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.

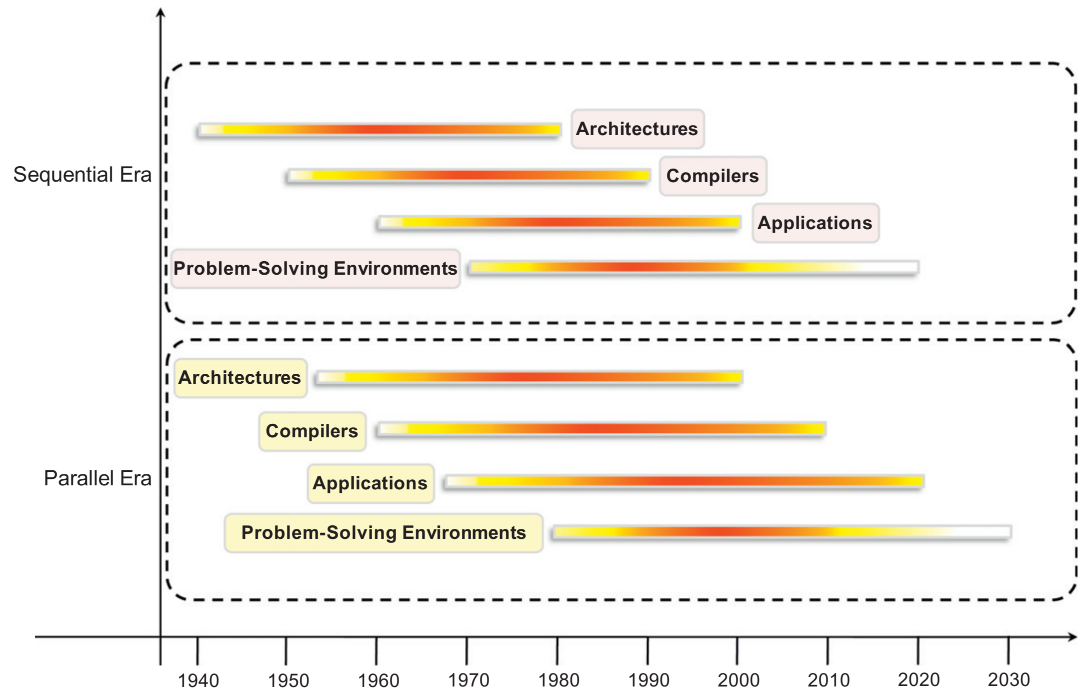


FIGURE 2.1

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.

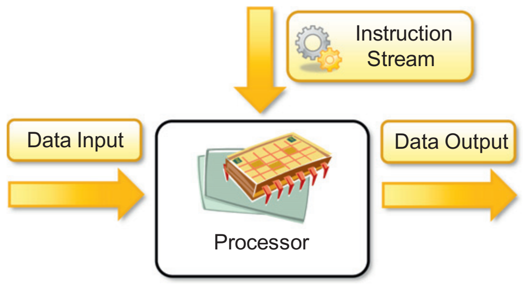


FIGURE 2.2

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

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

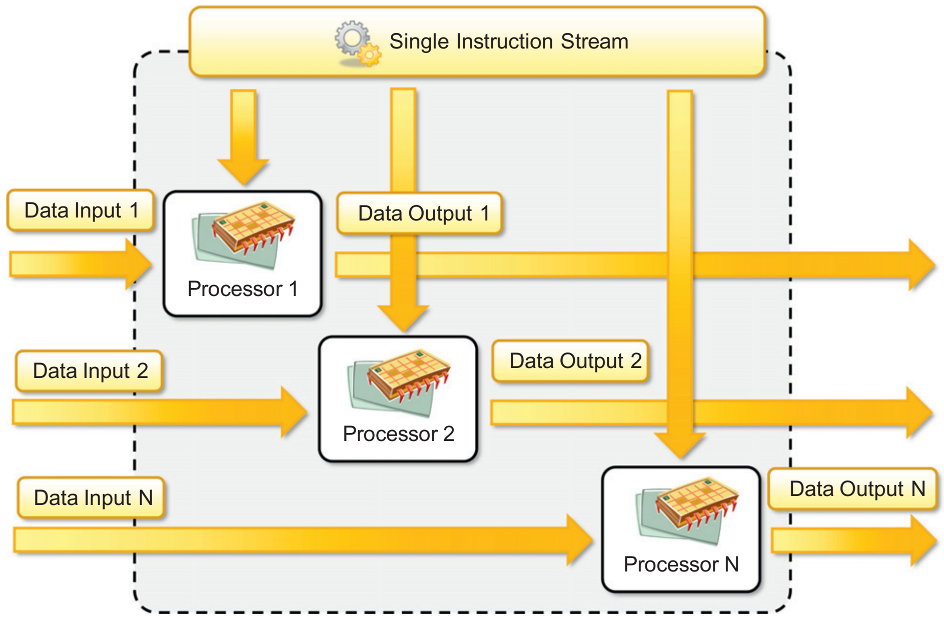


FIGURE 2.3

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

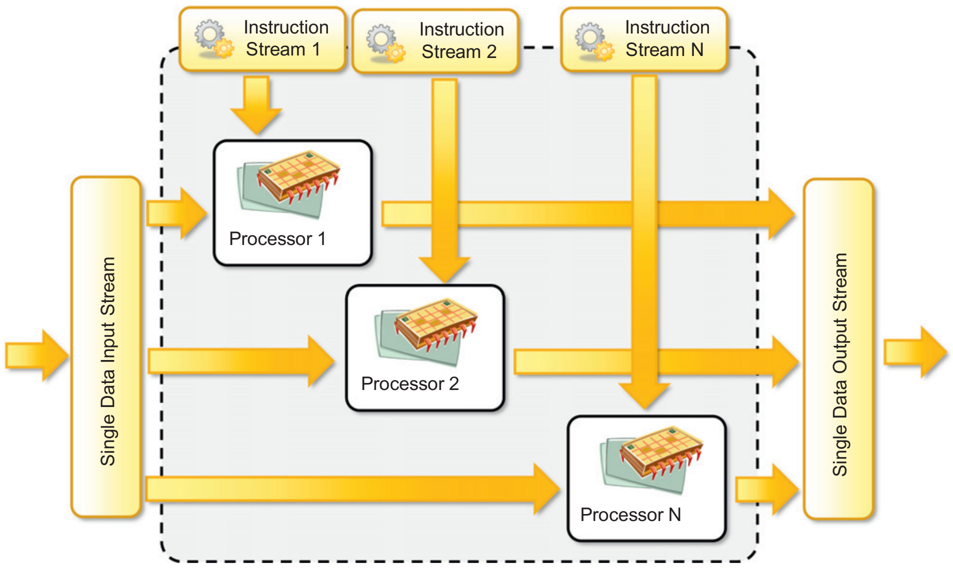


FIGURE 2.4

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).

