concerned with the management of access to the data.

The three major components in the client-server model: presentation, application logic, and data storage. In the thin-client model, the client embodies only the presentation component, while the server absorbs the other two. In the fat-client model, the client encapsulates presentation and most of the application logic, and the server is principally responsible for the data storage and maintenance.

Presentation, application logic, and data maintenance can be seen as conceptual layers, which are more appropriately called *tiers.* The mapping between the conceptual layers and their physical implementation in modules and components allows differentiating among several types of architectures, which go under the name of *multitiered architectures.* Two major classes exist:

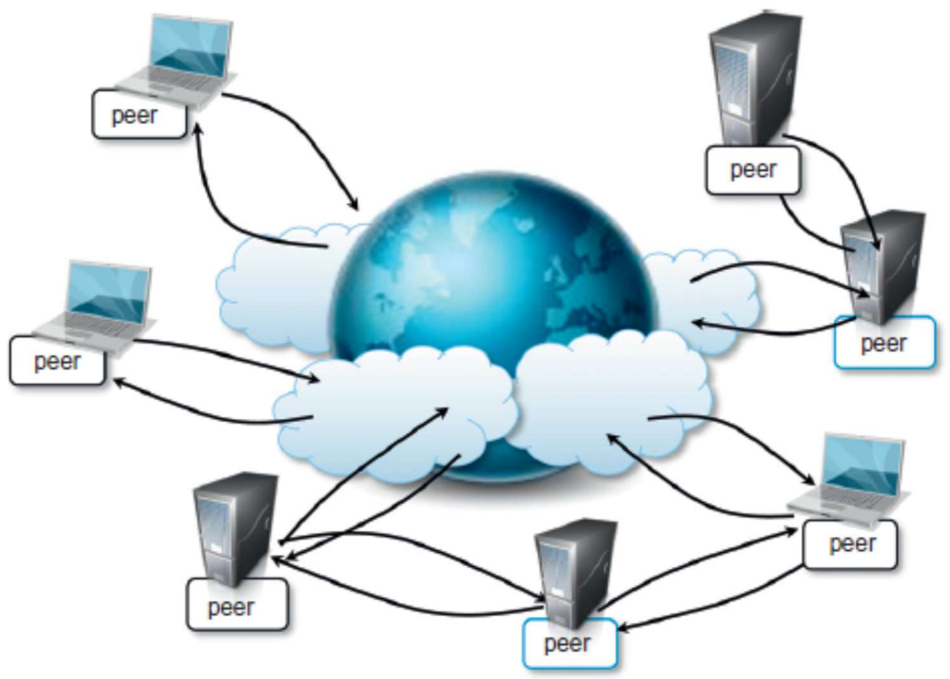
- ***Two-tier architecture***. This architecture partitions the systems into two tiers, which are located one in the client component and the other on the server. The client is responsible for the presentation tier by providing a user interface; the server concentrates the application logic and the data store into a single tier. The server component is generally deployed on a powerful machine that is capable of processing user requests, accessing data, and executing the application logic to provide a client with a response. This architecture is suitable for systems of limited size and suffers from scalability issues. In particular, as the number of users increases the performance of the server might dramatically decrease. Another limitation is caused by the dimension of the data to maintain, manage, and access, which might be prohibitive for a single computation node or too large for serving the clients with satisfactory performance.

- ***Three-tier architecture/N-tier architecture.*** The three-tier architecture separates the presentation of data, the application logic, and the data storage into three tiers. This architecture is generalized into an N-tier model in case it is necessary to further divide the stages composing the application logic and storage tiers. This model is generally more scalable than the two-tier one because it is possible to distribute the tiers into several computing nodes, thus isolating the performance bottlenecks. At the same time, these systems are also more complex to understand and manage. A classic example of three-tier architecture is constituted by a medium-size Web application that relies on a relational database management system for storing its data. In this scenario, the client component is represented by a Web browser that embodies the presentation tier, whereas the application server encapsulates the business logic tier, and a database server machine (possibly replicated for high availability) maintains the data storage. Application servers that rely on third-party (or external) services to satisfy client requests are examples of N-tiered architectures.

The client/server architecture has been the dominant reference model for designing and deploying distributed systems, and several applications to this model can be found. The most relevant is perhaps the Web in its original conception. Nowadays, the client/server model is an important building block of more complex systems, which implement some of their features by identifying a server and a client process interacting through the network. This model is generally suitable in the case of a many-to-one scenario, where the interaction is unidirectional and started by the clients and suffers from scalability issues, and therefore it is not appropriate in very large systems.

Peer-to-peer

The peer-to-peer model, 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.



The most relevant example of peer-to-peer systems [87] is constituted by file-sharing applications such as *Gnutella, BitTorrent,* and *Kazaa.* Despite the differences among these networks in coordinating nodes and sharing information on the files and their locations, all of them provide a user client that is at the same time a server providing files to other peers and a client downloading files from other peers. To address an incredibly large number of peers, different architectures have been designed that divert slightly from the peer-to-peer model. For example, in *Kazaa* not all the peers have the same role, and some of them are used to group the accessibility information of a group of peers. Another interesting example of peer-to-peer architecture is represented by the Skype network.

The system architectural styles presented in this section constitute a reference model that is further enhanced or diversified according to the specific needs of the application to be designed and implemented. For example, the client/server architecture, which originally included only two types of components, has been further extended and enriched by developing multitier architectures as the complexity of systems increased. Currently, this model is still the predominant reference architecture for distributed systems and applications. The *server* and *client* abstraction can be used in some cases to model the macro scale or the micro scale of the systems. For peer-to-peer systems, pure implementations are very hard to find and, as discussed for the case of *Kazaa,* evolutions of the model, which introduced some kind of hierarchy among the nodes, are common.

Models for interprocess communication

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. Among the most relevant that we can mention are shared memory, remote procedure call (RPC), and message passing. 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. Sockets provide the basic capability of transferring a sequence of bytes, which is converted at higher levels into a more meaningful representation (such as procedure parameters or return values or messages). Such a powerful abstraction allows system engineers to concentrate on the logic-coordinating distributed components and the information they exchange rather than the networking details. These two elements identify the model for IPC. In this section, we introduce the most important reference model for architecting the communication among processes.

Message-based communication

The abstraction of *message* has played an important role in the evolution of the models and technologies enabling distributed computing. Couloris et al. define a distributed system as "one in which components located at networked computers communicate and coordinate their actions only by passing messages." The term *message,* in this case, identifies any discrete amount of information that is passed from one entity to another. It encompasses any form of data representation that is limited in size and time, whereas this is an invocation to a remote procedure or a serialized object instance or a generic message. Therefore, the term *message-based communication model* can be used to refer to any model for IPC discussed in this section, which does not necessarily rely on the abstraction of data streaming.

Several distributed programming paradigms eventually use message-based communication despite the abstractions that are presented to developers for programming the interaction of distributed components. Here are some of the most popular and important:

- *Message passing.* This paradigm introduces the concept of a message as the main abstraction of the model. The entities exchanging information explicitly encode in the form of a message the data to be exchanged. The structure and the content of a message vary according to the model. Examples of this model are the *Message-Passing Interface (MPI)* and *OpenMP.*
- *Remote procedure call (RPC).* 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. A remote process hosts a server component, thus allowing client processes to request the invocation of methods, and returns the result of the execution. Messages, automatically created by the RPC implementation, convey the information about the procedure to execute along with the required parameters and the return values. The use of messages within this context is also referred as *marshaling* of parameters and return values.
- *Distributed objects.* This is an implementation of the RPC model for the object-oriented paradigm and contextualizes this feature for the remote invocation of methods exposed by objects. Each process registers a set of interfaces that are accessible remotely. Client processes can request a pointer to these interfaces and invoke the methods available through them. The underlying runtime infrastructure is in charge of transforming the local method invocation into a request to a remote process and collecting the result of the execution. The communication between the caller and the remote process is made through messages. With respect to the RPC model that is stateless by design, distributed object models introduce the complexity of object state management and lifetime. The methods that are remotely

executed operate within the context of an instance, which may be created for the sole execution of the method, exist for a limited interval of time, or are independent from the existence of requests. Examples of distributed object infrastructures are *Common Object Request Broker Architecture (CORBA), Component Object Model (COM, DCOM, and COM+), Java Remote Method Invocation (RMI), and .NET Remoting.*

- *Distributed agents and active objects.* Programming paradigms based on agents and active objects involve by definition the presence of instances, whether they are agents of objects, despite the existence of requests. This means that objects have their own control thread, which allows them to carry out their activity. These models often make explicit use of messages to trigger the execution of methods, and a more complex semantics is attached to the messages.
- *Web services.* 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)*.

Models for message-based communication

An important aspect characterizing the interaction among distributed components is the way these messages are exchanged and among how many components. In several cases, we identified the client/server model as the underlying reference model for the interaction. This, in its strictest form, represents a point-to-point communication model allowing a many-to-one interaction pattern. Variations of the client/server model allow for different interaction patterns. In this section, we briefly discuss the most important and recurring ones.

Point-to-point message model

This model organizes the communication among single components. Each message is sent from one component to another, and there is a direct addressing to identify the message receiver. In a point-to-point communication model it is necessary to know the location of or how to address another component in the system. There is no central infrastructure that dispatches the messages, and the communication is initiated by the message sender. It is possible to identify two major subcategories: direct communication and queue-based communication. In the former, the message is sent directly to the receiver and processed at the time of reception. In the latter, the receiver maintains a message queue in which the messages received are placed for later processing. The point-to- point message model is useful for implementing systems that are mostly based on one-to-one or many-to-one communication.

Publish-and-subscribe message model

This model introduces a different strategy, one that is based on notification among components. There are two major roles: the *publisher* and the *subscriber*. The former provides facilities for the latter to register its interest in a specific topic or event. Specific conditions holding true on the publisher side can trigger the creation of messages that are attached to a specific event. A message will be available to all the subscribers that registered for the corresponding event. There are two major strategies for dispatching the event to the subscribers:

- *Push strategy.* In this case it is the responsibility of the publisher to notify all the subscribers— for example, with a method invocation.
- *Pull strategy.* In this case the publisher simply makes available the message for a specific event, and it is responsibility of the subscribers to check whether there are messages on the events that are registered.

The publish-and-subscribe model is very suitable for implementing systems based on the one- to-many communication model and simplifies the implementation of indirect communication patterns. It is, in fact, not necessary for the publisher to know the identity of the subscribers to make the communication happen.

Request-reply message model

The request-reply message model identifies all communication models in which, for each message sent by a process, there is a reply. This model is quite popular and provides a different classification that does not focus on the number of the components involved in the communication but rather on how the dynamic of the interaction evolves. Point-to-point message models are more likely to be based on a request-reply interaction, especially in the case of direct communication. Publish- and-subscribe models are less likely to be based on request-reply since they rely on notifications.

Technologies for distributed computing

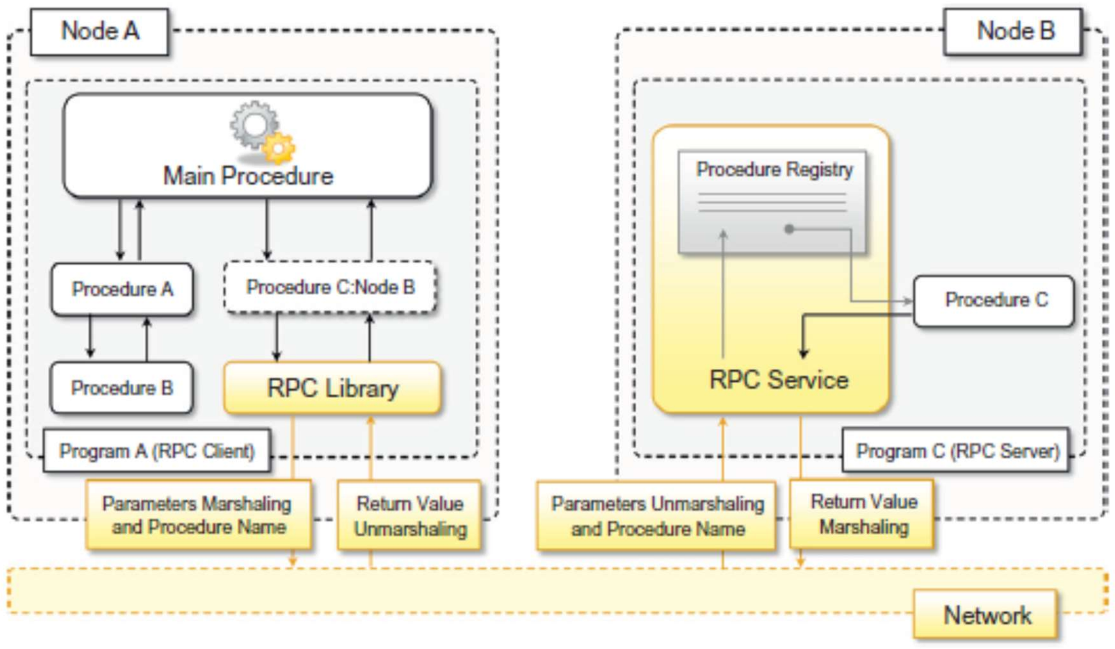
Remote procedure call (RPC), distributed object frameworks, and service-oriented computing provide concrete implementations of interaction models, which mostly rely on message-based communication.

