capabilities to serve multiple users. This consolidation model will reduce the waste of energy and carbon emissions, thus contributing to a greener IT on one end and increasing revenue on the other end.

1.1.2 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 1.2 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

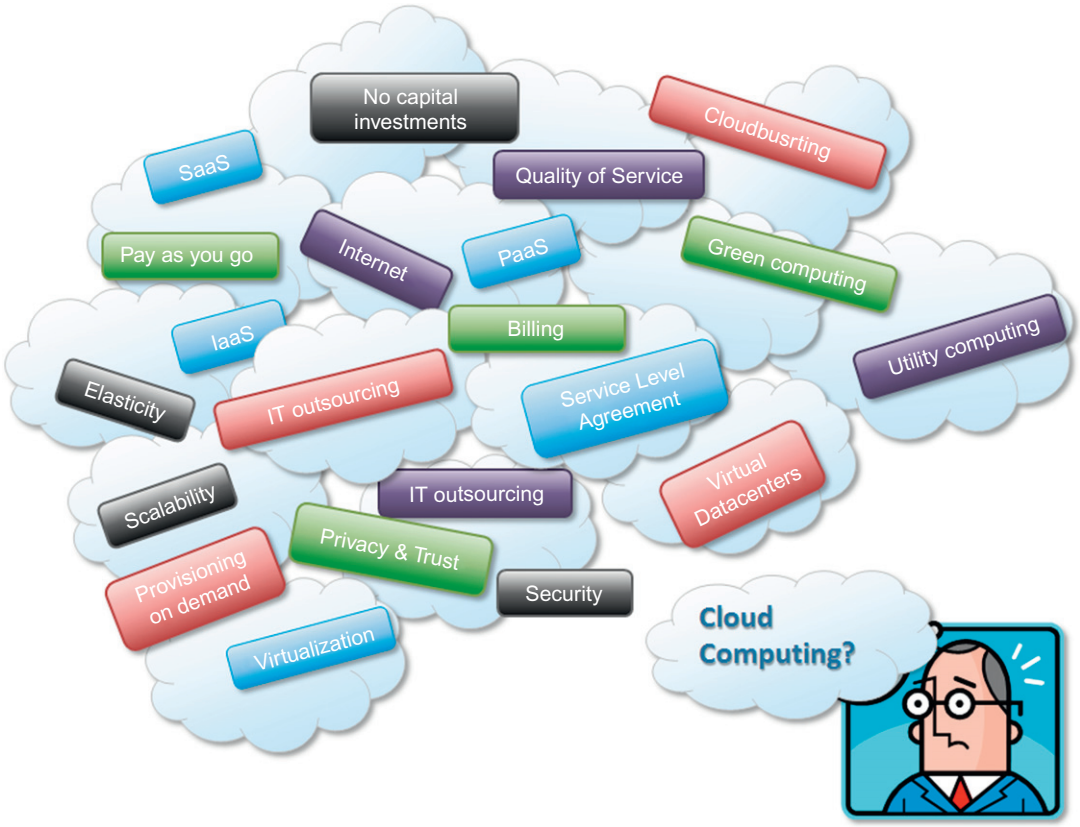

FIGURE 1.2

Cloud computing technologies, concepts, and ideas.

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. [28]:

> *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*,[2] 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. This helps us provide a different and more practical characterization of cloud computing. According to Reese [29], we can define three criteria to discriminate whether a service is delivered in the cloud computing style:

- The service is accessible via a Web browser (nonproprietary) or a Web services application programming interface (API).
- Zero capital expenditure is necessary to get started.
- You pay only for what you use as you use it.

Even though many cloud computing services are freely available for single users, enterprise-class services are delivered according a specific pricing scheme. In this case users subscribe to the service and establish with the service provider a service-level agreement (SLA) defining the

[2] *XaaS* is an acronym standing for *X-as-a-Service*, where the *X* letter can be replaced by one of a number of things: *S* for *software*, *P* for *platform*, *I* for *infrastructure*, *H* for *hardware*, *D* for *database*, and so on.

quality-of-service parameters under which the service is delivered. The utility-oriented nature of cloud computing is clearly expressed by Buyya et al. [30]:

> *A cloud is a type of parallel and distributed system consisting of a collection of interconnected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements established through negotiation between the service provider and consumers.*

1.1.3 A closer look

Cloud computing is helping enterprises, governments, public and private institutions, and research organizations shape more effective and demand-driven computing systems. Access to, as well as integration of, cloud computing resources and systems is now as easy as performing a credit card transaction over the Internet. Practical examples of such systems exist across all market segments:

- *Large enterprises can offload some of their activities to cloud-based systems.* Recently, the *New York Times* has converted its digital library of past editions into a Web-friendly format. This required a considerable amount of computing power for a short period of time. By renting Amazon EC2 and S3 Cloud resources, the *Times* performed this task in 36 hours and relinquished these resources, with no additional costs.
- *Small enterprises and start-ups can afford to translate their ideas into business results more quickly, without excessive up-front costs.* Animoto is a company that creates videos out of images, music, and video fragments submitted by users. The process involves a considerable amount of storage and backend processing required for producing the video, which is finally made available to the user. Animoto does not own a single server and bases its computing infrastructure entirely on Amazon Web Services, which are sized on demand according to the overall workload to be processed. Such workload can vary a lot and require instant scalability.[3] Up-front investment is clearly not an effective solution for many companies, and cloud computing systems become an appropriate alternative.
- *System developers can concentrate on the business logic rather than dealing with the complexity of infrastructure management and scalability.* Little Fluffy Toys is a company in London that has developed a widget providing users with information about nearby bicycle rental services. The company has managed to back the widget's computing needs on Google AppEngine and be on the market in only one week.
- *End users can have their documents accessible from everywhere and any device.* Apple iCloud is a service that allows users to have their documents stored in the Cloud and access them from any device users connect to it. This makes it possible to take a picture while traveling with a smartphone, go back home and edit the same picture on your laptop, and have it show as updated on your tablet computer. This process is completely transparent to the user, who does not have to set up cables and connect these devices with each other.