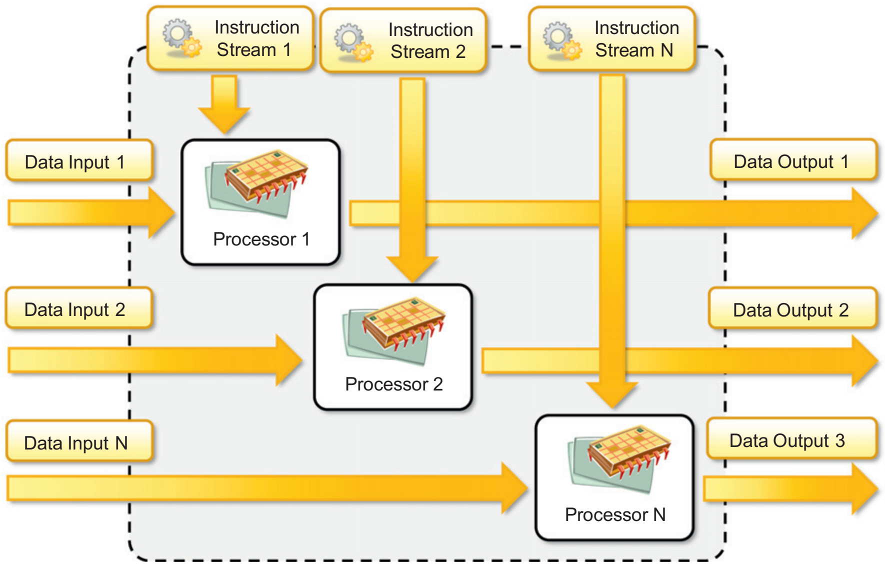


FIGURE 2.5

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

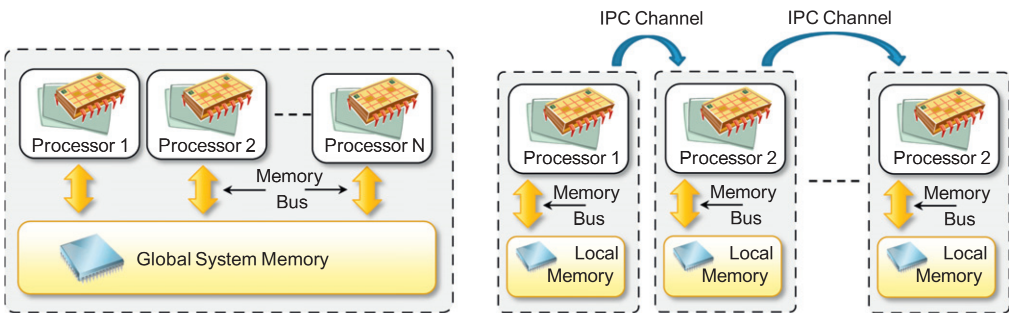


FIGURE 2.6

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

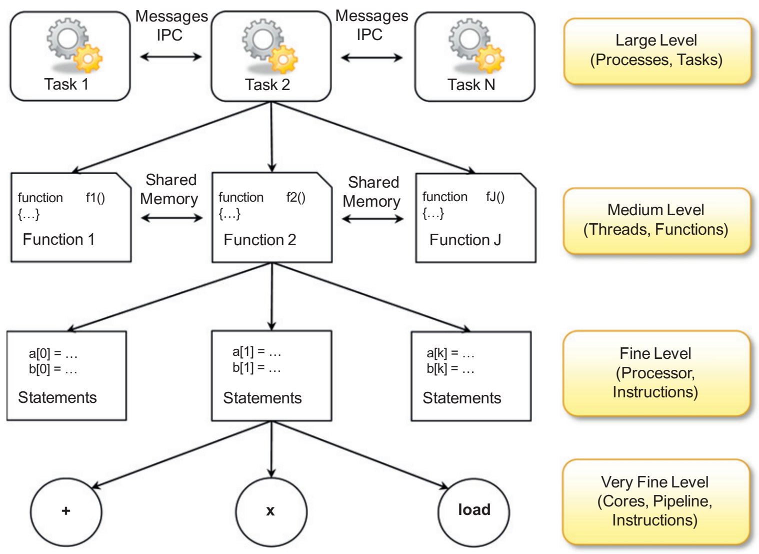


FIGURE 2.7

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.

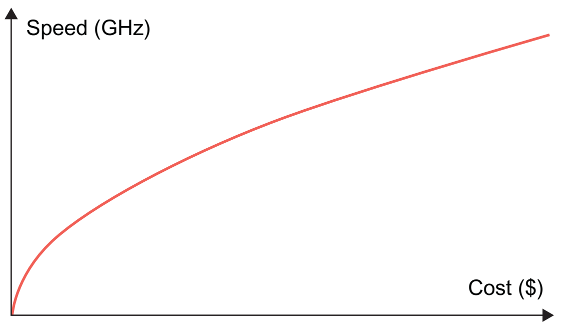


FIGURE 2.8

Cost versus speed.

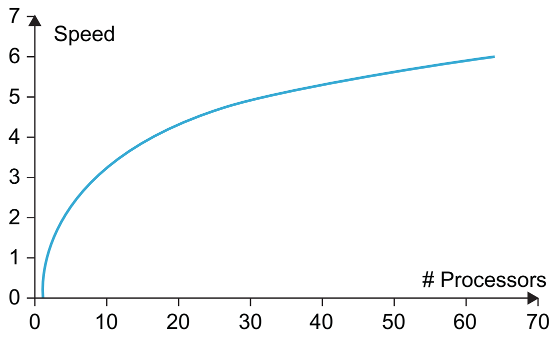


FIGURE 2.9

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.