Remote procedure call

RPC is the fundamental abstraction enabling the execution of procedures on client's request. RPC allows extending the concept of a procedure call beyond the boundaries of a process and a single memory address space. The called procedure and calling procedure may be on the same system or they may be on different systems in a network.

Below figure illustrates the major components that enable an RPC system. The system is based on a client/server model. The server process maintains a registry of all the available procedures that can be remotely invoked and listens for requests from clients that specify which procedure to invoke, together with the values of the parameters required by the procedure. RPC maintains the synchronous pattern that is natural in IPC and function calls. Therefore, the calling process thread remains blocked until the procedure on the server process has completed its execution and the result (if any) is returned to the client.

An important aspect of RPC is *marshaling,* which identifies the process of converting parameter and return values into a form that is more suitable to be transported over a network through a sequence of bytes. The term *unmarshaling* refers to the opposite procedure. Marshaling and unmarshaling are performed by the RPC runtime infrastructure, and the client and server user code does not necessarily have to perform these tasks. The RPC runtime, on the other hand, is not only responsible for parameter packing and unpacking but also for handling the request-reply interaction that happens between the client and the server process in a completely transparent manner.



Therefore, developing a system leveraging RPC for IPC consists of the following steps:

- Design and implementation of the server procedures that will be exposed for remote invocation.
- Registration of remote procedures with the RPC server on the node where they will be made available.
- Design and implementation of the client code that invokes the remote procedure(s).

Each RPC implementation generally provides client and server application programming interfaces (APIs) that facilitate the use of this simple and powerful abstraction. An important observation has to be made concerning the passing of parameters and return values. Since the server and the client processes are in two separate address spaces, the use of parameters passed by references or pointers is not suitable in this scenario, because once unmarshaled these will refer to a memory location that is not accessible from within the server process. Second, in user-defined parameters and return value types, it is necessary to ensure that the RPC runtime is able to marshal them. This is generally possible, especially when user-defined types are composed of simple types, for which marshaling is naturally provided.

RPC has been a dominant technology for IPC for quite a long time, and several programming languages and environments support this interaction pattern in the form of libraries and additional packages. For instance, RPyC is an RPC implementation for Python. There also exist platform-independent solutions such as XML-RPC and JSON-RPC, which provide RPC facilities over XML and JSON, respectively. Currently, the term RPC implementations encompass a variety of solutions including frameworks such distributed object programming (CORBA, DCOM, Java RMI, and .NET Remoting) and Web services that evolved from the original RPC concept.

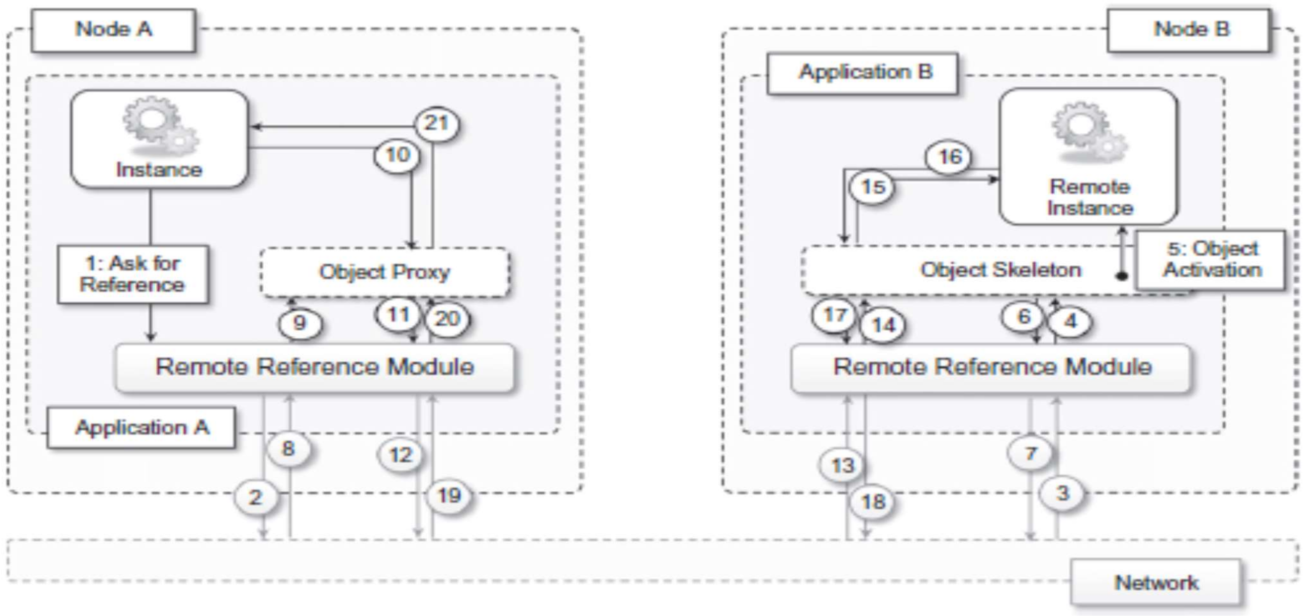
Distributed object frameworks

Distributed object frameworks extend object-oriented programming systems by allowing objects to be distributed across a heterogeneous network and provide facilities so that they can coherently act as though they were in the same address space. Distributed object frameworks leverage the basic mechanism introduced with RPC and extend it to enable the remote invocation of object methods and to keep track of references to objects made available through a network connection.

With respect to the RPC model, the infrastructure manages instances that are exposed through well-known interfaces instead of procedures. Therefore, the common interaction pattern is the following:

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

Distributed object frameworks give the illusion of interaction with a local instance while invoking remote methods. This is done by a mechanism called a *proxy skeleton.* Figure 2.15 gives an overview of how this infrastructure works. Proxy and skeleton always constitute a pair: the server process maintains the skeleton component, which is in charge of executing the methods that are remotely invoked, while the client maintains the proxy component, allowing its hosting environment to remotely invoke methods through the proxy interface.



The transparency of remote method invocation is achieved using one of the fundamental properties of object-oriented programming: inheritance and subclassing. Both the proxy and the active remote object expose the same interface, defining the set of methods that can be remotely called. On the client side, a runtime object subclassing the type published by the server is generated. This object translates the local method invocation into an RPC call for the corresponding method on the remote active object. On the server side, whenever an RPC request is received, it is unpacked and the method call is dispatched to the skeleton that is paired with the client that issued the request. Once the method execution on the server is completed, the return values are packed and sent back to the client, and the local method call on the proxy returns.

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.

Object activation and lifetime

The management of distributed objects poses additional challenges with respect to the simple invocation of a procedure on a remote node. Methods live within the context of an object instance, and they can alter the internal state of the object as a side effect of their execution. In particular, the lifetime of an object instance is a crucial element in distributed object-oriented systems. Within a single memory address space scenario, objects are explicitly created by the programmer, and their references are made available by passing them from one object instance to another. The memory allocated for them can be explicitly reclaimed by the programmer or automatically by the runtime system when there are no more references to that instance. A distributed scenario introduces additional issues that require a different management of the lifetime of objects exposed through remote interfaces.

The first element to be considered is the object's *activation,* which is the creation of a remote object. Various strategies can be used to manage object activation, from which we can distinguish two major classes: *server-based activation* and *client-based activation.* In server-based activation, the active object is created in the server process and registered as an instance that can be exposed beyond process boundaries. In this case, the active object has a life of its own and occasionally executes methods as a consequence of a remote method invocation. In client-based activation the active object does not originally exist on the server side; it is created when a request for method invocation comes from a client. This scenario is generally more appropriate when the active object is meant to be stateless and should exist for the sole purpose of invoking methods from remote clients. For example, if the remote object is simply a gateway to access and modify other components hosted within the server process, client-based activation is a more efficient pattern.

The second element to be considered is the lifetime of remote objects. In the case of server- based activation, the lifetime of an object is generally user-controlled, since the activation of the remote object is explicit and controlled by the user. In the case of client-based activation, the creation of the remote object is implicit, and therefore its lifetime is controlled by some policy of the runtime infrastructure. Different policies can be considered; the simplest one implies the creation of a new instance for each method invocation. This solution is quite demanding in terms of object instances and is generally integrated with some lease management strategy that allows objects to be reused for subsequent method invocations if they occur within a specified time interval (lease). Another policy might consider having only a single instance at a time, and the lifetime of the object is then controlled by the number and frequency of method calls. Different frameworks provide different levels of control of this aspect.

Object activation and lifetime management are features that are now supported to some extent in almost all the frameworks for distributed object programming, since they are essential to understanding the behavior of a distributed system. In particular, these two aspects are becoming fundamental in designing components that are accessible from other processes and that maintain states. Understanding how many objects representing the same component are created and for how long they last is essential in tracking inconsistencies due to erroneous updates to the instance internal data.

Examples of distributed object frameworks

The support for distributed object programming has evolved over time, and today it is a common feature of mainstream programming languages such as C# and Java, which provide these capabilities as part of the base class libraries.

Common object request broker architecture (CORBA)

CORBA is a specification introduced by the Object Management Group (OMG) for providing crossplatform and cross-language interoperability among distributed components. The specification was originally designed to provide an interoperation standard that could be effectively used at the industrial level.

Distributed component object model (DCOM/COM+)

DCOM, later integrated and evolved into COM+, is the solution provided by Microsoft for distributed object programming before the introduction of .NET technology. DCOM introduces a set of features allowing the use of COM components beyond the process boundaries. A COM object identifies a component that encapsulates a set of coherent and related operations; it was designed to be easily plugged into another application to leverage the features exposed through its interface. To support interoperability, COM standardizes a binary format, thus allowing the use of COM objects across different programming languages. DCOM enables such capabilities in a distributed environment by adding the required IPC support. The architecture of DCOM is quite similar to CORBA but simpler, since it does not aim to foster the same level of interoperability; its implementation is monopolized by Microsoft, which provides a single runtime environment.

Java remote method invocation (RMI)

Java RMI is a standard technology provided by Java for enabling RPC among distributed Java objects. RMI defines an infrastructure allowing the invocation of methods on objects that are located on different Java Virtual Machines (JVMs) residing either on the local node or on a remote one. As with CORBA, RMI is based on the *stub-skeleton* concept. Developers define an interface extending *java.rmi.Remote* that defines the contract for IPC. Java allows only publishing interfaces while it relies on actual types for the server and client part implementation.

NET remoting

Remoting is the technology allowing for IPC among .NET applications. It provides developers with a uniform platform for accessing remote objects from within any application developed in any of the languages supported by .NET. With respect to other distributed object technologies, Remoting is a fully customizable architecture that allows developers to control the transport protocols used to exchange information between the proxy and the remote object, the serialization format used to encode data, the lifetime of remote objects, and the server management of remote objects. Despite its modular and fully customizable architecture, Remoting allows a transparent interaction pattern with objects residing on different application domains. An application domain represents an isolated execution environment that can be accessible only through Remoting channels. A single process can host multiple application domains and must have at least one.

Service-oriented computing

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

What is a service?

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. Don Box identifies four major characteristics that identify a service:

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

- *Services share schema and contracts, not class or interface definitions.* Services are not expressed in terms of classes or interfaces, as happens in object-oriented systems, but they define themselves in terms of schemas and contracts. A service advertises a contract describing the structure of messages it can send and/or receive and additional constraint—if any—on their ordering. Because they are not expressed in terms of types and classes, services are more easily consumable in wider and heterogeneous environments. At the same time, a service orientation requires that contracts and schema remain stable over time, since it would be possible to propagate changes to all its possible clients. To address this issue, contracts and schema are defined in a way that allows services to evolve without breaking already deployed code. Technologies such as XML and SOAP provide the appropriate tools to support such features rather than class definition or an interface declaration.

  - *Services compatibility is determined based on policy.* Service orientation separates structural compatibility from semantic compatibility. Structural compatibility is based on contracts and schema and can be validated or enforced by machine-based techniques. Semantic compatibility is expressed in the form of policies that define the capabilities and requirements for a service. Policies are organized in terms of expressions that must hold true to enable the normal operation of a service.

Service-oriented architecture (SOA)

SOA is an architectural style supporting service orientation. It organizes a software system into a collection of interacting services. SOA encompasses a set of design principles that structure system development and provide means for integrating components into a coherent and decentralized system. SOA-based computing packages functionalities into a set of interoperable services, which can be integrated into different software systems belonging to separate business domains.