How is all of this made possible? The same concept of IT services on demand—whether computing power, storage, or runtime environments for applications—on a pay-as-you-go basis

[3] It has been reported that Animoto, in one single week, scaled from 70 to 8,500 servers because of user demand.

accommodates these four different scenarios. Cloud computing does not only contribute with the opportunity of easily accessing IT services on demand, it also introduces a new way of thinking about IT services and resources: as utilities. A bird's-eye view of a cloud computing environment is shown in Figure 1.3.

The three major models for deploying and accessing cloud computing environments are public clouds, private/enterprise clouds, and hybrid clouds (see Figure 1.4). *Public clouds* are the most common deployment models in which necessary IT infrastructure (e.g., virtualized datacenters) is established by a third-party service provider that makes it available to any consumer on a subscription basis. Such clouds are appealing to users because they allow users to quickly leverage compute, storage, and application services. In this environment, users' data and applications are deployed on cloud datacenters on the vendor's premises.

Large organizations that own massive computing infrastructures can still benefit from cloud computing by replicating the cloud IT service delivery model in-house. This idea has given birth to the concept of *private clouds* as opposed to public clouds. In 2010, for example, the U.S. federal government, one of the world's largest consumers of IT spending (around $76 billion on more than

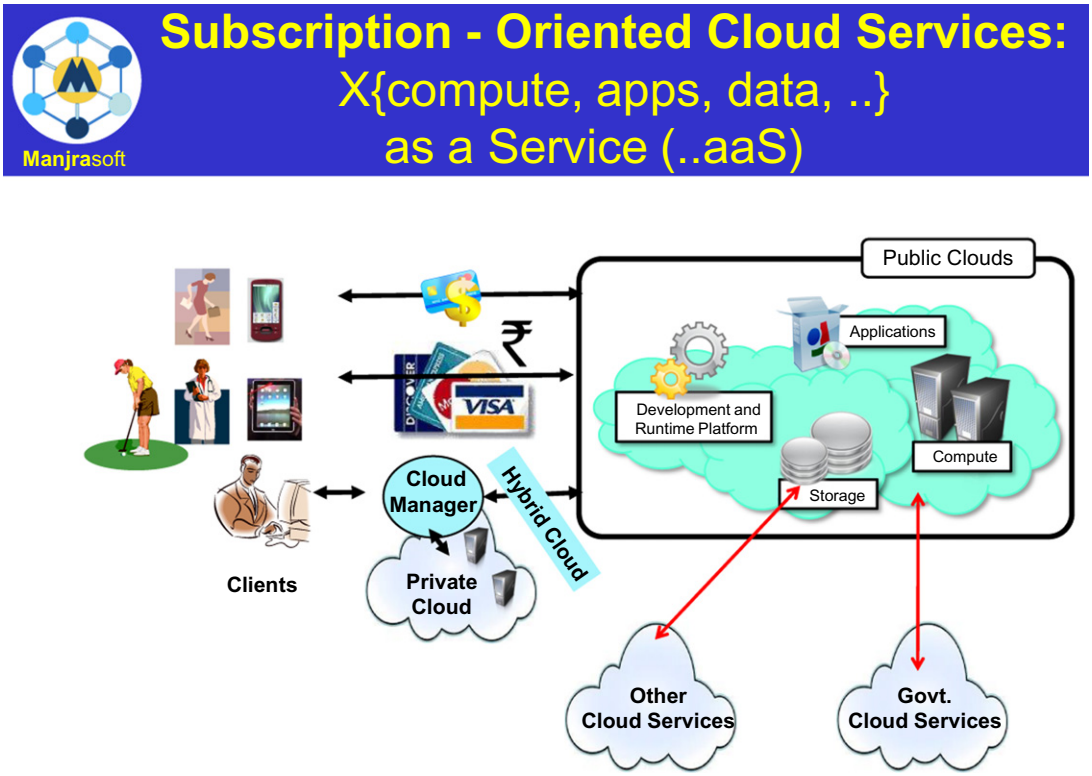

FIGURE 1.3

A bird's-eye view of cloud computing.

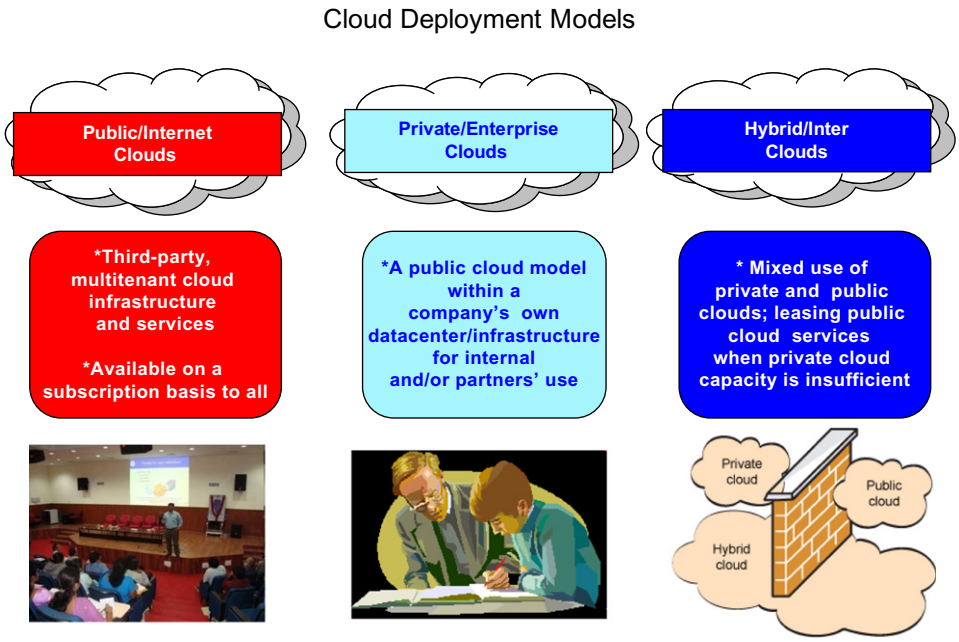

FIGURE 1.4

Major deployment models for cloud computing.

10,000 systems) started a cloud computing initiative aimed at providing government agencies with a more efficient use of their computing facilities. The use of cloud-based in-house solutions is also driven by the need to keep confidential information within an organization's premises. Institutions such as governments and banks that have high security, privacy, and regulatory concerns prefer to build and use their own private or enterprise clouds.