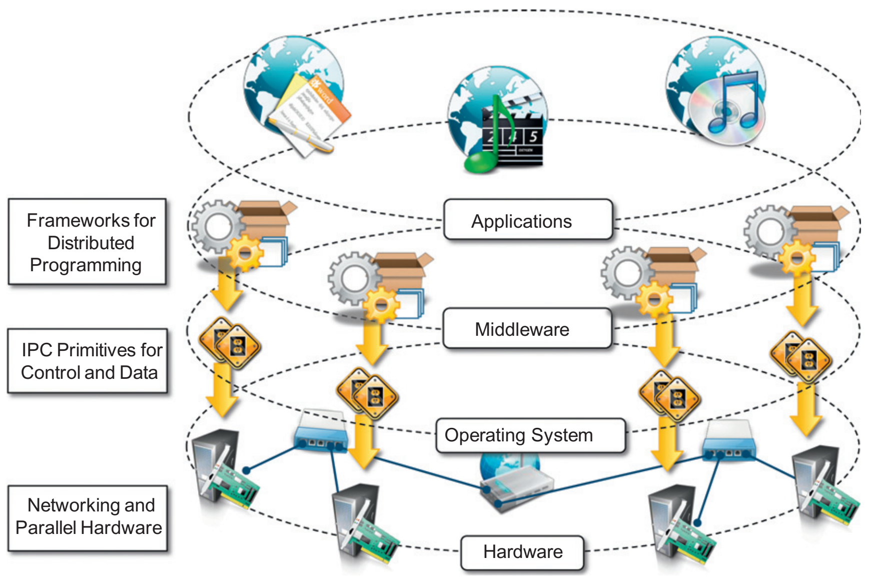

Frameworks for Distributed Programming
IPC Primitives for Control and Data
Networking and Parallel Hardware
Applications
Middleware
Operating System
Hardware

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

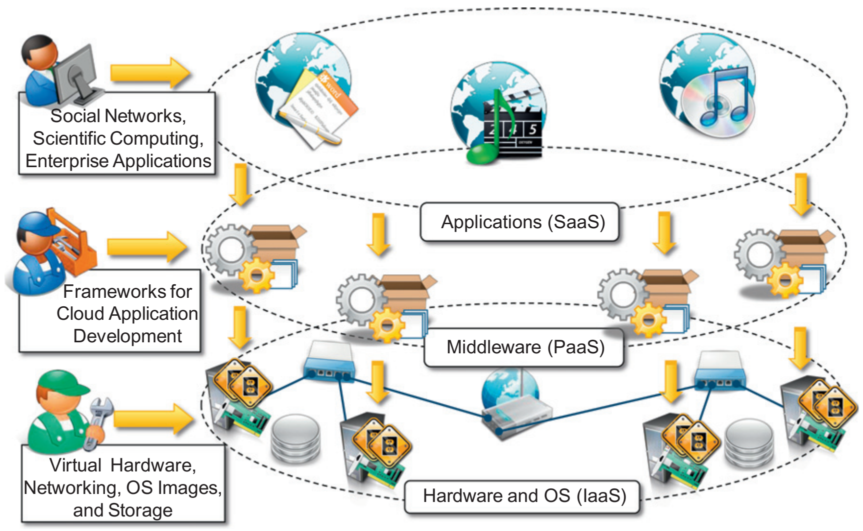

Social Networks, Scientific Computing, Enterprise Applications
Frameworks for Cloud Application Development
Virtual Hardware, Networking, OS Images, and Storage
Applications (SaaS)
Middleware (PaaS)
Hardware and OS (IaaS)

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.

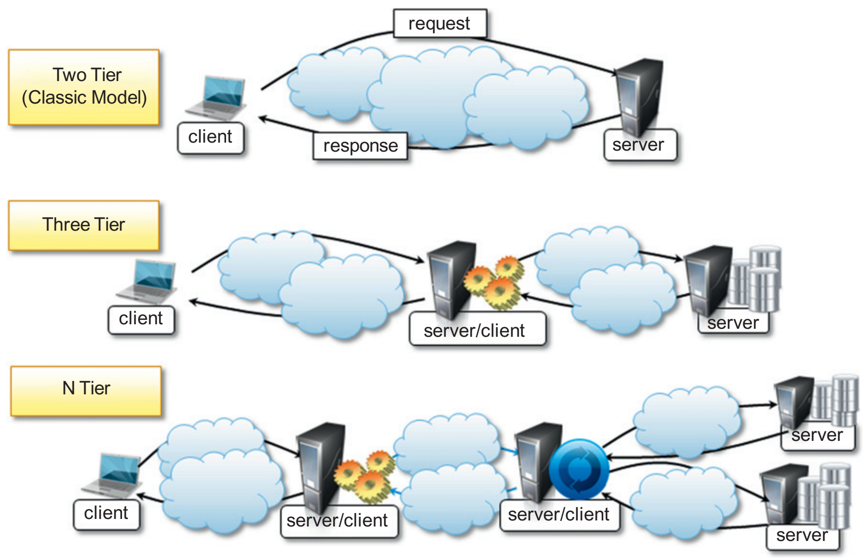


FIGURE 2.12

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.