There are two major roles within SOA: the *service provider* and the *service consumer.* The service provider is the maintainer of the service and the organization that makes available one or more services for others to use. To advertise services, the provider can publish them in a registry, together with a service contract that specifies the nature of the service, how to use it, the requirements for the service, and the fees charged. The service consumer can locate the service metadata in the registry and develop the required client components to bind and use the service. Service providers and consumers can belong to different organization bodies or business domains. It is very common in SOA-based computing systems that components play the roles of both service provider and service consumer. Services might aggregate information and data retrieved from other services or create workflows of services to satisfy the request of a given service consumer. This practice is known as *service orchestration,* which more generally describes the automated arrangement, coordination, and management of complex computer systems, middleware, and services. Another important interaction pattern is *service choreography,* which is the coordinated interaction of services without a single point of control.

SOA provides a reference model for architecting several software systems, especially enterprise business applications and systems. In this context, interoperability, standards, and service contracts play a fundamental role. In particular, the following guiding principles, which characterize SOA platforms, are winning features within an enterprise context:

- *Standardized service contract.* Services adhere to a given communication agreement, which is specified through one or more service description documents.
- *Loose coupling.* Services are designed as self-contained components, maintain relationships that

minimize dependencies on other services, and only require being aware of each other. Service contracts will enforce the required interaction among services. This simplifies the flexible aggregation of services and enables a more agile design strategy that supports the evolution of the enterprise business.

- *Abstraction.* A service is completely defined by service contracts and description documents. They hide their logic, which is encapsulated within their implementation. The use of service description documents and contracts removes the need to consider the technical implementation details and provides a more intuitive framework to define software systems within a business context.
- *Reusability*. Designed as components, services can be reused more effectively, thus reducing development time and the associated costs. Reusability allows for a more agile design and cost-effective system implementation and deployment. Therefore, it is possible to leverage third-party services to deliver required functionality by paying an appropriate fee rather developing the same capability in-house.
- *Autonomy.* Services have control over the logic they encapsulate and, from a service consumer point of view, there is no need to know about their implementation.
- *Lack of state.* By providing a stateless interaction pattern (at least in principle), services increase the chance of being reused and aggregated, especially in a scenario in which a single service is used by multiple consumers that belong to different administrative and business domains.
- *Discoverability.* Services are defined by description documents that constitute supplemental metadata through which they can be effectively discovered. Service discovery provides an effective means for utilizing third-party resources.
- *Composability.* Using services as building blocks, sophisticated and complex operations can be implemented. Service orchestration and choreography provide a solid support for composing services and achieving business goals.

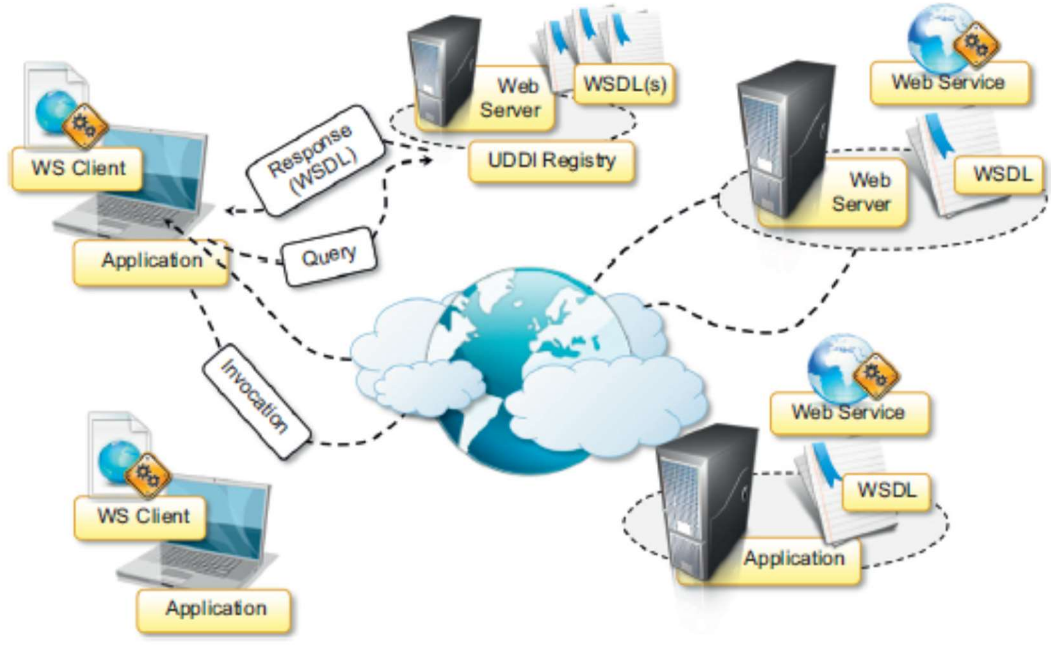
SOA can be realized through several technologies. The first implementations of SOA have leveraged distributed object programming technologies such as CORBA and DCOM. In particular, CORBA has been a suitable platform for realizing SOA systems because it fosters interoperability among different implementations and has been designed as a specification supporting the development of industrial applications. Nowadays, SOA is mostly realized through Web services technology, which provides an interoperable platform for connecting systems and applications.

Web services

Web services are the prominent technology for implementing SOA systems and applications. They leverage Internet technologies and standards for building distributed systems. Several aspects make Web services the technology of choice for SOA. First, they allow for interoperability across different platforms and programming languages. Second, they are based on well-known and vendor-independent standards such as HTTP, SOAP, XML, and WSDL. Third, they provide an intuitive and simple way to connect heterogeneous software systems, enabling the quick composition of services in a distributed environment. Finally, they provide the features required by enterprise business applications to be used in an industrial environment. They define facilities for enabling service discovery, which allows system architects to more efficiently compose SOA applications, and service metering to assess whether a specific service complies with the contract between the service provider and the service consumer.

The concept behind a Web service is very simple. Using as a basis the object-oriented abstraction, a Web service exposes a set of operations that can be invoked by leveraging Internet-based protocols. Method operations support parameters and return values in the form of complex and simple types. The semantics for invoking Web service methods is expressed through interoperable standards such as XML and WSDL, which also provide a complete framework for expressing simple and complex types in a platform-independent manner. Web services are made accessible by being hosted in a Web server; therefore, HTTP is the most popular transport protocol used for interacting with Web services. Below figure describes the common-use case scenarios for Web services.

System architects develop a Web service with their technology of choice and deploy it in compatible Web or application servers. The service description document, expressed by means of Web Service Definition Language (WSDL), can be either uploaded to a global registry or attached as a metadata to the service itself. Service consumers can look up and discover services in global catalogs using Universal Description Discovery and Integration (UDDI) or, most likely, directly retrieve the service metadata by interrogating the Web service first.



The Web service description document allows service consumers to automatically generate clients for the given service and embed them in their existing application. Web services are now extremely popular, so bindings exist for any mainstream programming language in the form of libraries or development support tools. This makes the use of Web services seamless and straightforward with respect to technologies such as CORBA that require much more integration effort. Moreover, being interoperable, Web services constitute a better solution for SOA with respect to several distributed object frameworks, such as .NET Remoting, Java RMI, and DCOM/COM+ , which limit their applicability to a single platform or environment.

Service orientation and cloud computing

Web services and Web 2.0-related technologies constitute a fundamental building block for cloud computing systems and applications. Web 2.0 applications are the front end of cloud computing systems, which deliver services either via Web service or provide a profitable interaction with AJAX-based clients.

Essentially, cloud computing fosters the vision of *Everything as a Service (XaaS):* infrastructure, platform, services, and applications. The entire IT computing stack—from infrastructure to applications—can be composed by relying on cloud computing services. Within this context, SOA is a winning approach because it encompasses design principles to structure, compose, and deploy software systems in terms of services. Therefore, a service orientation constitutes a natural approach to shaping cloud computing systems because it provides a means to flexibly compose and integrate additional capabilities into existing software systems. Cloud computing is also used to elastically scale and empower existing software applications on demand. Service orientation fosters interoperability and leverages platform-independent technologies by definition. Within this context, it constitutes a natural solution for solving integration issues and favoring cloud computing adoption.

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. The term *virtualization* is often synonymous with *hardware virtualization,* which plays a fundamental role in efficiently delivering *Infrastructure-as-a-Service* (IaaS) solutions for cloud computing. In fact, virtualization technologies have a long trail in the history of computer science and have been available in many flavors by providing virtual environments at the operating system level, the programming language level, and the application level. Moreover, virtualization technologies provide a virtual environment for not only executing applications but also for storage, memory, and networking.

Virtualization technologies have gained renewed interested recently due to the confluence of several phenomena:

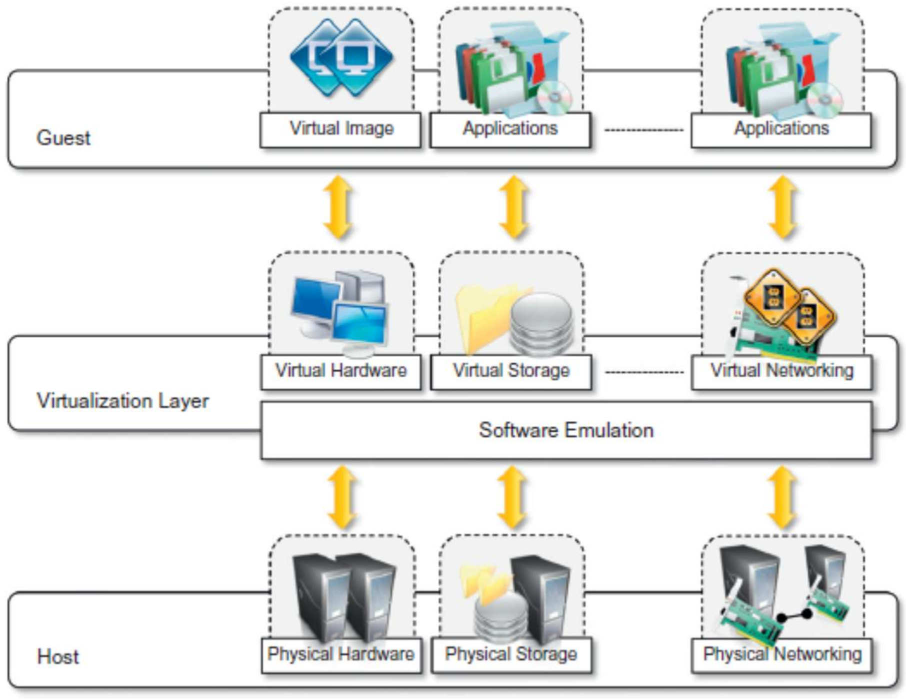
- ***Increased performance and computing capacity.*** Nowadays, the average end-user desktop PC is powerful enough to meet almost all the needs of everyday computing, with extra capacity that is rarely used. Almost all these PCs have resources enough to host a virtual machine manager and execute a virtual machine with by far acceptable performance. The same consideration applies to the high-end side of the PC market, where supercomputers can provide immense compute power that can accommodate the execution of hundreds or thousands of virtual machines.
- ***Underutilized hardware and software resources.*** Hardware and software underutilization is occurring due to (1) increased performance and computing capacity, and (2) the effect of limited or sporadic use of resources. Computers today are so powerful that in most cases only a fraction of their capacity is used by an application or the system. Moreover, if we consider the IT infrastructure of an enterprise, many computers are only partially utilized whereas they could be used without interruption on a 24/7/365 basis. For example, desktop PCs mostly devoted to office automation tasks and used by administrative staff are only used during work hours, remaining completely unused overnight. Using these resources for other purposes after hours could improve the efficiency of the IT infrastructure. To transparently provide such a service, it would be necessary to deploy a completely separate environment, which can be achieved through virtualization.
- ***Lack of space.*** The continuous need for additional capacity, whether storage or compute power, makes data centers grow quickly. Companies such as Google and Microsoft expand their infrastructures by building data centers as large as football fields that are able to host thousands of nodes. Although this is viable for IT giants, in most cases enterprises cannot afford to build another data center to

accommodate additional resource capacity. This condition, along with hardware underutilization, has led to the diffusion of a technique called *server consolidation,* for which virtualization technologies are fundamental.

- ***Greening initiatives.*** Recently, companies are increasingly looking for ways to reduce the amount of energy they consume and to reduce their carbon footprint. Data centers are one of the major power consumers; they contribute consistently to the impact that a company has on the environment. Maintaining a data center operation not only involves keeping servers on, but a great deal of energy is also consumed in keeping them cool. Infrastructures for cooling have a significant impact on the carbon footprint of a data center. Hence, reducing the number of servers through server consolidation will definitely reduce the impact of cooling and power consumption of a data center. Virtualization technologies can provide an efficient way of consolidating servers.
- ***Rise of administrative costs.*** Power consumption and cooling costs have now become higher than the cost of IT equipment. Moreover, the increased demand for additional capacity, which translates into more servers in a data center, is also responsible for a significant increment in administrative costs. Computers—in particular, servers—do not operate all on their own, but they require care and feeding from system administrators. Common system administration tasks include hardware monitoring, defective hardware replacement, server setup and updates, server resources monitoring, and backups. These are labor-intensive operations, and the higher the number of servers that have to be managed, the higher the administrative costs. Virtualization can help reduce the number of required servers for a given workload, thus reducing the cost of the administrative personnel.
-

Characteristics of virtualized environments

Virtualization is a broad concept that refers to the creation of a virtual version of something, whether hardware, a software environment, storage, or a network. 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.