Whenever private cloud resources are unable to meet users' quality-of-service requirements, hybrid computing systems, partially composed of public cloud resources and privately owned infrastructures, are created to serve the organization's needs. These are often referred as *hybrid clouds*, which are becoming a common way for many stakeholders to start exploring the possibilities offered by cloud computing.

1.1.4 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 1.5, which provides an organic view of cloud computing. We refer to this diagram as the *Cloud Computing Reference Model*, and we will use it throughout the

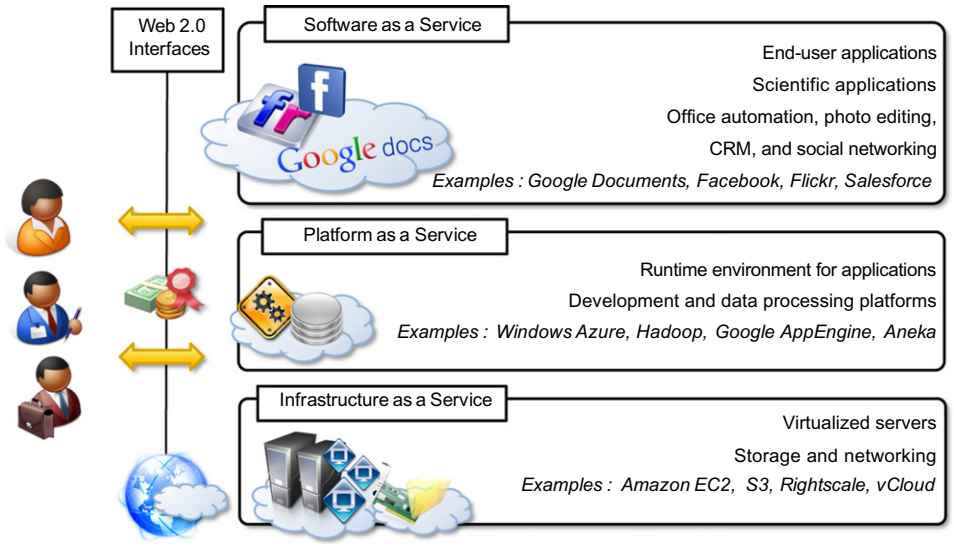

FIGURE 1.5

The Cloud Computing Reference Model.

book to explain the technologies and introduce the relevant research on this phenomenon. The model organizes the wide range of cloud computing services into a layered view that walks the computing stack from bottom to top.

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.

1.1.5 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.

Increased agility in defining and structuring software systems is another significant benefit of cloud computing. Since organizations rent IT services, they can more dynamically and flexibly compose their software systems, without being constrained by capital costs for IT assets. There is a reduced need for capacity planning, since cloud computing allows organizations to react to unplanned surges in demand quite rapidly. For example, organizations can add more servers to process workload spikes and dismiss them when they are no longer needed. Ease of scalability is another advantage. By leveraging the potentially huge capacity of cloud computing, organizations can extend their IT capability more easily. Scalability can be leveraged across the entire computing stack. Infrastructure providers offer simple methods to provision customized hardware and integrate it into existing systems. Platform-as-a-Service providers offer runtime environment and programming models that are designed to scale applications. Software-as-a-Service offerings can be elastically sized on demand without requiring users to provision hardware or to program application for scalability.

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.

1.1.6 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

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 [15]. 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 [25] 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*[6] 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.

1.3 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.

1.3.1 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

[6]The dot-com bubble was a phenomenon that started in the second half of the 1990s and reached its apex in 2000. During this period a large number of companies that based their business on online services and e-commerce started and quickly expanded without later being able to sustain their growth. As a result they suddenly went bankrupt, partly because their revenues were not enough to cover their expenses and partly because they never reached the required number of customers to sustain their enlarged business.

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.

1.3.2 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. These can be either completely transparent to developers or subject to fine control. Integration between cloud resources and existing system deployment is another element of concern.

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. As discussed before, the virtualization concept is more than 40 years old, but cloud computing introduces new challenges, especially in the management of virtual environments, whether they are abstractions of virtual hardware or a runtime environment. 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.

These are all considerations that influence the way we program applications and systems based on cloud computing technologies. Cloud computing essentially provides mechanisms to address surges in demand by replicating the required components of computing systems under stress (i.e., heavily loaded). Dynamism, scale, and volatility of such components are the main elements that should guide the design of such systems.

1.3.3 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.

1.3.3.1 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.

1.3.3.2 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.

1.3.3.3 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.

1.3.3.4 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.

1.3.3.5 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.

1.3.3.6 Manjrasoft Aneka

Manjrasoft Aneka [165] 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.

These platforms are key examples of technologies available for cloud computing. They mostly fall into the three major market segments identified in the reference model: *Infrastructure-as-a-Service*, *Platform-as-a-Service*, and *Software-as-a-Service*. In this book, we use Aneka as a reference platform for discussing practical implementations of distributed applications. We present different ways in which clouds can be leveraged by applications built using the various programming models and abstractions provided by Aneka.

SUMMARY

In this chapter, we discussed the vision and opportunities of cloud computing along with its characteristics and challenges. The cloud computing paradigm emerged as a result of the maturity and convergence of several of its supporting models and technologies, namely distributed computing, virtualization, Web 2.0, service orientation, and utility computing.

There is no single view on the cloud phenomenon. Throughout the book, we explore different definitions, interpretations, and implementations of this idea. The only element that is shared among all the different views of cloud computing is that cloud systems support dynamic provisioning of IT services (whether they are virtual infrastructure, runtime environments, or application services) and adopts a utility-based cost model to price these services. This concept is applied across the entire computing stack and enables the dynamic provisioning of IT infrastructure and runtime environments in the form of cloud-hosted platforms for the development of scalable applications and their services. This vision is what inspires the *Cloud Computing Reference Model*. This model identifies three major market segments (and service offerings) for cloud computing: *Infrastructure-as-a-Service (IaaS)*, *Platform-as-a-Service (PaaS)*, and *Software-as-a-Service (SaaS)*. These segments directly map the broad classifications of the different type of services offered by cloud computing.