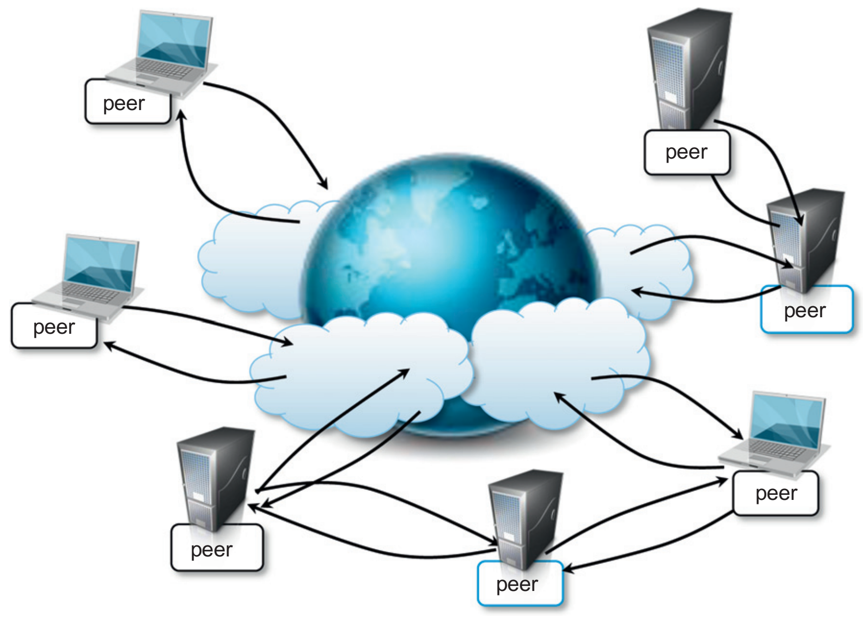


FIGURE 2.13

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

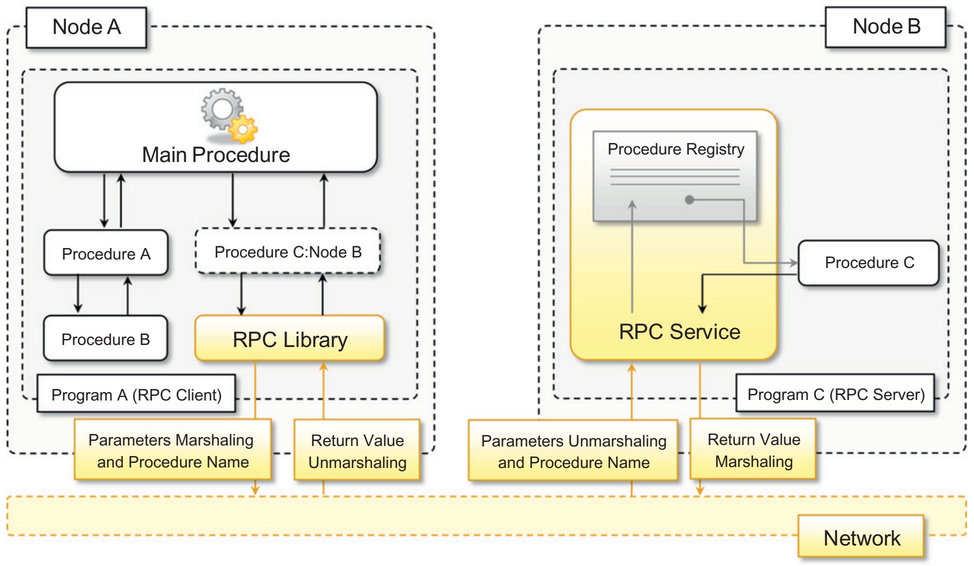


FIGURE 2.14

The RPC reference model.

can be remotely invoked and listens for requests from clients that specify which procedure to invoke, together with the values of the parameters required by the procedure. RPC maintains the synchronous pattern that is natural in IPC and function calls. Therefore, the calling process thread remains blocked until the procedure on the server process has completed its execution and the result (if any) is returned to the client.

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

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

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

RPC has been a dominant technology for IPC for quite a long time, and several programming languages and environments support this interaction pattern in the form of libraries and additional packages. For instance, RPyC is an RPC implementation for Python. There also exist platform-independent solutions such as XML-RPC and JSON-RPC, which provide RPC facilities over XML and JSON, respectively. Thrift [113] is the framework developed at Facebook for enabling a transparent cross-language RPC model. Currently, the term RPC implementations encompass a variety of solutions including frameworks such distributed object programming (CORBA, DCOM, Java RMI, and .NET Remoting) and Web services that evolved from the original RPC concept. We discuss the peculiarity of these approaches in the following sections.

2.5.2 Distributed object frameworks

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

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

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

Distributed object frameworks give the illusion of interaction with a local instance while invoking remote methods. This is done by a mechanism called a *proxy skeleton*. Figure 2.15 gives an overview of how this infrastructure works. Proxy and skeleton always constitute a pair: the server process maintains the skeleton component, which is in charge of executing the methods that are remotely invoked, while the client maintains the proxy component, allowing its hosting environment to remotely invoke methods through the proxy interface. The transparency of remote method invocation is achieved using one of the fundamental properties of object-oriented programming: inheritance and subclassing. Both the proxy and the active remote object expose the same interface, defining the set of methods that can be remotely called. On the client side, a runtime object subclassing the type published by the server is generated. This object translates the local method invocation into an RPC call

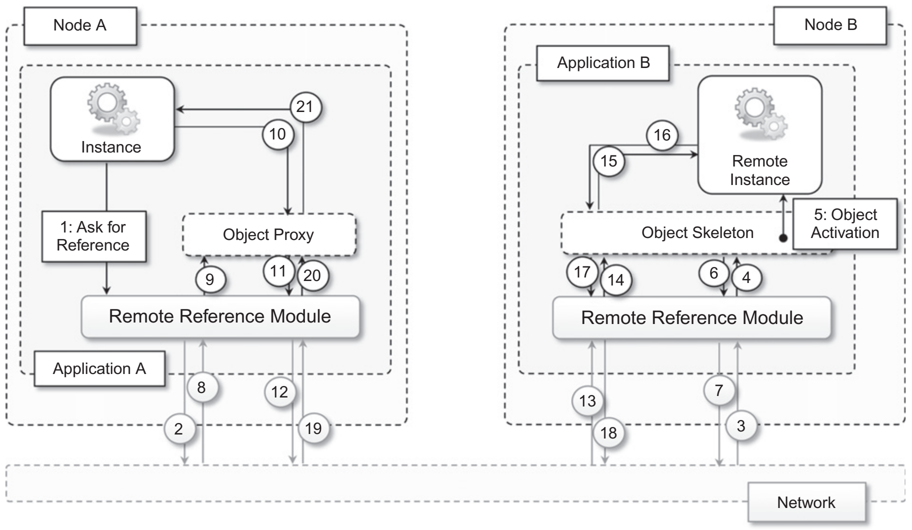


FIGURE 2.15

The distributed object programming model.

for the corresponding method on the remote active object. On the server side, whenever an RPC request is received, it is unpacked and the method call is dispatched to the skeleton that is paired with the client that issued the request. Once the method execution on the server is completed, the return values are packed and sent back to the client, and the local method call on the proxy returns.

Distributed object frameworks introduce objects as first-class entities for IPC. They are the principal gateway for invoking remote methods but can also be passed as parameters and return values. This poses an interesting problem, since object instances are complex instances that encapsulate a state and might be referenced by other components. Passing an object as a parameter or return value involves the duplication of the instance on the other execution context. This operation leads to two separate objects whose state evolves independently. The duplication becomes necessary since the instance needs to trespass the boundaries of the process. This is an important aspect to take into account in designing distributed object systems, because it might lead to inconsistencies. An alternative to this standard process, which is called *marshaling by value*, is *marshaling by reference*. In this second case the object instance is not duplicated and a proxy of it is created on the server side (for parameters) or the client side (for return values). Marshaling by reference is a more complex technique and generally puts more burden on the runtime infrastructure since remote references have to be tracked. Being more complex and resource demanding, marshaling by reference should be used only when duplication of parameters and return values lead to unexpected and inconsistent behavior of the system.

2.5.2.1 Object activation and lifetime

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

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

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

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

2.5.2.2 Examples of distributed object frameworks

The support for distributed object programming has evolved over time, and today it is a common feature of mainstream programming languages such as C# and Java, which provide these capabilities as part of the base class libraries. This level of integration is a sign of the maturity of this technology, which originally was designed as a separate component that could be used in several programming languages. In this section, we briefly review the most relevant approaches to and technologies for distributed object programming.