The virtualization reference model.

Such a general abstraction finds different applications and then implementations of the virtualization technology. The most intuitive and popular is represented by *hardware virtualization,* which also constitutes the original realization of the virtualization concept. In the case of hardware virtualization, the guest is represented by a system image comprising an operating system and installed applications. These are installed on top of virtual hardware that is controlled and managed by the virtualization layer, also called the *virtual machine manager.* The host is instead represented by the physical hardware, and in some cases the operating system, that defines the environment where the virtual machine manager is running. In the case of virtual storage, the guest might be client applications or users that interact with the virtual storage management software deployed on top of the real storage system. The case of virtual networking is also similar: The guest— applications and users—interacts with a virtual network, such as a *virtual private network (VPN),* which is managed by specific software (VPN client) using the physical network available on the node. VPNs are useful for creating the illusion of being within a different physical network and thus accessing the resources in it, which would otherwise not be available.

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 ability to use software to emulate such a wide variety of environments creates a lot of opportunities, previously less attractive because of excessive overhead introduced by the virtualization layer.

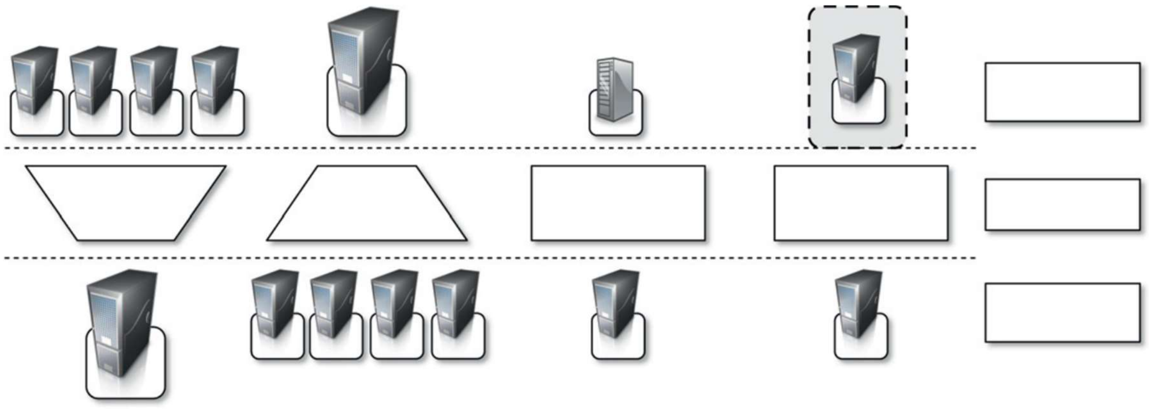
Increased security

The ability to control the execution of a guest in a completely transparent manner opens new possibilities for delivering a secure, controlled execution environment. The virtual machine represents an emulated environment in which the guest is executed. All the operations of the guest are generally performed against the virtual machine, which then translates and applies them to the host. This level of indirection allows the virtual machine manager to *control* and *filter* the activity of the guest, thus preventing some harmful operations from being performed. Resources exposed by the host can then be hidden or simply protected from the guest.

Moreover, 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. For example, applets downloaded from the Internet run in a sandboxed version of the *Java Virtual Machine (JVM),* which provides them with limited access to the hosting operating system resources. Both the JVM and the .NET runtime provide extensive security policies for customizing the execution environment of applications. Hardware virtualization solutions such as VMware Desktop, VirtualBox, and Parallels provide the ability to create a virtual computer with customized virtual hardware on top of which a new operating system can be installed. By default, the file system exposed by the virtual computer is completely separated from the one of the host machine. This becomes the perfect environment for running applications without affecting other users in the environment.

Managed execution

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



Functions enabled by managed execution

• *Sharing.* Virtualization allows the creation of a separate computing environments within the same host. In this way it is possible to fully exploit the capabilities of a powerful guest, which would otherwise be underutilized. As we will see in later chapters, sharing is a particularly important feature in virtualized data centers, where this basic feature is used to reduce the number of active servers and limit power consumption.

- *Aggregation.* Not only is it possible to share physical resource among several guests, but virtualization also allows aggregation, which is the opposite process. A group of separate hosts can be tied together and represented to guests as a single virtual host. This function is naturally implemented in middleware for distributed computing, with a classical example represented by cluster management software, which harnesses the physical resources of a homogeneous group of machines and represents them as a single resource.
- *Emulation.* Guest programs are executed within an environment that is controlled by the virtualization layer, which ultimately is a program. This allows for controlling and tuning the environment that is exposed to guests. For instance, a completely different environment with respect to the host can be

emulated, thus allowing the execution of guest programs requiring specific characteristics that are not present in the physical host. This feature becomes very useful for testing purposes, where a specific guest has to be validated against different platforms or architectures and the wide range of options is not easily accessible during development. Again, hardware virtualization solutions are able to provide virtual hardware and emulate a particular kind of device such as *Small Computer System Interface (SCSI)* devices for file I/O, without the hosting machine having such hardware installed. Old and legacy software that does not meet the requirements of current systems can be run on emulated hardware without any need to change the code. This is possible either by emulating the required hardware architecture or within a specific operating system sandbox, such as the MS-DOS mode in Windows 95/98. Another example of emulation is an arcade-game emulator that allows us to play arcade games on a normal personal computer.

- *Isolation.* Virtualization allows providing guests—whether they are operating systems, applications, or other entities—with a completely separate environment, in which they are executed. The guest program performs its activity by interacting with an abstraction layer, which provides access to the underlying resources. Isolation brings several benefits; for example, it allows multiple guests to run on the same host without interfering with each other. Second, it provides a separation between the host and the guest. The virtual machine can filter the activity of the guest and prevent harmful operations against the host.

Besides these characteristics, another important capability enabled by virtualization is *performance tuning*. This feature is a reality at present, given the considerable advances in hardware and software supporting virtualization. It becomes easier to control the performance of the guest by finely tuning the properties of the resources exposed through the virtual environment. This capability provides a means to effectively implement a quality-of-service (QoS) infrastructure that more easily fulfills the service-level agreement (SLA) established for the guest.

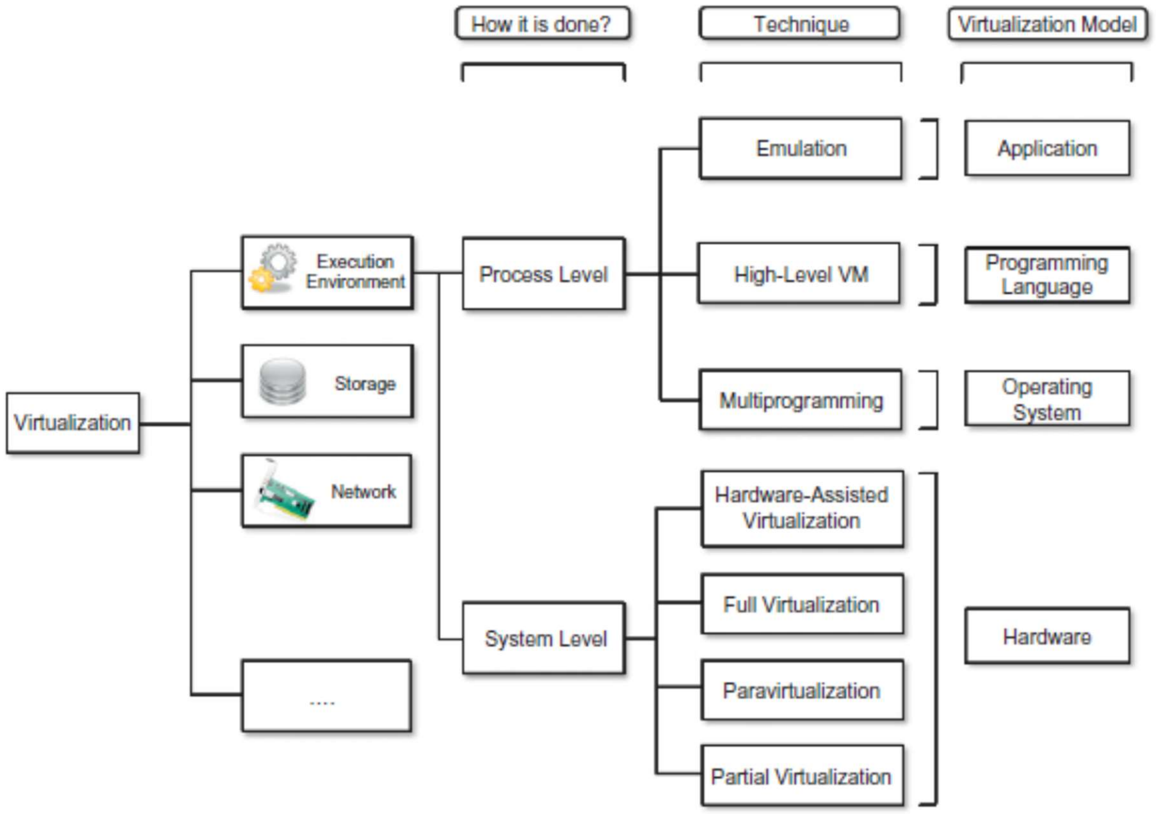
Portability

The concept of *portability* applies in different ways according to the specific type of virtualization considered. 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. Except for the file size, this happens with the same simplicity with which we can display a picture image in different computers. Virtual images are generally proprietary formats that require a specific virtual machine manager to be executed. In the case of programming-level virtualization, as implemented by the JVM or the .NET runtime, the binary code representing application components (jars or assemblies) can be run without any recompilation on any implementation of the corresponding virtual machine. This makes the application development cycle more flexible and application deployment very straightforward: One version of the application, in most cases, is able to run on different platforms with no changes. Finally, portability allows having your own system always with you and ready to use as long as the required virtual machine manager is available. This requirement is, in general, less stringent than having all the applications and services you need available to you anywhere you go.

Taxonomy of virtualization techniques

Virtualization covers a wide range of emulation techniques that are applied to different areas of computing. A classification of these techniques helps us better understand their characteristics and use.

The first classification discriminates against the service or entity that is being emulated. Virtualization is mainly used to emulate *execution environments, storage,* and *networks.* Among these categories, *execution virtualization* constitutes the oldest, most popular, and most developed area. Therefore, it deserves major investigation and a further categorization. In particular we can divide these execution virtualization techniques into two major categories by considering the type of host they require. *Process-level* techniques are implemented on top of an existing operating system, which has full control of the hardware. *System-level* techniques are implemented directly on hardware and do not require—or require a minimum of support from—an existing operating system. Within these two categories we can list various techniques that offer the guest a different type of virtual computation environment: bare hardware, operating system resources, low-level programming language, and application libraries.



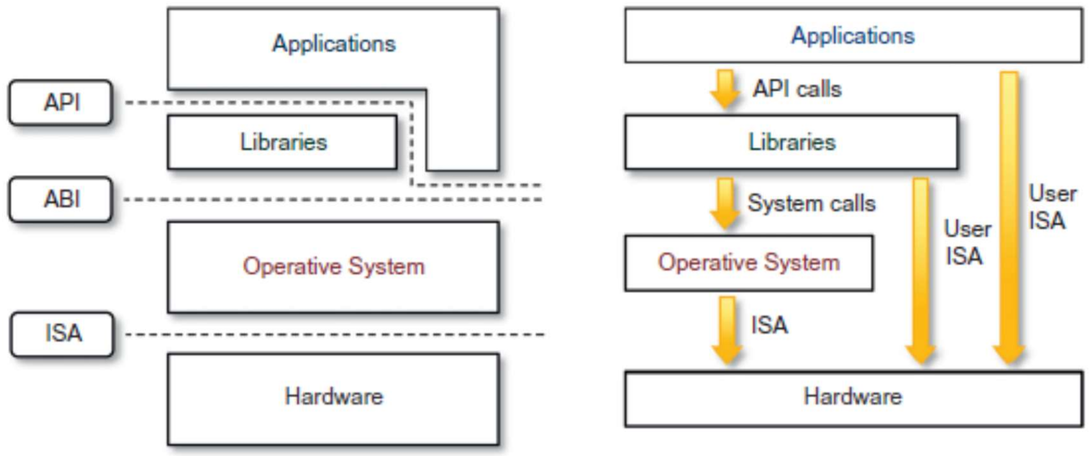
Execution virtualization

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

Machine reference model

Virtualizing an execution environment at different levels of the computing stack requires a reference model that defines the interfaces between the levels of abstractions, which hide implementation details. From this perspective, virtualization techniques actually replace one of the layers and intercept the calls that are directed toward it. Therefore, a clear separation between layers simplifies their implementation, which only requires the emulation of the interfaces and a proper interaction with the underlying layer.

