MASTERING CLOUD COMPUTING

FOUNDATIONS AND APPLICATIONS PROGRAMMING

MORGAN KAUFMANN

Rajkumar Buyya, Christian Vecchiola, S. Thamarai Selvi

Mastering Cloud Computing

This page intentionally left blank

Mastering Cloud Computing

Foundations and Applications Programming

Rajkumar Buyya
The University of Melbourne and Manjrasoft Pty Ltd, Australia

Christian Vecchiola
The University of Melbourne and IBM Research, Australia

S. Thamarai Selvi
Madras Institute of Technology, Anna University, Chennai, India

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Morgan Kaufmann is an imprint of Elsevier

Acquiring Editor: *Todd Green*
Editorial Project Manager: *Lindsay Lawrence*
Project Manager: *Punithavathy Govindaradjane*
Designer: *Matthew Limbert*

Morgan Kaufmann is an imprint of Elsevier
225 Wyman Street, Waltham, MA 02451, USA

Notices
Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods or professional practices, may become necessary. Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information or methods described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Cataloging-in-Publication Data
Application submitted

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

ISBN: 978-0-12-411454-8

Printed and bound in the United States of America
13 14 15 16 17 10 9 8 7 6 5 4 3 2 1

For information on all MK publications visit our website at www.mkp.com

Contents

Acknowledgments

First and foremost, we are grateful to all researchers and industrial developers worldwide for their contributions to various concepts and technologies discussed in this book. Our special thanks to all the members and consultants of Manjrasoft, the Cloud Computing and Distributed Systems (CLOUDS) Lab of the University of Melbourne, and Melbourne Ventures, who contributed to the development of the Aneka Cloud Application Platform, the preparation of associated application demonstrators and documents, and/or the commercialization of the Aneka technology. They include Chu Xingchen, Srikumar Venugopal, Krishna Nadiminti, Christian Vecchiola, Dileban Karunamoorthy, Chao Jin, Rodrigo Calheiros, Michael Mattess, Jessie Wei, Enayat Masoumi, Ivan Mellado, Richard Day, Wolfgang Gentzsch, Laurence Liew, David Sinclair, Suraj Pandey, Abhi Shekar, Dexter Duncan, Murali Sathya, Karthik Sukumar, Ravi Kumar Challa, and Sita Venkatraman.

We thank the Australian Research Council (ARC) and the Department of Innovation, Industry, Science, and Research (DIISR) for supporting our research and commercialization endeavors.

We thank all of our colleagues at the University of Melbourne, especially Professors Rao Kotagiri, Iven Mareels, and Glyn Davis, for their mentorship and positive support for our research and our efforts to impart the knowledge we have gained.

We thank all colleagues and users of the Aneka technology for their direct and indirect contributions to application case studies reported in the book. Our special thanks to Raghavendra Kune from ADRIN/ISRO for his enthusiastic efforts in creating a satellite image-processing application using Aneka and publishing articles in this area. We thank Srinivasa Iyengar from MSRIT for creating data-mining applications using Aneka and demonstrating the power of Aneka to academics from the early days of cloud computing.

We thank the members of the CLOUDS Lab for proofreading one or more chapters. They include Rodrigo Calheiros, Nikolay Grozev, Amir Vahid, Anton Beloglazov, Adel Toosi, Deepak Poola, Mohammed AlRokayan, Atefeh Khosravi, Sareh Piraghaj, and Yaser Mansouri.

We thank our family members, including Smrithi Buyya, Soumya Buyya, and Radha Buyya, for their love and understanding during the preparation of the book.

We sincerely thank external reviewers commissioned by the publisher for their critical comments and suggestions on enhancing the presentation and organization of many chapters at a finer level. This has greatly helped us improve the quality of the book.

Finally, we would like to thank the staff at Elsevier Inc for their enthusiastic support and guidance during the preparation of the book. In particular, we thank Todd Green for inspiring us to take up this project and for setting the process of publication in motion. The Elsevier staff were wonderful to work with!