Common object request broker architecture (CORBA)

CORBA is a specification introduced by the Object Management Group (OMG) for providing cross-platform and cross-language interoperability among distributed components. The specification was originally designed to provide an interoperation standard that could be effectively used at the industrial level. The current release of the CORBA specification is version 3.0 and currently the technology is not very popular, mostly because the development phase is a considerably complex task and the interoperability among components developed in different languages has never reached the proposed level of transparency. A fundamental component in the CORBA architecture is the *Object Request Broker (ORB)*, which acts as a central object bus. A CORBA object registers with the ORB the interface it is exposing, and clients can obtain a reference to that interface and invoke methods on it. The ORB is responsible for returning the reference to the client and managing all the low-level operations required to perform the remote method invocation. To simplify cross-platform interoperability, interfaces are defined in *Interface Definition Language (IDL)*, which provides a platform-independent specification of a component. An IDL specification is then translated into a *stub-skeleton* pair by specific CORBA compilers that generate the required client (stub) and server (skeleton) components in a specific programming language. These templates are completed with an appropriate implementation in the selected programming language. This allows CORBA components to be used across different runtime environment by simply using the stub and the skeleton that match the development language used. A specification meant to be used at the industry level, CORBA provides interoperability among different implementations of its runtime. In particular, at the lowest-level ORB implementations communicate with each other using the *Internet Inter-ORB Protocol (IIOP)*, which standardizes the interactions of different ORB implementations. Moreover, CORBA provides an additional level of abstraction and separates the ORB, which mostly deals with the networking among nodes, from the *Portable Object Adapter (POA)*, which is the runtime environment in which the skeletons are hosted and managed. Again, the interface of these two layers is clearly defined, thus giving more freedom and allowing different implementations to work together seamlessly.

Distributed component object model (DCOM/COM+)

DCOM, later integrated and evolved into COM+, is the solution provided by Microsoft for distributed object programming before the introduction of .NET technology. DCOM introduces a set of features allowing the use of COM components beyond the process boundaries. A COM object identifies a component that encapsulates a set of coherent and related operations; it was designed to be easily plugged into another application to leverage the features exposed through its interface. To support interoperability, COM standardizes a binary format, thus allowing the use of COM objects across different programming languages. DCOM enables such capabilities in a distributed environment by adding the required IPC support. The architecture of DCOM is quite similar to CORBA but simpler, since it does not aim to foster the same level of interoperability; its implementation is monopolized by Microsoft, which provides a single runtime environment. A DCOM server object can expose several interfaces, each representing a different behavior of the object. To invoke the methods exposed by the interface, clients obtain a pointer to that interface and use it as though it were a pointer to an object in the client's address space. The DCOM runtime is responsible for performing all the operations required to create this illusion. This technology provides a reasonable level of interoperability among Microsoft-based environments, and there are third-party implementations that allow the use of DCOM, even in Unix-based environments. Currently, even if still used in industry, this technology is no longer popular and has been replaced by other approaches, such as .NET Remoting and Web Services.

Java remote method invocation (RMI)

Java RMI is a standard technology provided by Java for enabling RPC among distributed Java objects. RMI defines an infrastructure allowing the invocation of methods on objects that are located on different Java Virtual Machines (JVMs) residing either on the local node or on a remote one. As with CORBA, RMI is based on the *stub-skeleton* concept. Developers define an interface extending *java.rmi.Remote* that defines the contract for IPC. Java allows only publishing interfaces while it relies on actual types for the server and client part implementation. A class implementing the previous interface represents the *skeleton* component that will be made accessible beyond the JVM boundaries. The *stub* is generated from the skeleton class definition using the *rmic* command-line tool. Once the *stub-skeleton* pair is prepared, an instance of the skeleton is registered with the RMI registry that maps URIs, through which instances can be reached, to the corresponding objects. The RMI registry is a separate component that keeps track of all the instances that can be reached on a node. Clients contact the RMI registry and specify a URI, in the form *rmi://host:port/serviceName*, to obtain a reference to the corresponding object. The RMI runtime will automatically retrieve the class information for the stub component paired with the skeleton mapped with the given URI and return an instance of it properly configured to interact with the remote object. In the client code, all the services provided by the skeleton are accessed by invoking the methods defined in the remote interface. RMI provides a quite transparent interaction pattern. Once the development and deployment phases are completed and a reference to a remote object is obtained, the client code interacts with it as though it were a local instance, and RMI performs all the required operations to enable the IPC. Moreover, RMI also allows customizing the security that has to be applied for remote objects. This is done by leveraging the standard Java security infrastructure, which allows specifying policies defining the permissions attributed to the JVM hosting the remote object.

.NET remoting

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

Remoting allows objects located in different application domains to interact in a completely transparent manner, whether the two domains are in the same process, in the same machine, or on different nodes. The reference architecture is based on the classic client/server model whereby the application domain hosting the remote object is the server and the application domain accessing it is the client. Developers define a class that inherits by *MarshalByRefObject*, the base class that provides the built-in facilities to obtain a reference of an instance from another application domain. Instances of types that do not inherit from *MarshalByRefObject* are copied across application domain boundaries. There is no need to manually generate a stub for a type that needs to be exposed remotely. The Remoting infrastructure will automatically provide all the required information to generate a proxy on a client application domain. To make a component accessible through Remoting requires the component to be registered with the Remoting runtime and mapping it to a specific URI in the form *scheme://host:port/ServiceName,* where *scheme* is generally TCP or HTTP. It is possible to use different strategies to publish the remote component: Developers can provide an instance of the type developed or simply the type information. When only the type information is provided, the activation of the object is automatic and client-based, and developers can control the lifetime of the objects by overriding the default behavior of *MarshalByRefObject*. To interact with a remote object, client application domains have to query the remote infrastructure by providing a URI identifying the remote object and they will obtain a proxy to the remote object. From there on, the interaction with the remote object is completely transparent. As happens for Java RMI, Remoting allows customizing the security measures applied for the execution of code triggered by Remoting calls.

These are the most popular technologies for enabling distributed object programming. CORBA is an industrial-standard technology for developing distributed systems spanning different platforms and vendors. The technology has been designed to be interoperable among a variety of implementations and languages. Java RMI and .NET Remoting are built-in infrastructures for IPC, serving the purpose of creating distributed applications based on a single technology: Java and .NET, respectively. With respect to CORBA, they are less complex to use and deploy but are not natively interoperable. By relying on a unified platform, both Java and .NET Remoting are very straightforward and intuitive and provide a transparent interaction pattern that naturally fits in the structure of the supported languages. Although the two architectures are similar, they have some minor differences: Java relies on an external component called *RMI registry* to locate remote objects and allows only the publication of interfaces, whereas .NET Remoting does not use a registry and allows developers to expose class types as well. Both technologies have been extensively used to develop distributed applications.