Modern computing systems can be expressed in terms of the reference model described in Figure 3.4. At the bottom layer, the model for the hardware is expressed in terms of the *Instruction Set Architecture (ISA),* which defines the instruction set for the processor, registers, memory, and interrupt management. ISA is the interface between hardware and software, and it is important to the operating system (OS) developer *(System ISA)* and developers of applications that directly manage the underlying hardware *(User ISA).* The *application binary interface (ABI)* separates the operating system layer from the applications and libraries, which are managed by the OS. ABI covers details such as low-level data types, alignment, and call conventions and defines a format for executable programs. System calls are defined at this level. This interface allows portability of applications and libraries across operating systems that implement the same ABI. The highest level of abstraction is represented by the *application programming interface (API),* which interfaces applications to libraries and/or the underlying operating system.



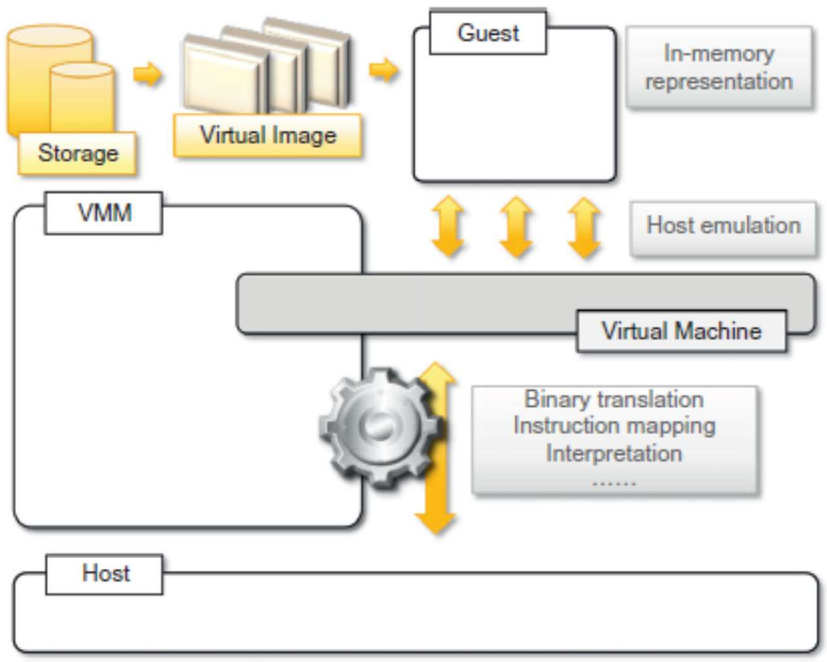
For any operation to be performed in the application level API, ABI and ISA are responsible for making it happen. The high-level abstraction is converted into machine-level instructions to perform the actual operations supported by the processor. The machine-level resources, such as processor registers and main memory capacities, are used to perform the operation at the hardware level of the central processing unit (CPU). This layered approach simplifies the development and implementation of computing systems and simplifies the implementation of multitasking and the coexistence of multiple executing environments. In fact, such a model not only requires limited knowledge of the entire computing stack, but it also provides ways to implement a minimal security model for managing and accessing shared resources.

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 *master mode* or *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.

The distinction between *user* and *supervisor* mode allows us to understand the role of the *hypervisor* and why it is called that. Conceptually, the hypervisor runs above the supervisor mode, and from here the prefix *hyper-* is used. 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.

Hardware-level virtualization

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



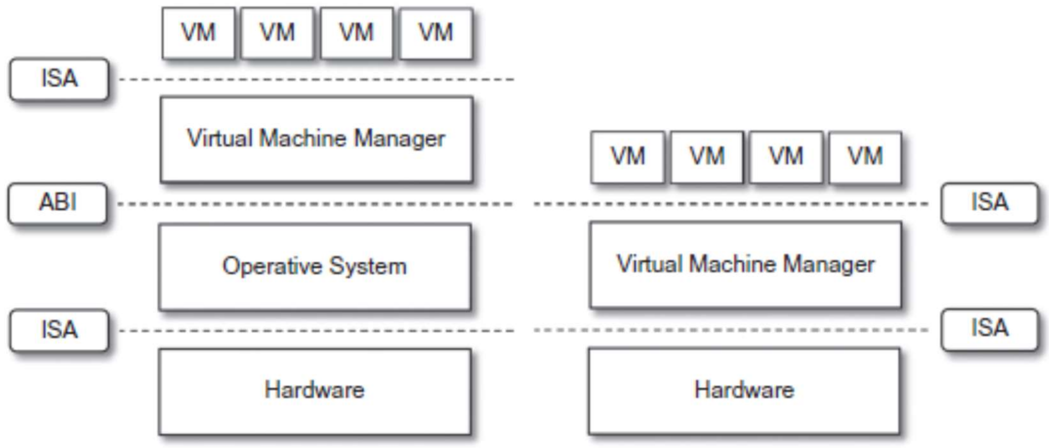
Hardware-level virtualization is also called *system virtualization,* since it provides ISA to virtual machines, which is the representation of the hardware interface of a system. This is to differentiate it from *process virtual machines,* which expose ABI to virtual machines.

Hypervisors

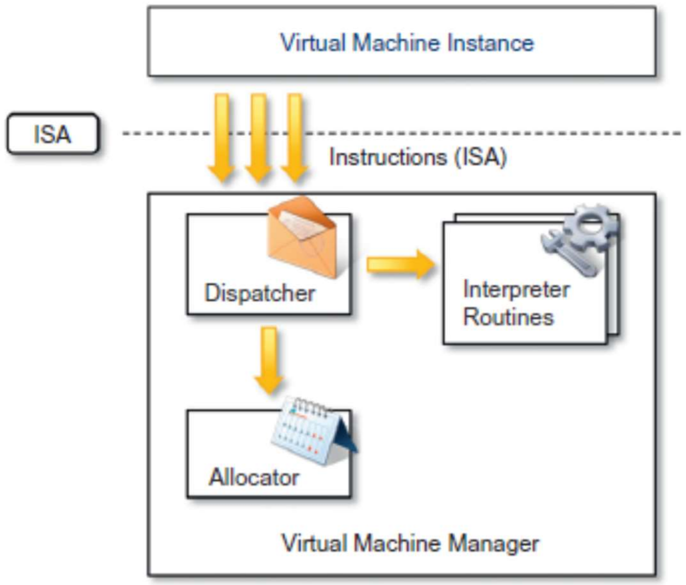
A fundamental element of hardware virtualization is the hypervisor, or virtual machine manager (VMM). It recreates a hardware environment in which guest operating systems are installed. There are two major types of hypervisor: *Type I* and *Type II*.

- *Type I* hypervisors run directly on top of the hardware. Therefore, they take the place of the operating systems and interact directly with the ISA interface exposed by the underlying hardware, and they emulate this interface in order to allow the management of guest operating systems. This type of hypervisor is also called a *native virtual machine* since it runs natively on hardware.
- *Type II* hypervisors require the support of an operating system to provide virtualization services. This means that they are programs managed by the operating system, which interact with it through the ABI and emulate the ISA of virtual hardware for guest operating systems. This type of hypervisor is also

called a *hosted virtual machine* since it is hosted within an operating system.



Conceptually, a virtual machine manager is internally organized as described in below Figure . Three main modules, *dispatcher*, *allocator*, and *interpreter*, coordinate their activity in order to emulate the underlying hardware. The dispatcher constitutes the entry point of the monitor and reroutes the instructions issued by the virtual machine instance to one of the two other modules. The allocator is responsible for deciding the system resources to be provided to the VM: whenever a virtual machine tries to execute an instruction that results in changing the machine resources associated with that VM, the allocator is invoked by the dispatcher. The interpreter module consists of interpreter routines. These are executed whenever a virtual machine executes a privileged instruction: a trap is triggered and the corresponding routine is executed.



The design and architecture of a virtual machine manager, together with the underlying hardware design of the host machine, determine the full realization of hardware virtualization, where a guest operating system can be transparently executed on top of a VMM as though it were run on the underlying hardware. Three properties have to be satisfied:

- *Equivalence.* A guest running under the control of a virtual machine manager should exhibit the same behavior as when it is executed directly on the physical host.

- *Resource control.* The virtual machine manager should be in complete control of virtualized resources.
- *Efficiency.* A statistically dominant fraction of the machine instructions should be executed without intervention from the virtual machine manager.

Hardware virtualization techniques

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. This technique was originally introduced in the IBM System/370. At present, examples of hardware-assisted virtualization are the extensions to the x86-64 bit architecture introduced with *Intel VT* (formerly known as *Vanderpool)* and *AMD V* (formerly known as *Pacifica).* These extensions, which differ between the two vendors, are meant to reduce the performance penalties experienced by emulating x86 hardware with hypervisors. Before the introduction of hardware-assisted virtualization, software emulation of x86 hardware was significantly costly from the performance point of view.

Full virtualization. *Full virtualization* refers to the ability to run a program, most likely an operating system, directly on top of a virtual machine and without any modification, as though it were run on the raw hardware. To make this possible, virtual machine managers are required to provide a complete emulation of the entire underlying hardware. The principal advantage of full virtualization is complete isolation, which leads to enhanced security, ease of emulation of different architectures, and coexistence of different systems on the same platform. Whereas it is a desired goal for many virtualization solutions, full virtualization poses important concerns related to performance and technical implementation. A key challenge is the interception of privileged instructions such as I/O instructions: Since they change the state of the resources exposed by the host, they have to be contained within the virtual machine manager. A simple solution to achieve full virtualization is to provide a virtual environment for all the instructions, thus posing some limits on performance. A successful and efficient implementation of full virtualization is obtained with a combination of hardware and software, not allowing potentially harmful instructions to be executed directly on the host. This is what is accomplished through hardware-assisted virtualization.

Paravirtualization. This is a not-transparent virtualization solution that allows implementing thin virtual machine managers. Paravirtualization techniques expose a software interface to the virtual machine that is slightly modified from the host and, as a consequence, guests need to be modified. The aim of paravirtualization is to provide the capability to demand the execution of performance-critical operations directly on the host, thus preventing performance losses that would otherwise be experienced in managed execution. This allows a simpler implementation of virtual machine managers that have to simply transfer the execution of these operations, which were hard to virtualize, directly to the host. To take advantage of such an opportunity, guest operating systems need to be modified and explicitly ported by remapping the performance-critical operations through the virtual machine software interface. This is possible when the source code of the operating system is available, and this is the reason that paravirtualization was mostly explored in the open- source and academic environment. 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. Partial virtualization provides a partial emulation of the underlying hardware, thus not allowing the complete execution of the guest operating system in complete isolation. Partial virtualization allows many applications to run transparently, but not all the features of the operating system can be supported, as happens with full virtualization. An example of partial virtualization is address space virtualization used in time-sharing systems; this allows multiple applications and users to run concurrently in a separate memory space, but they still share the same hardware resources (disk, processor, and network). Historically, partial virtualization has been an important milestone for achieving full virtualization, and it was implemented on the experimental IBM M44/44X. Address space virtualization is a common feature of contemporary operating systems.

Operating system-level virtualization

Operating system-level virtualization offers the opportunity to create different and separated execution environments for applications that are managed concurrently. Differently from hardware virtualization, there is no virtual machine manager or hypervisor, and the virtualization is done within a single operating system, where the OS kernel allows for multiple isolated user space instances. The kernel is also responsible for sharing the system resources among instances and for limiting the impact of instances on each other. A user space instance in general contains a proper view of the file system, which is completely isolated, and separate IP addresses, software configurations, and access to devices. Operating systems supporting this type of virtualization are general-purpose, time- shared operating systems with the capability to provide stronger namespace and resource isolation.

This virtualization technique can be considered an evolution of the *chroot* mechanism in Unix systems. The *chroot* operation changes the file system root directory for a process and its children to a specific directory. As a result, the process and its children cannot have access to other portions of the file system than those accessible under the new root directory. Because Unix systems also expose devices as parts of the file system, by using this method it is possible to completely isolate a set of processes. Following the same principle, operating system-level virtualization aims to provide separated and multiple execution containers for running applications. Compared to hardware virtualization, this strategy imposes little or no overhead because applications directly use OS system calls and there is no need for emulation. There is no need to modify applications to run them, nor to modify any specific hardware, as in the case of hardware-assisted virtualization.

This technique is an efficient solution for server consolidation scenarios in which multiple application servers share the same technology: operating system, application server framework, and other components. When different servers are aggregated into one physical server, each server is run in a different user space, completely isolated from the others.

Examples of operating system-level virtualizations are FreeBSD Jails, IBM Logical Partition (LPAR), SolarisZones and Containers, Parallels Virtuozzo Containers, and others.

Programming language-level virtualization

Programming language-level virtualization is mostly used to achieve ease of deployment of applications, managed execution, and portability across different platforms and operating systems. It consists of a virtual machine executing the byte code of a program, which is the result of the compilation process. Compilers implemented and used this technology to produce a binary format representing the machine code for an abstract architecture. The characteristics of this architecture vary from implementation to implementation. Generally these virtual machines constitute a simplification of the underlying hardware instruction set and provide some high-level instructions that map some of the features of the languages compiled for them. At runtime, the byte code can be either interpreted or compiled on the fly—or *jitted*—against the underlying hardware instruction set.