The long-term vision of cloud computing is to fully realize the utility model that drives its service offering. It is envisioned that new technological developments and the increased familiarity with cloud computing delivery models will lead to the establishment of a global market for trading computing utilities. This area of study is called *market-oriented cloud computing*, where the term *market-oriented* further stresses the fact that cloud computing services are traded as utilities. The realization of this vision is still far from reality, but cloud computing has already brought economic, environmental, and technological benefits. By turning IT assets into utilities, it allows organizations to reduce operational costs and increase revenues. This and other advantages also have downsides that are diverse in nature. Security and legislation are two of the challenging aspects of cloud computing that are beyond the technical sphere.

From the perspective of software design and development, new challenges arise in engineering computing systems. Cloud computing offers a rich mixture of different technologies, and harnessing them is a challenging engineering task. Cloud computing introduces both new opportunities and new techniques and strategies for architecting software applications and systems. Some of the key elements that have to be taken into account are virtualization, scalability, dynamic provisioning, big datasets, and cost models. To provide a practical grasp of such concepts, we will use Aneka as a reference platform for illustrating cloud systems and application programming environments.

Review questions

1. What is the innovative characteristic of cloud computing?
2. Which are the technologies on which cloud computing relies?
3. Provide a brief characterization of a distributed system.
4. Define cloud computing and identify its core features.
5. What are the major distributed computing technologies that led to cloud computing?
6. What is virtualization?
7. What is the major revolution introduced by Web 2.0?
8. Give some examples of Web 2.0 applications.
9. Describe the main characteristics of a service orientation.
10. What is utility computing?
11. Describe the vision introduced by cloud computing.
12. Briefly summarize the Cloud Computing Reference Model.
13. What is the major advantage of cloud computing?
14. Briefly summarize the challenges still open in cloud computing.
15. How is cloud development different from traditional software development?

CHAPTER

Principles of Parallel and Distributed Computing

2

Cloud computing is a new technological trend that supports better utilization of IT infrastructures, services, and applications. It adopts a service delivery model based on a pay-per-use approach, in which users do not own infrastructure, platform, or applications but use them for the time they need them. These IT assets are owned and maintained by service providers who make them accessible through the Internet.

This chapter presents the fundamental principles of parallel and distributed computing and discusses models and conceptual frameworks that serve as foundations for building cloud computing systems and applications.

2.1 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 (see Figure 2.1). 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 mainstream. Moreover, every aspect of this era underwent a three-phase process: *research and development (R&D), commercialization*, and *commoditization*.

2.2 Parallel vs. distributed computing

The terms *parallel computing* and *distributed computing* are often used interchangeably, even though they mean slightly 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.

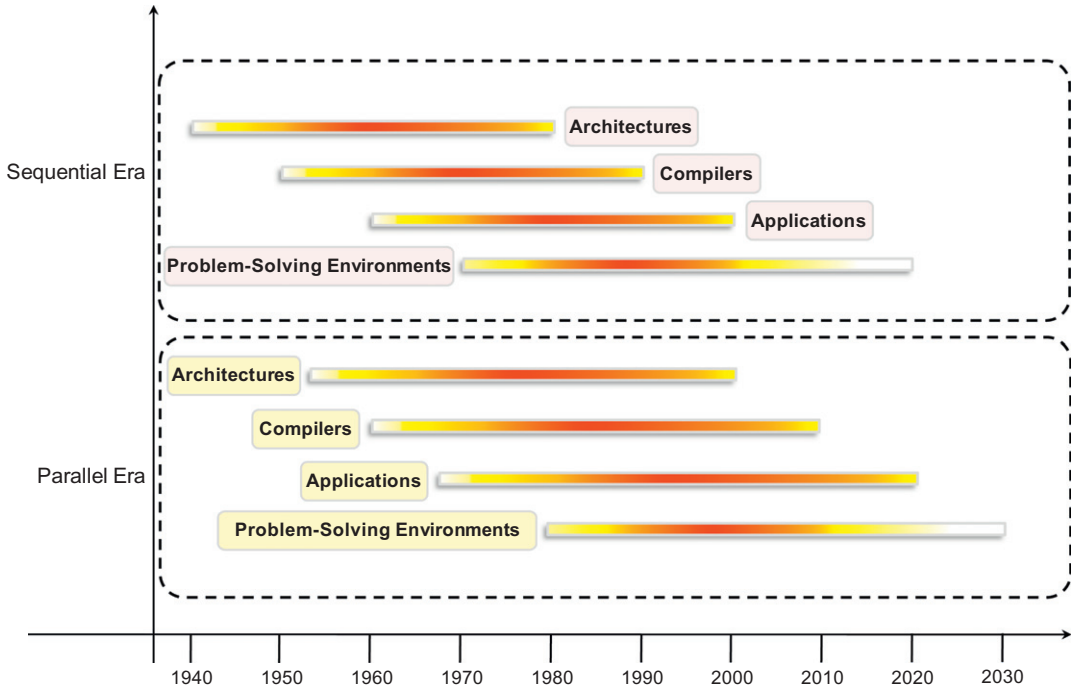

FIGURE 2.1

Eras of computing, 1940s–2030s.

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. Over time, 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.

2.3 Elements of parallel computing

It is now clear that silicon-based processor chips are reaching their physical limits. Processing speed is constrained by the speed of light, and the density of transistors packaged in a processor is constrained by thermodynamic limitations. A viable solution to overcome this limitation is to connect multiple processors working in coordination with each other to solve "Grand Challenge" problems. The first steps in this direction led to the development of parallel computing, which encompasses techniques, architectures, and systems for performing multiple activities in parallel. As we already discussed, the term *parallel computing* has blurred its edges with the term *distributed computing* and is often used in place of the latter term. In this section, we refer to its proper characterization, which involves the introduction of parallelism within a single computer by coordinating the activity of multiple processors together.

2.3.1 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.

2.3.2 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

2.3.2.1 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 2.2). 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.