2.5.3 Service-oriented computing

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

2.5.3.1 What is a service?

A *service* encapsulates a software component that provides a set of coherent and related functionalities that can be reused and integrated into bigger and more complex applications. The term *service* is a general abstraction that encompasses several different implementations using different technologies and protocols. Don Box [107] identifies four major characteristics that identify a service:

- *Boundaries are explicit.* A service-oriented application is generally composed of services that are spread across different domains, trust authorities, and execution environments. Generally, crossing such boundaries is costly; therefore, service invocation is explicit by design and often leverages message passing. With respect to distributed object programming, whereby remote method invocation is transparent, in a service-oriented computing environment the interaction with a service is explicit and the interface of a service is kept minimal to foster its reuse and simplify the interaction.
- *Services are autonomous.* Services are components that exist to offer functionality and are aggregated and coordinated to build more complex system. They are not designed to be part of a specific system, but they can be integrated in several software systems, even at the same time. With respect to object orientation, which assumes that the deployment of applications is atomic, service orientation considers this case an exception rather than the rule and puts the focus on the design of the service as an autonomous component. The notion of autonomy also affects the way services handle failures. Services operate in an unknown environment and interact with third-party applications. Therefore, minimal assumptions can be made concerning such environments: applications may fail without notice, messages can be malformed, and clients can be unauthorized. Service-oriented design addresses these issues by using transactions, durable queues, redundant deployment and failover, and administratively managed trust relationships among different domains.
- *Services share schema and contracts, not class or interface definitions.* Services are not expressed in terms of classes or interfaces, as happens in object-oriented systems, but they define themselves in terms of schemas and contracts. A service advertises a contract describing the structure of messages it can send and/or receive and additional constraint—if any—on their ordering. Because they are not expressed in terms of types and classes, services are more easily consumable in wider and heterogeneous environments. At the same time, a service orientation requires that contracts and schema remain stable over time, since it would be possible to propagate changes to all its possible clients. To address this issue, contracts and schema are defined in a way that allows services to evolve without breaking already deployed code. Technologies such as XML and SOAP provide the appropriate tools to support such features rather than class definition or an interface declaration.
- *Services compatibility is determined based on policy.* Service orientation separates structural compatibility from semantic compatibility. Structural compatibility is based on contracts and schema and can be validated or enforced by machine-based techniques. Semantic compatibility is expressed in the form of policies that define the capabilities and requirements for a service. Policies are organized in terms of expressions that must hold true to enable the normal operation of a service.

Today services constitute the most popular abstraction for designing complex and interoperable systems. Distributed systems are meant to be heterogeneous, extensible, and dynamic. By abstracting away from a specific implementation technology and platform, they provide a more efficient way to achieve integration. Furthermore, being designed as autonomous components, they can be more easily reused and aggregated. These features are not carved from a smart system design and implementation—as happens in the case of distributed object programming—but instead are part of the service characterization.

2.5.3.2 Service-oriented architecture (SOA)

SOA [20] is an architectural style supporting service orientation.[5] It organizes a software system into a collection of interacting services. SOA encompasses a set of design principles that structure system development and provide means for integrating components into a coherent and decentralized system. SOA-based computing packages functionalities into a set of interoperable services, which can be integrated into different software systems belonging to separate business domains.

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

SOA provides a reference model for architecting several software systems, especially enterprise business applications and systems. In this context, interoperability, standards, and service contracts play a fundamental role. In particular, the following guiding principles [108], which characterize SOA platforms, are winning features within an enterprise context:

- *Standardized service contract*. Services adhere to a given communication agreement, which is specified through one or more service description documents.
- *Loose coupling*. Services are designed as self-contained components, maintain relationships that minimize dependencies on other services, and only require being aware of each other. Service contracts will enforce the required interaction among services. This simplifies the flexible aggregation of services and enables a more agile design strategy that supports the evolution of the enterprise business.
- *Abstraction*. A service is completely defined by service contracts and description documents. They hide their logic, which is encapsulated within their implementation. The use of service description documents and contracts removes the need to consider the technical implementation

[5]This definition is given by the Open Group (www.opengroup.org), which is a vendor- and technology-neutral consortium that includes over 300 member organizations. Its activities include management, innovation, research, standards, certification, and test development. The Open Group is most popular as a certifying body for the UNIX trademark, since it is also the creator of the official definition of a UNIX system. The documentation and the standards related to SOA can be found at the following address: www.opengroup.org/soa/soa/def.htm.

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

Together with these principles, other resources guide the use of SOA for *enterprise application integration (EAI)*. The SOA manifesto[6] integrates the previously described principles with general considerations about the overall goals of a service-oriented approach to enterprise application software design and what is valued in SOA. Furthermore, modeling frameworks and methodologies, such as the *Service-Oriented Modeling Framework (SOMF)* [110] and reference architectures introduced by *the Organization for Advancement of Structured Information Standards (OASIS)* [110], provide means for effectively realizing service-oriented architectures.

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

2.5.3.3 Web services

Web services [21] are the prominent technology for implementing SOA systems and applications. They leverage Internet technologies and standards for building distributed systems. Several aspects make Web services the technology of choice for SOA. First, they allow for interoperability across different platforms and programming languages. Second, they are based on well-known and vendor-independent standards such as HTTP, SOAP [23], XML, and WSDL [22]. Third, they provide an intuitive and simple way to connect heterogeneous software systems, enabling the quick

[6]The SOA manifesto is a document authored by 17 practitioners of SOA that defines guidelines and principles for designing and architecting software systems using a service orientation. The document is available online at: www.soa-manifesto.org.

composition of services in a distributed environment. Finally, they provide the features required by enterprise business applications to be used in an industrial environment. They define facilities for enabling service discovery, which allows system architects to more efficiently compose SOA applications, and service metering to assess whether a specific service complies with the contract between the service provider and the service consumer.

The concept behind a Web service is very simple. Using as a basis the object-oriented abstraction, a Web service exposes a set of operations that can be invoked by leveraging Internet-based protocols. Method operations support parameters and return values in the form of complex and simple types. The semantics for invoking Web service methods is expressed through interoperable standards such as XML and WSDL, which also provide a complete framework for expressing simple and complex types in a platform-independent manner. Web services are made accessible by being hosted in a Web server; therefore, HTTP is the most popular transport protocol used for interacting with Web services. Figure 2.16 describes the common-use case scenarios for Web services.

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

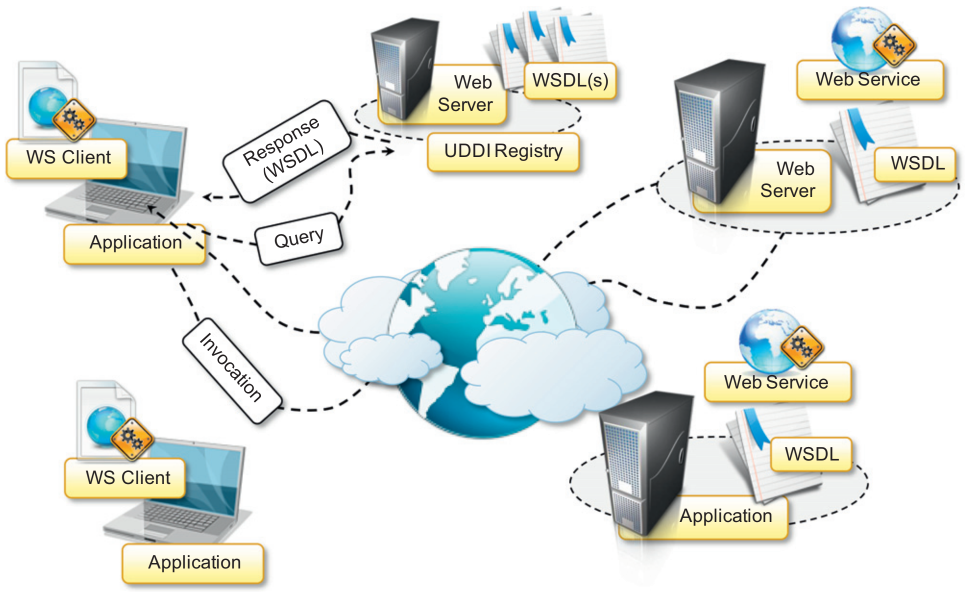


FIGURE 2.16

A Web services interaction reference scenario.