Programming language-level virtualization has a long trail in computer science history and originally was used in 1966 for the implementation of *Basic Combined Programming Language (BCPL),* a language for writing compilers and one of the ancestors of the C programming language. Virtual machine programming languages become popular again with Sun's introduction of the Java platform in 1996. Originally created as a platform for developing Internet applications, Java became one of the technologies of choice for enterprise applications, and a large community of developers formed around it. The Java virtual machine was originally designed for the execution of programs written in the Java language, but other languages such as Python, Pascal, etc. were made available. The ability to support multiple programming languages has been one of the key elements of the *Common Language Infrastructure (CLI),* which is the specification behind .NET Framework. Currently, the Java platform and .NET Framework represent the most popular technologies for enterprise application development.

Both Java and the CLI are *stack-based* virtual machines: The reference model of the abstract architecture is based on an execution stack that is used to perform operations. The byte code generated by compilers for these architectures contains a set of instructions that load operands on the stack, perform some operations with them, and put the result on the stack. Additionally, specific instructions for invoking methods and managing objects and classes are included.

The main advantage of programming-level virtual machines, also called *process virtual machines,* is the ability to provide a uniform execution environment across different platforms. Programs compiled into byte code can be executed on any operating system and platform for which a virtual machine able to execute that code has been provided. The implementation of the virtual machine for different platforms is still a costly task, but it is done once and not for any application. Moreover, process virtual machines allow for more control over the execution of programs since they do not provide direct access to the memory. Security is another advantage of managed programming languages; by filtering the I/O operations, the process virtual machine can easily support sandboxing of applications. As an example, both Java and .NET provide an infrastructure for pluggable security policies and code access security frameworks. All these advantages come with a price: performance. Virtual machine programming languages generally expose an inferior performance compared to languages compiled against the real architecture. This performance difference is getting smaller, and the high compute power available on average processors makes it even less important.

Application-level virtualization

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

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

Emulation, as described, is different from hardware-level virtualization. The former simply allows the execution of a program compiled against a different hardware, whereas the latter emulates a complete hardware environment where an entire operating system can be installed.

Application virtualization is a good solution in the case of missing libraries in the host operating system; in this case a replacement library can be linked with the application, or library calls can be remapped to existing functions available in the host system. Another advantage is that in this case the virtual machine manager is much lighter since it provides a partial emulation of the runtime environment compared to hardware virtualization. Moreover, this technique allows incompatible applications to run together. Compared to programming-level virtualization, which works across all the applications developed for that virtual machine, application-level virtualization works for a specific environment: It supports all the applications that run on top of a specific environment.

One of the most popular solutions implementing application virtualization is *Wine,* which is a software application allowing Unix-like operating systems to execute programs written for the Microsoft Windows platform. Wine features a software application acting as a container for the guest application and a set of libraries, called *Winelib,* that developers can use to compile applications to be ported on Unix systems.

Storage virtualization

Storage virtualization is a system administration practice that allows decoupling the physical orga-nization of the hardware from its logical representation. Using this technique, users do not have to be worried about the specific location of their data, which can be identified using a logical path.
Storage virtualization allows us to harness a wide range of storage facilities and represent them under a single logical file system. There are different techniques for storage virtualization, one of the most popular being network-based virtualization by means of *storage area networks (SANs).* SANs use a network-accessible device through a large bandwidth connection to provide storage facilities.

Network virtualization

Network virtualization combines hardware appliances and specific software for the creation and management of a virtual network. Network virtualization can aggregate different physical networks into a single logical network *(external* network virtualization) or provide network-like functionality to an operating system partition *(internal* network virtualization). The result of external network virtualization is generally a *virtual LAN (VLAN).* A VLAN is an aggregation of hosts that communicate with each other as though they were located under the same broadcasting domain. Internal network virtualization is generally applied together with hardware and operating system-level virtualization, in which the guests obtain a virtual network interface to communicate with. There are several options for implementing internal network virtualization: The guest can share the same network interface of the host and use Network Address Translation (NAT) to access the network; the virtual machine manager can emulate, and install on the host, an additional network device, together with the driver; or the guest can have a private network only with the guest.

Desktop virtualization

Desktop virtualization abstracts the desktop environment available on a personal computer in order to provide access to it using a client/server approach. Desktop virtualization provides the same outcome of hardware virtualization but serves a different purpose. Similarly to hardware virtualization, desktop virtualization makes accessible a different system as though it were natively installed on the host, but this system is remotely stored on a different host and accessed through a network connection. Moreover, desktop virtualization addresses the problem of making the same desktop environment accessible from everywhere. Although the term *desktop virtualization* strictly refers to the 91
ability to remotely access a desktop environment, generally the desktop environment is stored in a remote server or a data center that provides a high-availability infrastructure and ensures the accessibility and persistence of the data.

Application server virtualization

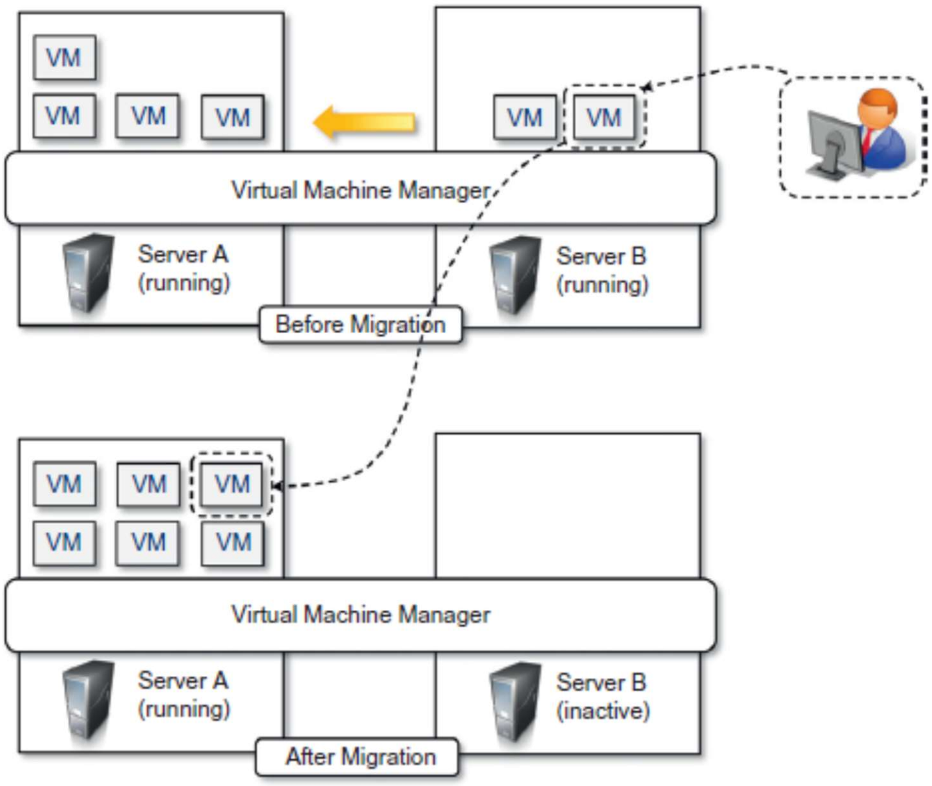
Application server virtualization abstracts a collection of application servers that provide the same services as a single virtual application server by using load-balancing strategies and providing a high-availability infrastructure for the services hosted in the application server. This is a particular form of virtualization and serves the same purpose of storage virtualization: providing a better quality of service rather than emulating a different environment.

Virtualization and cloud computing

Virtualization plays an important role in cloud computing since it allows for the appropriate degree of customization, security, isolation, and manageability that are fundamental for delivering IT services on demand. Virtualization technologies are primarily used to offer configurable computing environments and storage. Network virtualization is less popular and, in most cases, is a complementary feature, which is naturally needed in build virtual computing systems.

Particularly important is the role of virtual computing environment and execution virtualization techniques. Among these, hardware and programming language virtualization are the techniques adopted in cloud computing systems. Hardware virtualization is an enabling factor for solutions in the Infrastructure-as-a-Service (IaaS) market segment, while programming language virtualization is a technology leveraged in Platform-as-a-Service (PaaS) offerings. In both cases, the capability of offering a customizable and sandboxed environment constituted an attractive business opportunity for companies featuring a large computing infrastructure that was able to sustain and process huge workloads. Moreover, virtualization also allows isolation and a finer control, thus simplifying the leasing of services and their accountability on the vendor side.

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 opportunity is particularly attractive when resources are underutilized, because it allows reducing the number of active resources by aggregating virtual machines over a smaller number of resources that become fully utilized. 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.

Server consolidation and virtual machine migration are principally used in the case of hardware virtualization, even though they are also technically possible in the case of programming language virtualization. Storage virtualization constitutes an interesting opportunity given by virtualization technologies, often complementary to the execution of virtualization.

Finally, cloud computing revamps the concept of desktop virtualization, initially introduced in the mainframe era. The ability to recreate the entire computing stack—from infrastructure to application services—on demand opens the path to having a complete virtual computer hosted on the infrastructure of the provider and accessed by a thin client over a capable Internet connection.

Advantages of virtualization

Managed execution and isolation are perhaps the most important advantages of virtualization. In the case of techniques supporting the creation of virtualized execution environments, these two characteristics allow building secure and controllable computing environments. A virtual execution environment can be configured as a sandbox, thus preventing any harmful operation to cross the borders of the virtual host. Moreover, allocation of resources and their partitioning among different guests is simplified, being the virtual host controlled by a program. This enables fine-tuning of resources, which is very important in a server consolidation scenario and is also a requirement for effective quality of service.

Portability is another advantage of virtualization, especially for execution virtualization techniques. Virtual machine instances are normally represented by one or more files that can be easily transported with respect to physical systems. Moreover, they also tend to be self-contained since they do not have other dependencies besides the virtual machine manager for their use. Portability and self-containment simplify their administration. Java programs are "compiled once and run everywhere"; they only require that the Java virtual machine be installed on the host. The same applies to hardware-level virtualization. It is in fact possible to build our own operating environment within a virtual machine instance and bring it with us wherever we go, as though we had our own laptop. This concept is also an enabler for migration techniques in a server consolidation scenario.

Portability and self-containment also contribute to reducing the costs of maintenance, since the number of hosts is expected to be lower than the number of virtual machine instances. Since the guest program is executed in a virtual environment, there is very limited opportunity for the guest program to damage the underlying hardware. Moreover, it is expected that there will be fewer virtual machine managers with respect to the number of virtual machine instances managed.

Finally, by means of virtualization it is possible to achieve a more efficient use of resources. Multiple systems can securely coexist and share the resources of the underlying host, without interfering with each other. This is a prerequisite for server consolidation, which allows adjusting the number of active physical resources dynamically according to the current load of the system, thus creating the opportunity to save in terms of energy consumption and to be less impacting on the environment.

Disadvantages of virtualization

The most evident is represented by a performance decrease of guest systems as a result of the intermediation performed by the virtualization layer. In addition, suboptimal use of the host because of the abstraction layer introduced by virtualization management software can lead to a very inefficient utilization of the host or a degraded user experience.

1. Performance degradation

Performance is definitely one of the major concerns in using virtualization technology. Since virtualization interposes an abstraction layer between the guest and the host, the guest can experience increased latencies.

For instance, in the case of hardware virtualization, where the intermediate emulates a bare machine on top of which an entire system can be installed, the causes of performance degradation can be traced back to the overhead introduced by the following activities:

- Maintaining the status of virtual processors
- Support of privileged instructions (trap and simulate privileged instructions)

- Support of paging within VM
- Console functions

Furthermore, when hardware virtualization is realized through a program that is installed or executed on top of the host operating systems, a major source of performance degradation is represented by the fact that the virtual machine manager is executed and scheduled together with other applications, thus sharing with them the resources of the host.

2. Inefficiency and degraded user experience

Virtualization can sometime lead to an inefficient use of the host. In particular, some of the specific features of the host cannot be exposed by the abstraction layer and then become inaccessible. In the case of hardware virtualization, this could happen for device drivers: The virtual machine can sometime simply provide a default graphic card that maps only a subset of the features available in the host. In the case of programming-level virtual machines, some of the features of the underlying operating systems may become inaccessible unless specific libraries are used. For example, in the first version of Java the support for graphic programming was very limited and the look and feel of applications was very poor compared to native applications.

3. Security holes and new threats

Virtualization opens the door to a new and unexpected form of *phishing.* The capability of emulating a host in a completely transparent manner led the way to malicious programs that are designed to extract sensitive information from the guest.

In the case of hardware virtualization, malicious programs can preload themselves before the operating system and act as a thin virtual machine manager toward it. The operating system is then controlled and can be manipulated to extract sensitive information of interest to third parties.

Logical Network Perimeter

Defined as the isolation of a network environment from the rest of a communications network, the *logical network perimeter* establishes a virtual network boundary that can encompass and isolate a group of related cloud-based IT resources that may be physically distributed.