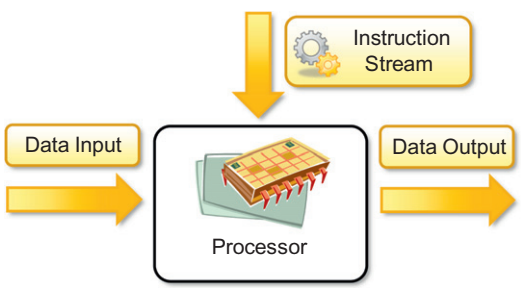

FIGURE 2.2

Single-instruction, single-data (SISD) architecture.

2.3.2.2 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 Figure 2.3). 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

$$\mathrm{Ci} = \mathrm{Ai} * \mathrm{Bi}$$

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*.

2.3.2.3 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 Figure 2.4). For instance, statements such as

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

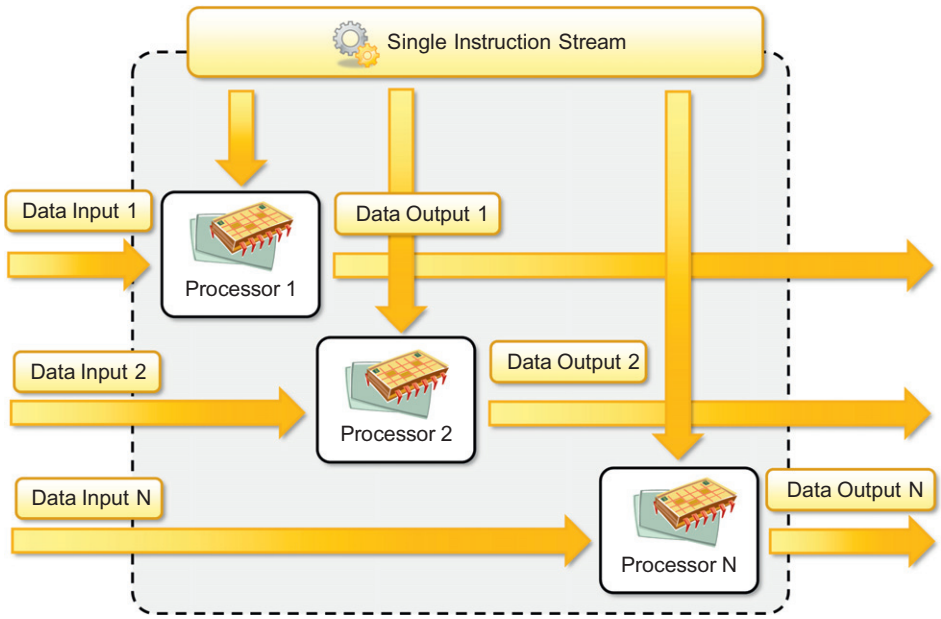

FIGURE 2.3

Single-instruction, multiple-data (SIMD) architecture.

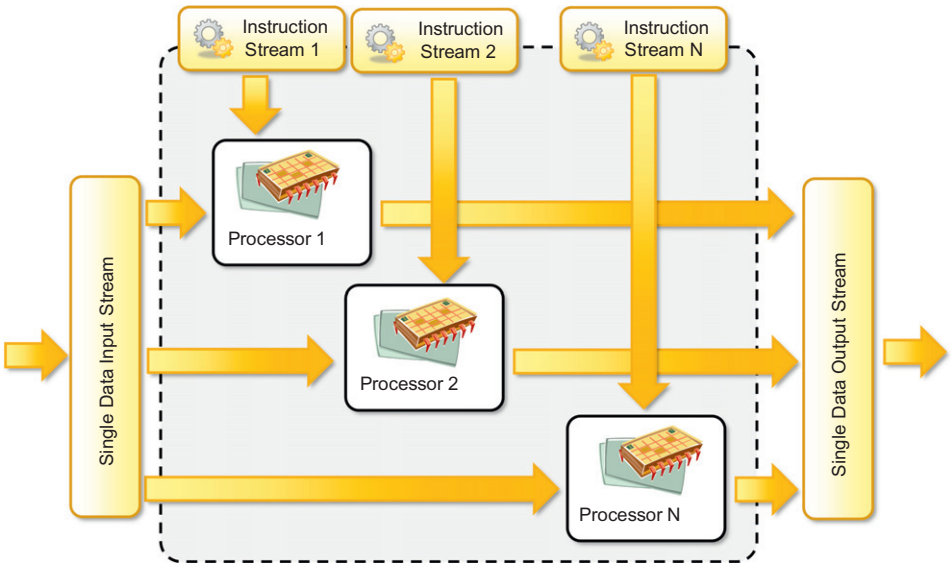

FIGURE 2.4

Multiple-instruction, single-data (MISD) architecture.

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.

2.3.2.4 Multiple-instruction, multiple-data (MIMD) systems

An MIMD computing system is a multiprocessor machine capable of executing multiple instructions on multiple data sets (see Figure 2.5). 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).

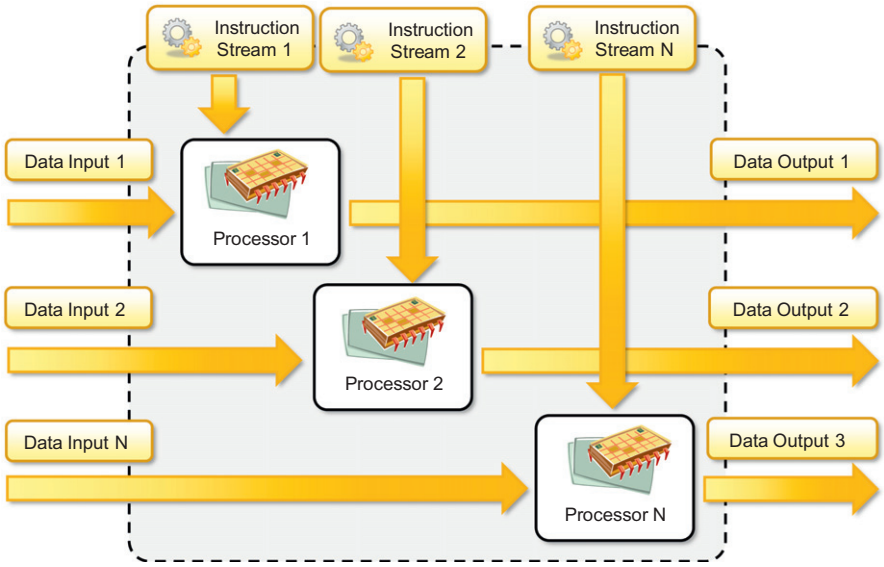

FIGURE 2.5

Multiple-instructions, multiple-data (MIMD) architecture.