exist for any mainstream programming language in the form of libraries or development support tools. This makes the use of Web services seamless and straightforward with respect to technologies such as CORBA that require much more integration effort. Moreover, being interoperable, Web services constitute a better solution for SOA with respect to several distributed object frameworks, such as .NET Remoting, Java RMI, and DCOM/COM+ , which limit their applicability to a single platform or environment.

Besides the main function of enabling remote method invocation by using Web-based and interoperable standards, Web services encompass several technologies that put together and facilitate the integration of heterogeneous applications and enable service-oriented computing. Figure 2.17 shows the Web service technologies stack that lists all the components of the conceptual framework describing and enabling the Web services abstraction. These technologies cover all the aspects that allow Web services to operate in a distributed environment, from the specific requirements for the networking to the discovery of services. The backbone of all these technologies is XML, which is also one of the causes of Web services' popularity and ease of use. XML-based languages are used to manage the low-level interaction for Web service method calls (SOAP), for providing metadata about the services (WSDL), for discovery services (UDDI), and other core operations. In practice, the core components that enable Web services are SOAP and WSDL.

Simple Object Access Protocol (SOAP) [23], an XML-based language for exchanging structured information in a platform-independent manner, constitutes the protocol used for Web service method invocation. Within a distributed context leveraging the Internet, SOAP is considered an application layer protocol that leverages the transport level, most commonly HTTP, for IPC. SOAP structures the interaction in terms of messages that are XML documents mimicking the structure of a letter, with an envelope, a header, and a body. The envelope defines the boundaries of the SOAP message. The header is optional and contains relevant information on how to process the message. In addition, it contains information such as routing and delivery settings, authentication and authorization assertions, and transaction contexts. The body contains the actual message to be processed.

The main uses of SOAP messages are method invocation and result retrieval. Figure 2.18 shows an example of a SOAP message used to invoke a Web service method that retrieves the price of a

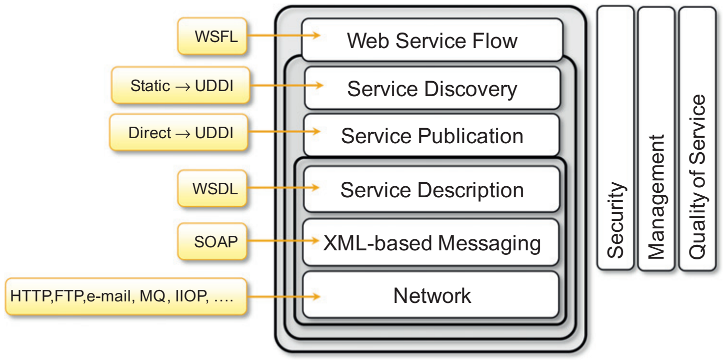
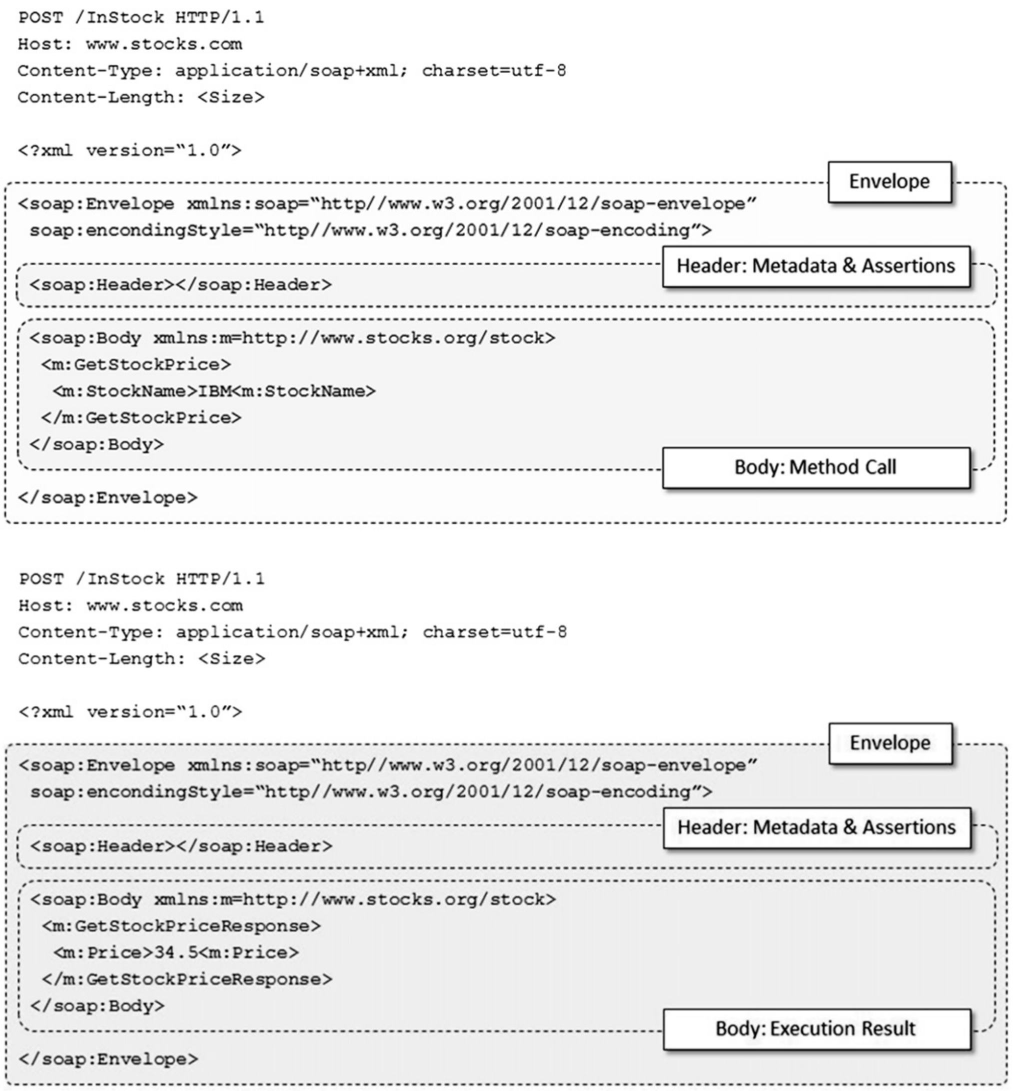