This mechanism can be implemented to:

- isolate IT resources in a cloud from non-authorized users
- isolate IT resources in a cloud from non-users
- isolate IT resources in a cloud from cloud consumers
- control the bandwidth that is available to isolated IT resources

Logical network perimeters are typically established via network devices that supply and control the connectivity of a data center and are commonly deployed as virtualized IT environments that include:

- *Virtual Firewall* – An IT resource that actively filters network traffic to and from the isolated network while controlling its interactions with the Internet.
- *Virtual Network* – Usually acquired through VLANs, this IT resource isolates the network environment within the data center infrastructure.

Below figure depicts a scenario in which one logical network perimeter contains a cloud consumer's on-premise environment, while another contains a cloud provider's cloud-based environment. These perimeters are connected through a VPN that protects communications, since the VPN is typically implemented by point-to- point encryption of the data packets sent between the communicating endpoints.

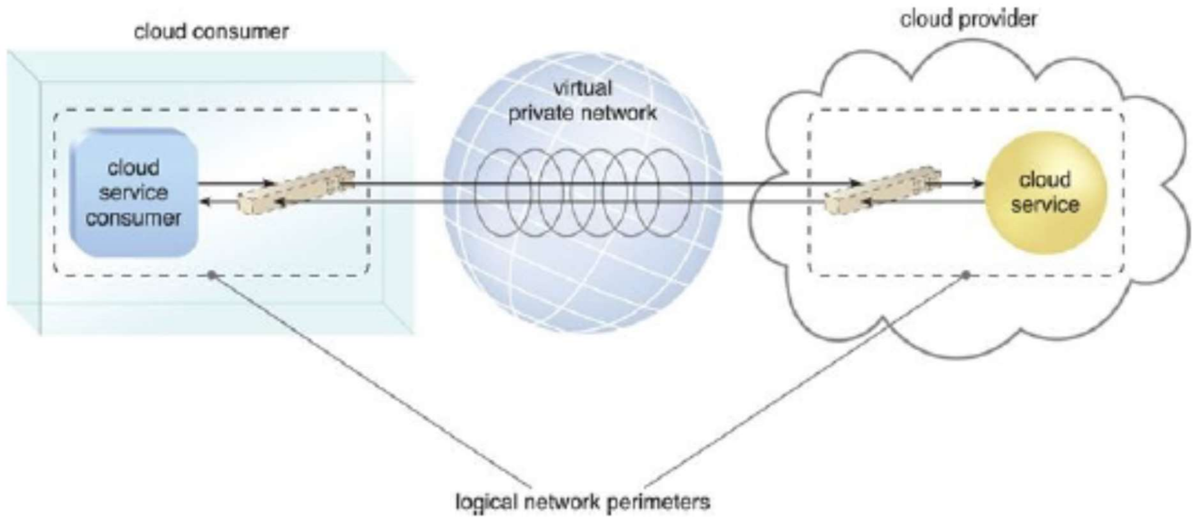


Figure 7.3. Two logical network perimeters surround the cloud consumer and

Virtual Server

A *virtual server* is a form of virtualization software that emulates a physical server. Virtual servers are used by cloud providers to share the same physical server with multiple cloud consumers by providing cloud consumers with individual virtual server instances. Figure 7.5 shows three virtual servers being hosted by two physical servers. The number of instances a given physical server can share is limited by its capacity.

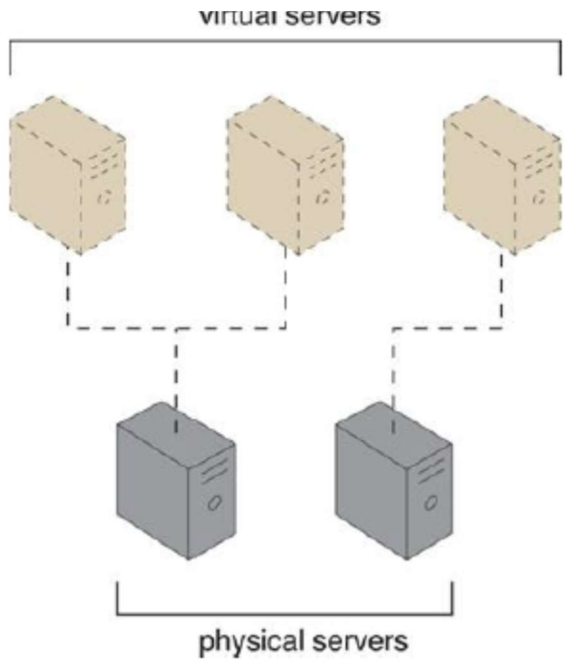
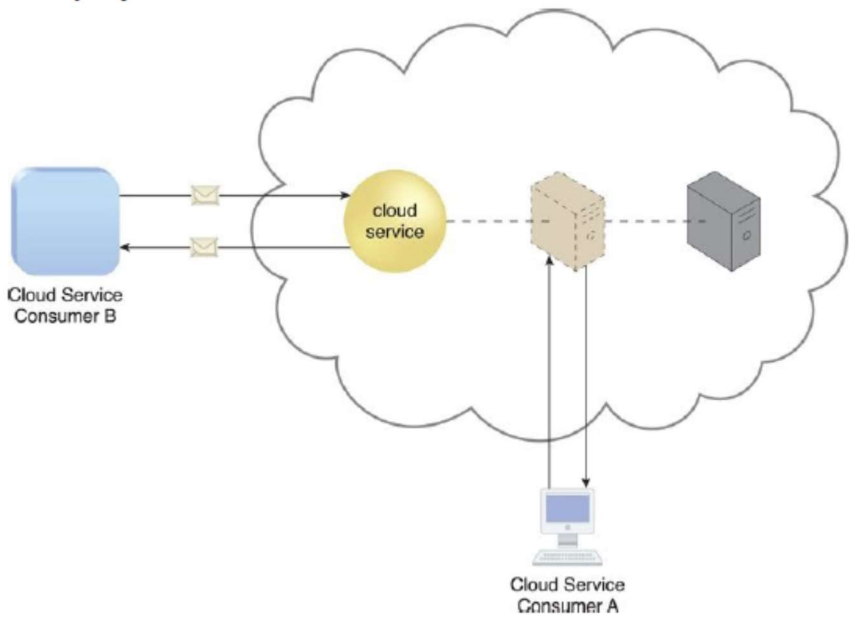


Figure 7.5. The first physical server hosts two virtual servers, while the second physical server hosts one virtual server.

As a commodity mechanism, the virtual server represents the most foundational building block of cloud environments. Each virtual server can host numerous IT resources, cloud-based solutions, and various other cloud computing mechanisms. The instantiation of virtual servers from image files is a resource allocation process that can be completed rapidly and on-demand.

Cloud consumers that install or lease virtual servers can customize their environments independently from other cloud consumers that may be using virtual servers hosted by the same underlying physical server. Figure below depicts a virtual server that hosts a cloud service being accessed by Cloud Service Consumer B, while Cloud Service Consumer A accesses the virtual server directly to perform an administration task.



Cloud Storage Device

The *cloud storage device* mechanism represents storage devices that are designed specifically for cloud-based provisioning. Instances of these devices can be virtualized, similar to how physical servers can spawn virtual server images. Cloud storage devices are commonly able to provide fixed-increment capacity allocation in support of the pay-per-use mechanism. Cloud storage devices can be exposed for remote access via cloud storage services.

A primary concern related to cloud storage is the security, integrity, and confidentiality of data, which becomes more prone to being compromised when entrusted to external cloud providers and other third parties. There can also be legal and regulatory implications that result from relocating data across geographical or national boundaries. Another issue applies specifically to the performance of large databases. LANs provide locally stored data with network reliability and latency levels that are superior to those of WANs.

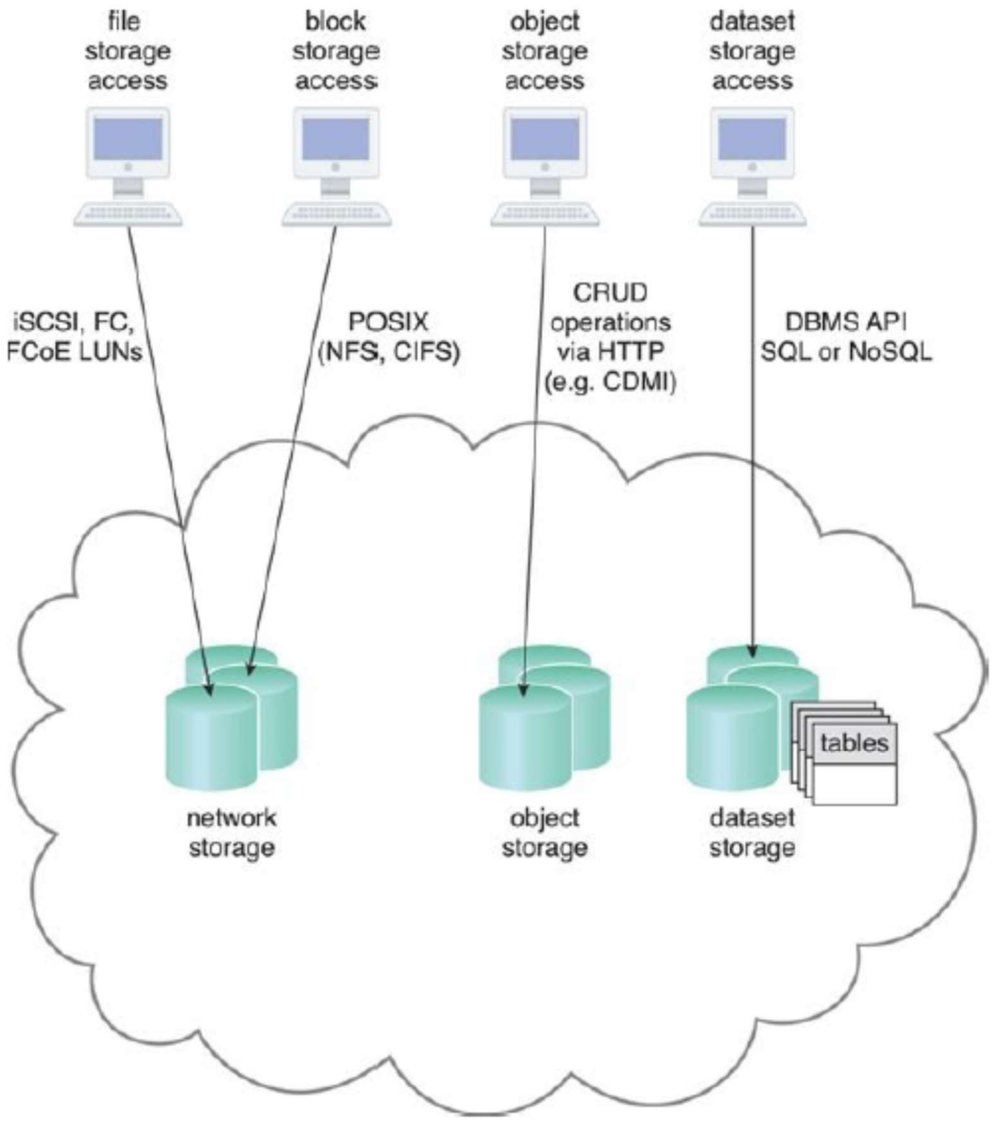
Cloud Storage Levels

Cloud storage device mechanisms provide common logical units of data storage, such as:

- *Files* - Collections of data are grouped into files that are located in folders.
- *Blocks* - The lowest level of storage and the closest to the hardware, a block is the smallest unit of data that is still individually accessible.

- *Datasets* - Sets of data are organized into a table-based, delimited, or record format.
- *Objects* - Data and its associated metadata are organized as Web-based resources.

Each of these data storage levels is commonly associated with a certain type of technical interface which corresponds to a particular type of cloud storage device and cloud storage service used to expose its API.



Network Storage Interfaces

Legacy network storage most commonly falls under the category of network storage interfaces. It includes storage devices in compliance with industry standard protocols, such as SCSI for storage blocks and the server message block (SMB), common Internet file system (CIFS), and network file system (NFS) for file and network storage. File storage entails storing individual data in separate files that can be different sizes and formats and organized into folders and subfolders. Original files are often replaced by the new files that are created when data has been modified.

When a cloud storage device mechanism is based on this type of interface, its data searching and extraction performance will tend to be suboptimal. Storage processing levels and thresholds for file allocation are usually determined by the file system itself. Block storage requires data to be in a fixed format (known as a *data block),* which is the smallest unit that can be stored and accessed and the storage format closest to hardware. Using either the logical unit number (LUN) or virtual volume block-level storage will typically have better performance than file-level storage.

Object Storage Interfaces

Various types of data can be referenced and stored as Web resources. This is referred to as object storage, which is based on technologies that can support a range of data and media types. Cloud Storage Device mechanisms that implement this interface can typically be accessed via REST or Web service-based cloud services using HTTP as the prime protocol. The Storage Networking Industry Association's Cloud Data Management Interface (SNIA's CDMI) supports the use of object storage interfaces.