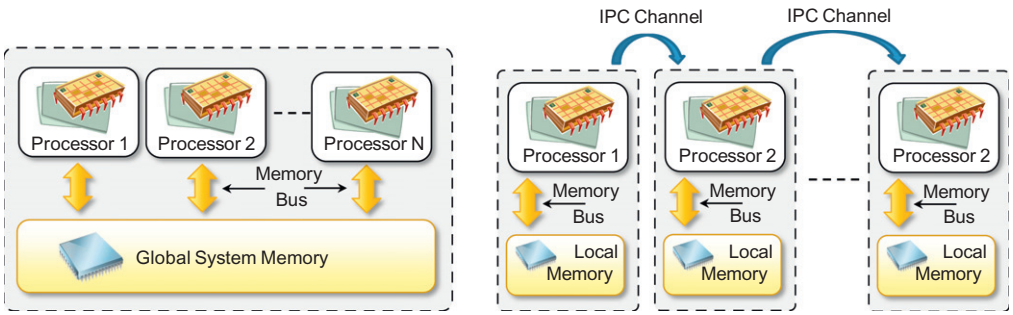

FIGURE 2.6

Shared (left) and distributed (right) memory MIMD architecture.

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.

The shared-memory MIMD architecture is easier to program but is less tolerant to failures and harder to extend with respect to the distributed memory MIMD model. Failures in a shared-memory MIMD affect the entire system, whereas this is not the case of the distributed model, in which each of the PEs can be easily isolated. Moreover, shared memory MIMD architectures are less likely to scale because the addition of more PEs leads to memory contention. This is a situation that does not happen in the case of distributed memory, in which each PE has its own memory. As a result, distributed memory MIMD architectures are most popular today.

2.3.3 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.

2.3.4 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 2.1 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 2.7, parallelism within an application can be detected at several levels:

- Large grain (or task level)
- Medium grain (or control level)

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

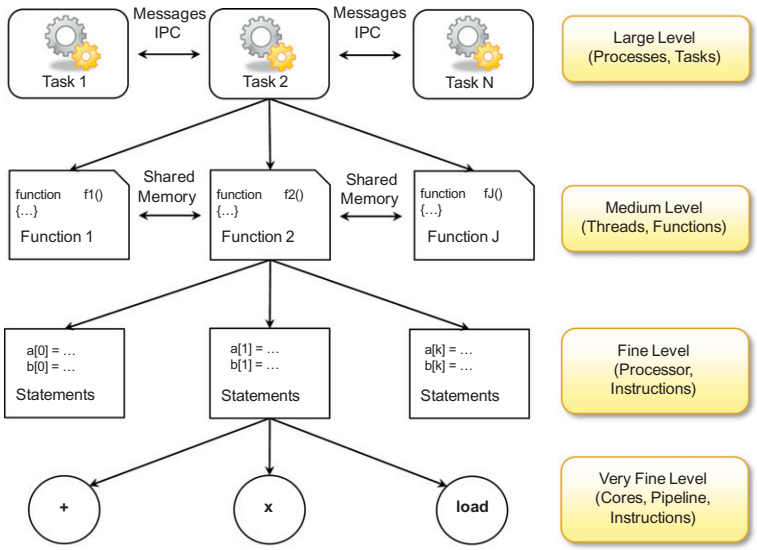

FIGURE 2.7

Levels of parallelism in an application.

- Fine grain (data level)
- Very fine grain (multiple-instruction issue)

In this book, we consider parallelism and distribution at the top two levels, which involve the distribution of the computation among multiple threads or processes.

2.3.5 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.

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 (Figure 2.8).
- 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 2.9.

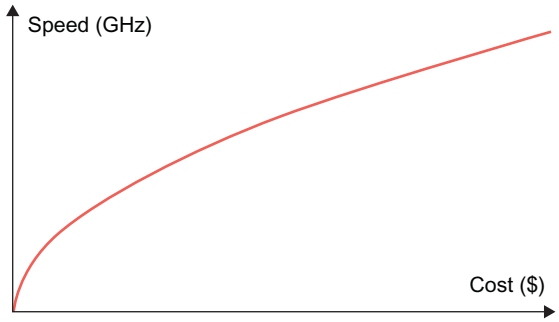

FIGURE 2.8

Cost versus speed.

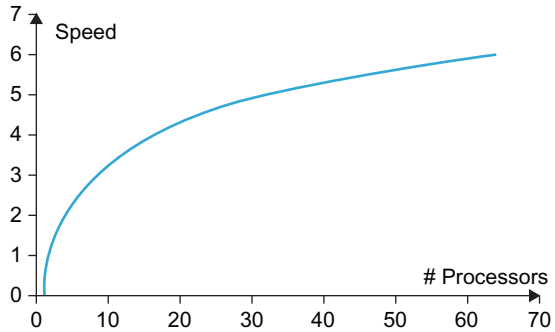

FIGURE 2.9

Number processors versus speed.

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.

2.4 Elements of distributed computing

In the previous section, we discussed techniques and architectures that allow introduction of parallelism within a single machine or system and how parallelism operates at different levels of the computing stack. In this section, we extend these concepts and explore how multiple activities can be performed by leveraging systems composed of multiple heterogeneous machines and systems. We discuss what is generally referred to as *distributed computing* and more precisely introduce the most common guidelines and patterns for implementing distributed computing systems from the perspective of the software designer.

2.4.1 General concepts and definitions

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 (Coulouris et al. [2]):

> *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.

2.4.2 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 2.10 provides an overview of the different layers that are involved in providing the services of a distributed system.