Professor Rajkumar Buyya
The University of Melbourne and Manjrasoft Pty Ltd, Australia
Dr. Christian Vecchiola
The University of Melbourne and IBM Research, Australia
Professor S. Thamarai Selvi
Madras Institute of Technology, Anna University, Chennai, India

This page intentionally left blank

Preface

The growing popularity of the Internet and the Web, along with the availability of powerful handheld computing, mobile, and sensing devices, are changing the way we interact, manage our lives, conduct business, and access or deliver services. The lowering costs of computation and communication are driving the focus from personal to datacenter-centric computing. Although parallel and distributed computing has been around for several years, its new forms, multicore and cloud computing, have brought about a sweeping change in the industry. These trends are pushing the industry focus from developing applications for PCs to cloud datacenters that enable millions of users to use software simultaneously.

Computing is being transformed to a model consisting of commoditized services delivered in a manner similar to utilities such as water, electricity, gas, and telephony. As a result, information technology (IT) services are billed and delivered as "computing utilities" over shared delivery networks, akin to water, electricity, gas, and telephony services delivery. In such a model, users access services based on their requirements, regardless of where those services are hosted. Several computing paradigms have promised to deliver this utility computing vision. Cloud computing is the most recent emerging paradigm promising to turn the vision of "computing utilities" into a reality.

Cloud computing has become one of the buzzwords in the IT industry. Several IT vendors are promising to offer storage, computation, and application hosting services and to provide coverage on several continents, offering service-level agreements-backed performance and uptime promises for their services. They offer subscription-based access to infrastructure, platforms, and applications that are popularly termed Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS). These emerging services have reduced the cost of computation and application hosting by several orders of magnitude, but there is significant complexity involved in the development and delivery of applications and their services in a seamless, scalable, and reliable manner.

There are several cloud technologies and platforms on the market—to mention a few: Google AppEngine, Microsoft Azure, and Manjrasoft Aneka. Google AppEngine provides an extensible runtime environment for Web-based applications that leverage the huge Google IT infrastructure. Microsoft Azure provides a wide array of Windows-based services for developing and deploying Windows applications on the cloud. Manjrasoft Aneka provides a flexible model for creating cloud applications and deploying them on a wide variety of infrastructures, including public clouds such as Amazon EC2.

With this sweeping shift from developing applications on PCs to datacenters, there is a huge demand for manpower with new skill sets in cloud computing. Universities play an important role in this regard by training the next generation of IT professionals and equipping them with the necessary tools and knowledge to tackle these challenges. These institutions need to be able to set up a cloud computing environment for teaching and learning with minimal investment. One of the attractive cloud application platforms that meet this need is Manjrasoft's Aneka, which (1) enables the construction of a private/enterprise cloud by harnessing the existing network of computers

(LAN-connected PCs), (2) provides a software development kit (SDK) that supports application programming interfaces (APIs) for multiple programming models such as Thread, Task, and MapReduce, and (3) supports, in a seamless manner, the deployment and execution of applications on diverse infrastructures such as multicore servers, private clouds, and public clouds.

Currently, expert developers are required to create cloud applications and services. Cloud researchers, practitioners, and vendors alike are working to ensure that potential users are educated about the benefits of cloud computing and the best way to harness its full potential. However, because it's a new and popular paradigm, the very definition of cloud computing depends on which computing expert is asked. So, although the realization of true utility computing appears closer than ever, its acceptance is currently restricted to cloud experts due to the perceived complexities of interacting with cloud computing providers. This book aims to change the game by simplifying and imparting cloud computing foundations, technologies, and programming skills to readers so that even average programmers and software engineers are able to develop cloud applications easily.