Database Storage Interfaces

Cloud storage device mechanisms based on database storage interfaces typically support a query language in addition to basic storage operations. Storage management is carried out using a standard API or an administrative user- interface.

This classification of storage interface is divided into two main categories according to storage structure, as follows.

- ***Relational Data Storage***

  Traditionally, many on-premise IT environments store data using relational databases or relational database management systems (RDBMSs). Relational databases (or relational storage devices) rely on tables to organize similar data into rows and columns. Tables can have relationships with each other to give the data increased structure, to protect data integrity, and to avoid data redundancy (which is referred to as data normalization). Working with relational storage commonly involves the use of the industry standard Structured Query Language (SQL).

  A cloud storage device mechanism implemented using relational data storage could be based on any number of commercially available database products, such as IBM DB2, Oracle Database, Microsoft SQL Server, and MySQL.

  Challenges with cloud-based relational databases commonly pertain to scaling and performance. Scaling a relational cloud storage device vertically can be more complex and cost-ineffective than horizontal scaling. Databases with complex relationships and/or containing large volumes of data can be afflicted with higher processing overhead and latency, especially when accessed remotely via cloud services.

- ***Non-Relational Data Storage***

  Non-relational storage (also commonly referred to as *NoSQL* storage) moves away from the traditional relational database model in that it establishes a "looser" structure for stored data with less emphasis on defining relationships and realizing data normalization. The primary motivation for using nonrelational storage is to avoid the potential complexity and processing overhead that can be imposed by relational databases. Also, non-relational storage can be more horizontally scalable than relational storage.

  The trade-off with non-relational storage is that the data loses much of the native form and validation due to limited or primitive schemas or data models. Furthermore, non-relational repositories don't tend to support relational database functions, such as transactions or joins.

Normalized data exported into a non-relational storage repository will usually become denormalized, meaning that the size of the data will typically grow. An extent of normalization can be preserved, but usually not for complex relationships. Cloud providers often offer non-relational storage that provides scalability and availability of stored data over multiple server environments. However, many non-relational storage mechanisms are proprietary and therefore can severely limit data portability.

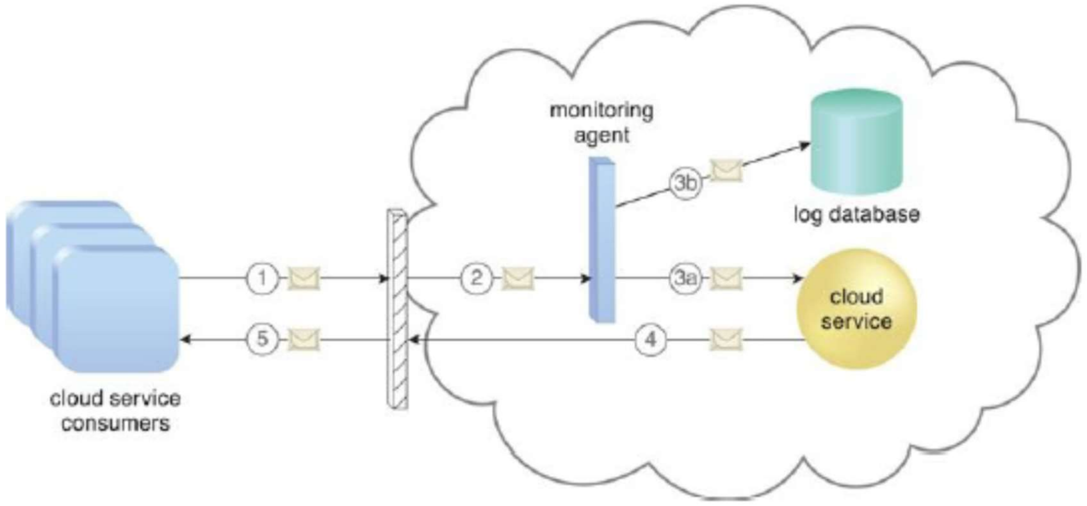
Cloud Usage Monitor

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. The upcoming sections describe three common agent- based implementation formats. Each can be designed to forward collected usage data to a log database for post-processing and reporting purposes.

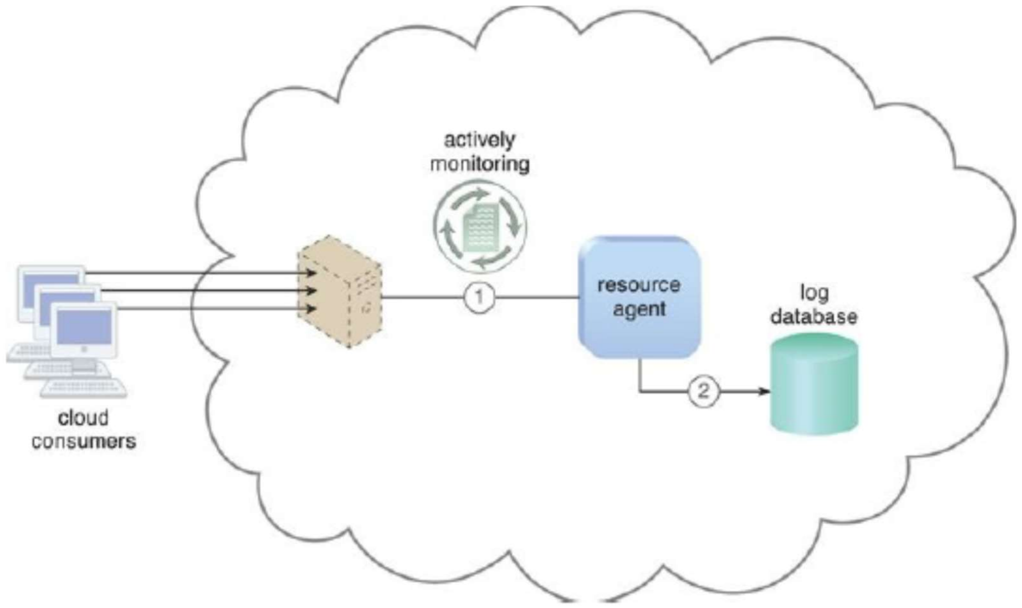
- **Monitoring Agent**

A *monitoring agent* is 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.



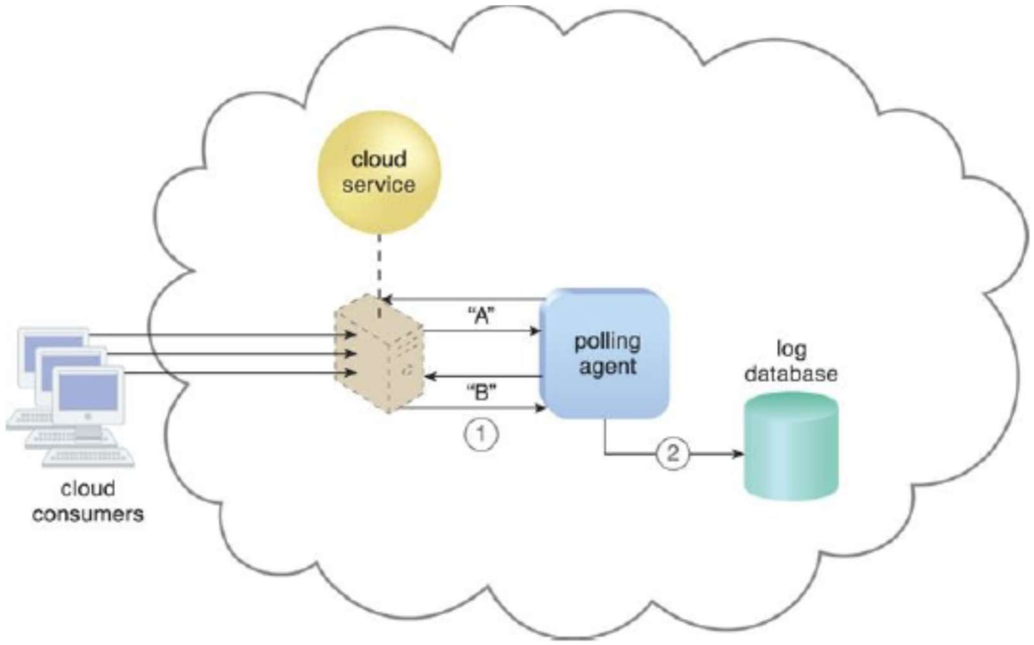
- **Resource Agent**

A *resource agent* is 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 pre-defined, observable events at the resource software level, such as initiating, suspending, resuming, and vertical scaling.



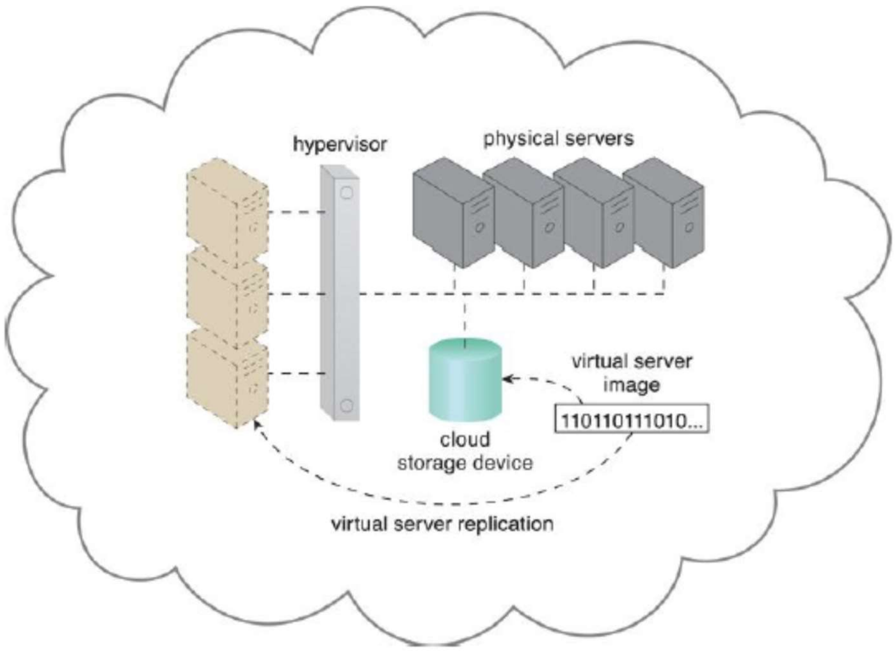
- **Polling Agent**

A *polling agent* is 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.



Resource Replication

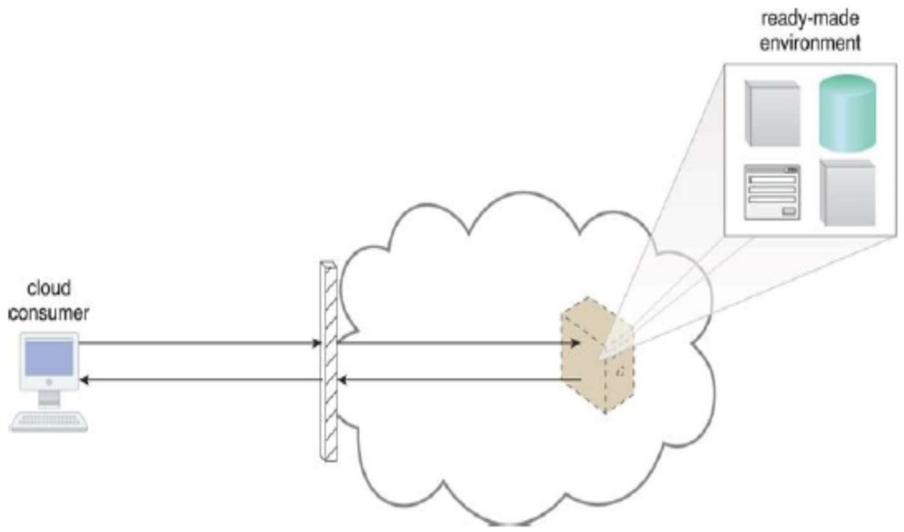
Defined as the creation of multiple instances of the same IT resource, replication is typically performed when an IT resource's availability and performance need to be enhanced. Virtualization technology is used to implement the *resource replication* mechanism to replicate cloud-based IT resources.



Ready-Made Environment

The *ready-made environment* mechanism is a defining component of the PaaS cloud delivery model that represents a pre-defined, cloud-based platform comprised of a set of already installed IT resources, ready to be used and customized by a cloud consumer. These environments are utilized by cloud consumers to remotely develop and deploy their own services and applications within a cloud. Typical ready-made environments include pre-installed IT resources, such as databases, middleware, development tools, and governance tools.

A ready-made environment is generally equipped with a complete software development kit (SDK) that provides cloud consumers with programmatic access to the development technologies that comprise their preferred programming stacks.



Middleware is available for multitenant platforms to support the development and deployment of Web applications. Some cloud providers offer runtime execution environments for cloud services that are based on different runtime performance and billing parameters. For example, a front-end instance of a cloud service can be configured to respond to time-sensitive requests more effectively than a back-end instance. The former variation will be billed at a different rate than the latter.