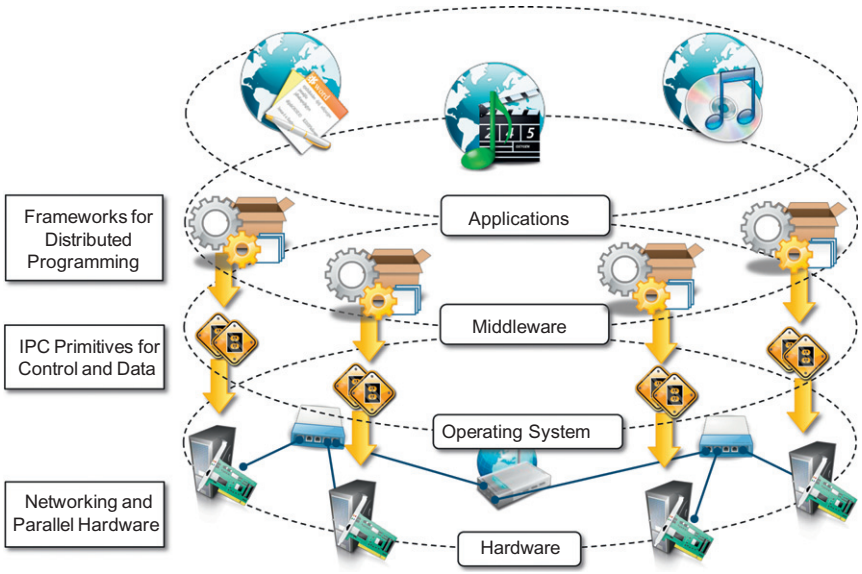

FIGURE 2.10

A layered view of a distributed system.

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, which we will discuss in Chapters 5–7 of this book. 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

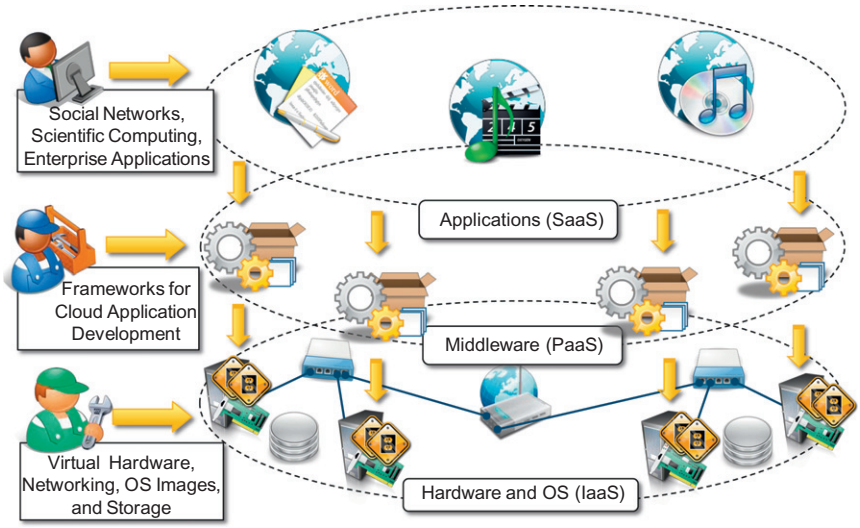

FIGURE 2.11

A cloud computing distributed system.

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. Figure 2.11 shows an example of how the general reference architecture of a distributed system is contextualized in the case of a cloud computing system.

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.

2.4.3 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 [104] 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 [105].*

Design patterns [106] 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.

2.4.3.1 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.

2.4.3.2 Software architectural styles

Software architectural styles are based on the logical arrangement of software components. They are helpful because they provide an intuitive view of the whole system, despite its physical deployment. They also identify the main abstractions that are used to shape the components of the system and the expected interaction patterns between them. According to Garlan and Shaw [105], architectural styles are classified as shown in Table 2.2.

These models constitute the foundations on top of which distributed systems are designed from a logical point of view, and they are discussed in the following sections.

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

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

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.

Within this reference scenario, knowledge sources, which represent the intelligent agents sharing the blackboard, react opportunistically to changes in the knowledge base, almost in the same way that a group of specialists brainstorm in a room in front of a blackboard. Blackboard models have become popular and widely used for artificial intelligent applications in which the blackboard maintains the knowledge about a domain in the form of assertions and rules, which are entered by domain experts. These operate through a control shell that controls the problem-solving activity of the system. Particular and successful applications of this model can be found in the domains of speech recognition and signal processing.

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. Another example are the Unix shell pipes (i.e., *cat <filename>| grep<pattern>| wc −l*), where the filters are the single shell programs composed together and the connections are their input and output streams that are chained together. Applications of this architecture can also be found in the compiler design (e.g., the lex/yacc model is based on a pipe of the following phases: *scanning | parsing | semantic analysis | code generation*), image and signal processing, and voice and video streaming.

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 2.3.

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

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

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

[1]A *programmable logic controller* (PLC) is a digital computer that is used for automation or electromechanical processes. Differently from general-purpose computers, PLCs are designed to manage multiple input lines and produce several outputs. In particular, their physical design makes them robust to more extreme environmental conditions or shocks, thus making them fit for use in factory environments. PLCs are an example of a hard real-time system because they are expected to produce the output within a given time interval since the reception of the input.

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 applications 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. Layers can be replaced as long as they are compliant with the expected protocols and interfaces, thus making the system flexible. The main disadvantage is constituted by the lack of extensibility, since it is not possible to add layers without changing the protocols and the interfaces between layers.[2] This also makes it complex to add operations. 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.[3] 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.[4] The main

[2]The only option given is to partition a layer into sublayers so that the external interfaces remain the same, but the internal architecture can be reorganized into different layers that can define different abstraction levels. From the point of view of the adjacent layer, the new reorganized layer still appears as a single block.

[3]The terms *client/server* and *peer-to-peer* will be further discussed in the next section.

[4]The *Observer* pattern [106] is a fundamental element of software designs, whereas programming languages such as C#, VB.NET, and other languages implemented for the *Common Language Infrastructure* [53] expose the *event* language constructs to model implicit invocation patterns.

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.

In this section, we reviewed the most popular software architectural styles that can be utilized as a reference for modeling the logical arrangement of components in a system. They are a subset of all the architectural styles; other styles can be found in [105].