FIGURE 2.17

A Web services technologies stack.



```
POST /InStock HTTP/1.1
Host: www.stocks.com
Content-Type: application/soap+xml; charset=utf-8
Content-Length: <Size>

<?xml version="1.0">
<soap:Envelope xmlns:soap="http//www.w3.org/2001/12/soap-envelope"
 soap:encondingStyle="http//www.w3.org/2001/12/soap-encoding">
<soap:Header></soap:Header>
<soap:Body xmlns:m=http://www.stocks.org/stock>
 <m:GetStockPrice>
  <m:StockName>IBM<m:StockName>
 </m:GetStockPrice>
</soap:Body>
</soap:Envelope>
```

```
POST /InStock HTTP/1.1
Host: www.stocks.com
Content-Type: application/soap+xml; charset=utf-8
Content-Length: <Size>

<?xml version="1.0">
<soap:Envelope xmlns:soap="http//www.w3.org/2001/12/soap-envelope"
 soap:encondingStyle="http//www.w3.org/2001/12/soap-encoding">
<soap:Header></soap:Header>
<soap:Body xmlns:m=http://www.stocks.org/stock>
 <m:GetStockPriceResponse>
  <m:Price>34.5<m:Price>
 </m:GetStockPriceResponse>
</soap:Body>
</soap:Envelope>
```

FIGURE 2.18

SOAP messages for Web service method invocation.

given stock and the corresponding reply. Despite the fact that XML documents are easy to produce and process in any platform or programming language, SOAP has often been considered quite inefficient because of the excessive use of markup that XML imposes for organizing the information into a well-formed document. Therefore, lightweight alternatives to the SOAP/XML pair have been proposed to support Web services. The most relevant alternative is *Representational State Transfer*

(REST), which provides a model for designing network-based software systems utilizing the client/server model and leverages the facilities provided by HTTP for IPC without additional burden.

In a *RESTful* system, a client sends a request over HTTP using the standard HTTP methods (*PUT*, *GET*, *POST*, and *DELETE*), and the server issues a response that includes the representation of the resource. By relying on this minimal support, it is possible to provide whatever it needed to replace the basic and most important functionality provided by SOAP, which is method invocation. The GET, PUT, POST, and DELETE methods constitute a minimal set of operations for retrieving, adding, modifying, and deleting data. Together with an appropriate URI organization to identify resources, all the atomic operations required by a Web service are implemented. The content of data is still transmitted using XML as part of the HTTP content, but the additional markup required by SOAP is removed. For this reason, REST represents a lightweight alternative to SOAP, which works effectively in contexts where additional aspects beyond those manageable through HTTP are absent. One of them is security; *RESTful* Web services operate in an environment where no additional security beyond the one supported by HTTP is required. This is not a great limitation, and *RESTful* Web services are quite popular and used to deliver functionalities at enterprise scale: *Twitter*, *Yahoo!* (search APIs, maps, photos, etc), *Flickr*, and *Amazon.com* all leverage REST.

Web Service Description Language (WSDL) [22] is an XML-based language for the description of Web services. It is used to define the interface of a Web service in terms of methods to be called and types and structures of the required parameters and return values. In Figure 2.18 we notice that the SOAP messages for invoking the *GetStockPrice* method and receiving the result do not have any information about the type and structure of the parameters and the return values. This information is stored within the WSDL document attached to the Web service. Therefore, Web service consumer applications already know which types of parameters are required and how to interpret results. As an XML-based language, WSDL allows for the automatic generation of Web service clients that can be easily embedded into existing applications. Moreover, XML is a platform- and language-independent specification, so clients for web services can be generated for any language that is capable of interpreting XML data. This is a fundamental feature that enables Web service interoperability and one of the reasons that make such technology a solution of choice for SOA.

Besides those directly supporting Web services, other technologies that characterize Web 2.0 [27] provide and contribute to enrich and empower Web applications and then SOA-based systems. These fall under the names of *Asynchronous JavaScript and XML (AJAX)*, *JavaScript Standard Object Notation (JSON)*, and others. AJAX is a conceptual framework based on JavaScript and XML that enables asynchronous behavior in Web applications by leveraging the computing capabilities of modern Web browsers. This transforms simple Web pages in full-fledged applications, thus enriching the user experience. AJAX uses XML to exchange data with Web services and applications; an alternative to XML is JSON, which allows representing objects and collections of objects in a platform-independent manner. Often it is preferred to transmit data in a AJAX context because, compared to XML, it is a lighter notation and therefore allows transmitting the same amount of information in a more concise form.

2.5.3.4 Service orientation and cloud computing

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

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

SUMMARY

In this chapter, we provided an introduction to parallel and distributed computing as a foundation for better understanding cloud computing. Parallel and distributed computing emerged as a solution for solving complex/"grand challenge" problems by first using multiple processing elements and then multiple computing nodes in a network. The transition from sequential to parallel and distributed processing offers high performance and reliability for applications. But it also introduces new challenges in terms of hardware architectures, technologies for interprocess communication, and algorithms and system design. We discussed the evolution of technologies supporting parallel processing and introduced the major reference models for designing and implementing distributed systems.

Parallel computing introduces models and architectures for performing multiple tasks within a single computing node or a set of tightly coupled nodes with homogeneous hardware. Parallelism is achieved by leveraging hardware capable of processing multiple instructions in parallel. Different architectures exploit parallelism to increase the performance of a computing system, depending on whether parallelism is realized on data, instructions, or both. The development of parallel applications often requires specific environments and compilers that provide transparent access to the advanced capabilities of the underlying architectures.

Unification of parallel and distributed computing allows one to harness a set of networked and heterogeneous computers and present them as a unified resource. Distributed systems constitute a large umbrella under which several different software systems are classified. Architectural styles help categorize and provide reference models for distributed systems. More precisely, software architectural styles define logical organizations of components and their roles, whereas system architectural styles are more concerned with the physical deployment of such systems. We have briefly reviewed the major reference software architectural styles and discussed the most important system architectural styles: the client/server and peer-to-peer models. These two styles are the fundamental deployment blocks of any distributed system. In particular, the client/server model is the foundation of the most popular interaction patterns among components within a distributed system.

Interprocess communication (IPC) is a fundamental element in distributed systems; it is the element that ties together separate processes and allows them to be seen as a whole. Message-based communication is the most relevant abstraction for IPC and forms the basis for several different 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.