The book at a glance

This book introduces the fundamental principles of cloud computing and its related paradigms. It discusses the concepts of virtualization technologies along with the architectural models of cloud computing. It presents prominent cloud computing technologies that are available in the marketplace, including the Aneka Cloud Application Platform. The book contains chapters dedicated to discussion of concurrent, high-throughput, and data-intensive computing paradigms and their use in programming cloud applications. Various application case studies from domains such as science, engineering, gaming, and social networking are introduced, along with their architecture and how they leverage various cloud technologies. These case studies allow the reader to understand the mechanisms needed to harness cloud computing in their own respective endeavors. Finally, the book details many open research problems and opportunities that have arisen from the rapid uptake of cloud computing. We hope that this motivates the reader to address these in their own future research and development. The book also comes with an associated Website (hosted at www.buyya.com/MasteringClouds) that contains pointers to advanced online resources.

The book contains 11 chapters, which are organized into three major parts:

Part 1: Foundations
- Chapter 1—Introduction
- Chapter 2—Principles of Parallel and Distributed Computing
- Chapter 3—Virtualization
- Chapter 4—Cloud Computing Architecture

Part 2: Cloud Application Programming and the Aneka Platform
- Chapter 5—Aneka: Cloud Application Platform
- Chapter 6—Concurrent Computing: Thread Programming
- Chapter 7—High-Throughput Computing: Task Programming
- Chapter 8—Data-Intensive Computing: MapReduce Programming

Part 3: Industrial Platforms and New Developments
Chapter 9—Cloud Platforms in Industry
Chapter 10—Cloud Applications
Chapter 11—Advanced Topics in Cloud Computing

The book serves as a perfect guide to the world of cloud computing. Starting with the fundamentals, the book drives students and professionals through the practical use of these concepts via hands-on sessions on how to develop cloud applications, using Aneka as a reference platform. Part 3 goes beyond the reference platform and introduces other industrial technologies and solutions (Amazon Web Services, Google AppEngine, and Microsoft Azure) and real applications, identifies emerging trends, and offers future directions for cloud computing.

Benefits and readership

Given the rapid emergence of cloud computing as a mainstream computing paradigm, it is essential to have both a solid understanding of the core concepts characterizing the phenomenon and a practical grasp of how to design and implement cloud computing applications and systems. This set of skills is already fundamental today for software architects, engineers, and developers because many applications are being moved to the cloud. It will become even more important in the future, when this technology matures further. This book provides an ideal blend of background information, theory, and practical cloud computing development techniques, expressed in a language that is accessible to a wide range of readers: from graduate-level students to practitioners, developers, and engineers who want to, or need to, design and implement cloud computing solutions. Moreover, more advanced topics presented at the end of the manuscript make the book an interesting read for researchers in the field of cloud computing who want an overview of the next challenges in cloud computing that will arise in coming years.

This book is a timely contribution to the cloud computing field, which is gaining considerable commercial interest and momentum. The book is targeted at graduate students and IT professionals such as system architects, practitioners, software engineers, and application programmers. As cloud computing is recognized as one of the top five emerging technologies that will have a major impact on the quality of science and society over the next 20 years, the knowledge conveyed through this book will help position our readers at the forefront of the field.

Directions for adoption: theory, labs, and projects

Given the importance of the cloud computing paradigm and its rapid uptake in industry, universities and educational institutions need to upgrade their curriculum by introducing one or more subjects in the area of cloud computing and related topics, such as parallel computing and distributed systems. We recommend that they offer at least one subject on cloud computing as part of their undergraduate and postgraduate degree programs, such as B.E./B.Tech./BSc in computer science and related areas and Masters, including the Master of Computer Applications (MCA). We believe that

this book will serve as an excellent textbook for such subjects. If the students have already had exposure to the concepts of parallel and distributed computing, Chapter 2 can be skipped.

For those aiming to make their curriculum rich with cloud computing, we recommend offering two courses: "Introduction to Cloud Computing" and "Advanced Cloud Computing," in two different semesters. This book has sufficient content to cater to both of them. The first subject can be based on Chapters 1−6 and the second one based on Chapters 7−11.