2.4.3.3 System architectural styles

System architectural styles cover the physical organization of components and processes over a distributed infrastructure. They provide a set of reference models for the deployment of such systems and help engineers not only have a common vocabulary in describing the physical layout of systems but also quickly identify the major advantages and drawbacks of a given deployment and whether it is applicable for a specific class of applications. 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 Figure 2.12, 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.

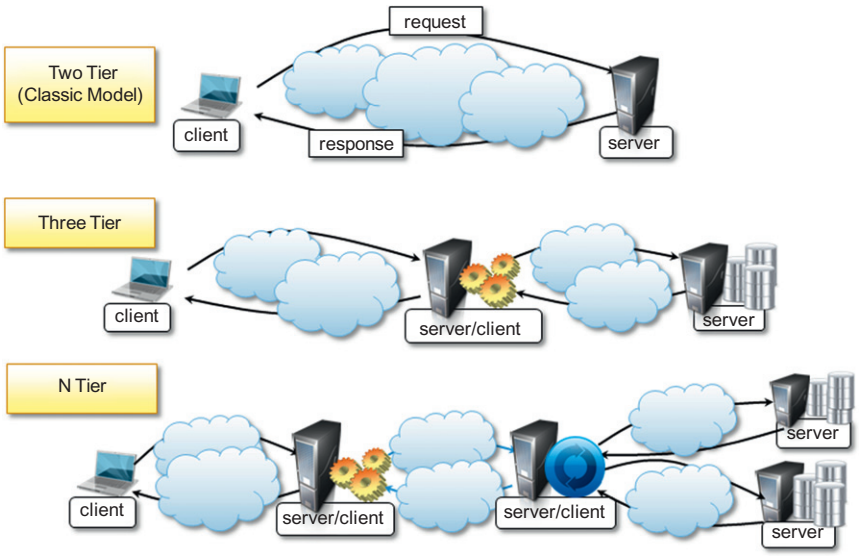

FIGURE 2.12

Client/server architectural styles.

- *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, depicted in Figure 2.13, 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.

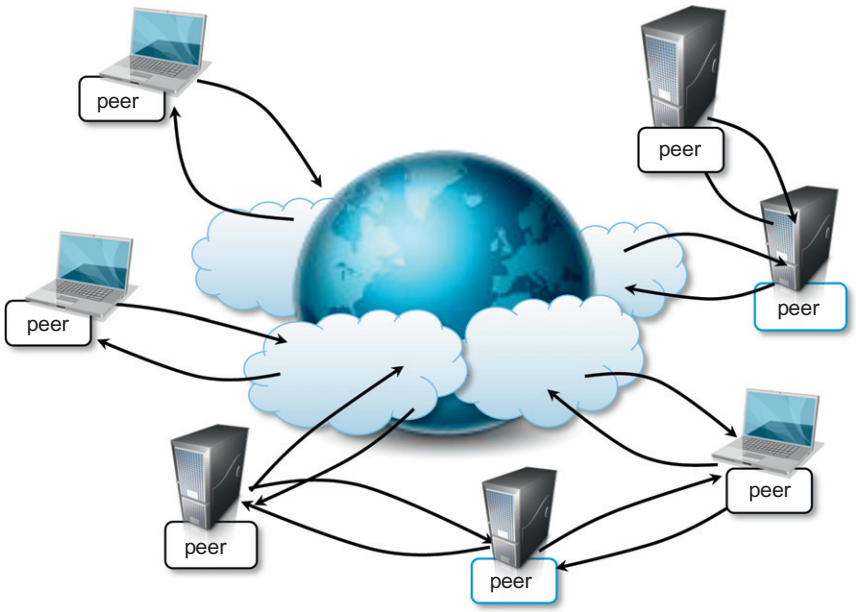

FIGURE 2.13

Peer-to-peer architectural style.

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.

2.4.4 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.

2.4.4.1 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. [2] 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)*.

It is important to observe that the concept of a message is a fundamental abstraction of IPC, and it is used either explicitly or implicitly. Messages' principal use—in any of the cases discussed—is to define interaction protocols among distributed components for coordinating their activity and exchanging data.

2.4.4.2 Models for message-based communication

We have seen how message-based communication constitutes a fundamental block for several distributed programming paradigms. Another 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.

The models presented here constitute a reference for structuring the communication among components in a distributed system. It is very uncommon that one single mode satisfies all the communication needs within a system. More likely, a composition of modes or their conjunct use in order to design and implement different aspects is the common case.

2.5 Technologies for distributed computing

In this section, we introduce relevant technologies that provide concrete implementations of interaction models, which mostly rely on message-based communication. They are remote procedure call (RPC), distributed object frameworks, and service-oriented computing.

2.5.1 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. The concept of RPC has been discussed since 1976 and completely formalized by Nelson [111] and Birrell [112] in the early 1980s. From there on, it has not changed in its major components. Even though it is a quite old technology, RPC is still used today as a fundamental component for IPC in more complex systems.

Figure 2.14 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

techniques for IPC: remote procedure calls, distributed objects, and services. We reviewed the reference models that are used to organize the communication within the components of a distributed system and presented the major features of each of the abstractions.

Cloud computing leverages these models, abstractions, and technologies and provides a more efficient way to design and use distributed systems by making entire systems or components available on demand.

Review questions

1. What is the difference between parallel and distributed computing?
2. Identify the reasons that parallel processing constitutes an interesting option for computing.
3. What is an SIMD architecture?
4. List the major categories of parallel computing systems.
5. Describe the different levels of parallelism that can be obtained in a computing system.
6. What is a distributed system? What are the components that characterize it?
7. What is an architectural style, and what is its role in the context of a distributed system?
8. List the most important software architectural styles.
9. What are the fundamental system architectural styles?
10. What is the most relevant abstraction for interprocess communication in a distributed system?
11. Discuss the most important model for message-based communication.
12. Discuss RPC and how it enables interprocess communication.
13. What is the difference between distributed objects and RPC?
14. What are object activation and lifetime? How do they affect the consistency of state within a distributed system?
15. What are the most relevant technologies for distributed objects programming?
16. Discuss CORBA.
17. What is service-oriented computing?
18. What is market-oriented cloud computing?
19. What is SOA?
20. Discuss the most relevant technologies supporting service computing.