In addition to theory, we strongly recommend the introduction of a laboratory subject that offers hands-on experience. The lab exercises and assignments can focus on creating high-performance cloud applications and assignments on a range of topics, including parallel execution of mathematical functions, sorting of large data in parallel, image processing, and data mining. Using cloud software systems such as Aneka, institutions can easily set up a private/enterprise cloud computing facility by utilizing existing LAN-connected PCs running Windows. Students can use this facility to learn about various cloud application programming models and interfaces discussed in Chapter 6 (Thread Programming), Chapter 7 (Task Programming), and Chapter 8 (MapReduce Programming). Students need to learn various programming examples discussed in these chapters and execute them on an Aneka-based cloud facility. We encourage students to take up some of the programming exercises noted in the "Review Questions" sections of these chapters as lab assignments and develop their own solutions.

Students can also carry out their final-year projects focused on developing cloud applications to solve real-world problems. For example, students can work with academics, researchers, and experts from other science and engineering disciplines, such as life and medical sciences or civil and mechanical engineering, and develop suitable applications that can harness the power of cloud computing. For inspiration, please read various application case studies presented in Chapter 10.

Supplemental materials

Supplemental materials for instructors or students can be downloaded from Elsevier: http://store.elsevier.com/product.jsp?isbn=9780124114548

PART

Foundations

1

This page intentionally left blank

CHAPTER

1 Introduction

Computing is being transformed into a model consisting of services that are commoditized and delivered in a manner similar to utilities such as water, electricity, gas, and telephony. In such a model, users access services based on their requirements, regardless of where the services are hosted. Several computing paradigms, such as grid computing, have promised to deliver this utility computing vision. *Cloud computing* is the most recent emerging paradigm promising to turn the vision of "computing utilities" into a reality.

Cloud computing is a technological advancement that focuses on the way we design computing systems, develop applications, and leverage existing services for building software. It is based on the concept of *dynamic provisioning*, which is applied not only to services but also to compute capability, storage, networking, and information technology (IT) infrastructure in general. Resources are made available through the Internet and offered on a *pay-per-use* basis from cloud computing vendors. Today, anyone with a credit card can subscribe to cloud services and deploy and configure servers for an application in hours, growing and shrinking the infrastructure serving its application according to the demand, and paying only for the time these resources have been used.

This chapter provides a brief overview of the cloud computing phenomenon by presenting its vision, discussing its core features, and tracking the technological developments that have made it possible. The chapter also introduces some key cloud computing technologies as well as some insights into development of cloud computing environments.

1.1 Cloud computing at a glance

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

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

This vision of computing utilities based on a service-provisioning model anticipated the massive transformation of the entire computing industry in the 21st century, whereby computing services will be readily available on demand, just as other utility services such as water, electricity, telephone, and gas are available in today's society. Similarly, users (consumers) need to pay providers

only when they access the computing services. In addition, consumers no longer need to invest heavily or encounter difficulties in building and maintaining complex IT infrastructure.

In such a model, users access services based on their requirements without regard to where the services are hosted. This model has been referred to as *utility computing* or, recently (since 2007), as *cloud computing*. The latter term often denotes the infrastructure as a "cloud" from which businesses and users can access applications as services from anywhere in the world and on demand. Hence, cloud computing can be classified as a new paradigm for the dynamic provisioning of computing services supported by state-of-the-art data centers employing virtualization technologies for consolidation and effective utilization of resources.

Cloud computing allows renting infrastructure, runtime environments, and services on a pay-per-use basis. This principle finds several practical applications and then gives different images of cloud computing to different people. Chief information and technology officers of large enterprises see opportunities for scaling their infrastructure on demand and sizing it according to their business needs. End users leveraging cloud computing services can access their documents and data anytime, anywhere, and from any device connected to the Internet. Many other points of view exist.[1] One of the most diffuse views of cloud computing can be summarized as follows:

> *I don't care where my servers are, who manages them, where my documents are stored, or where my applications are hosted. I just want them always available and access them from any device connected through Internet. And I am willing to pay for this service for as a long as I need it.*

The concept expressed above has strong similarities to the way we use other services, such as water and electricity. In other words, cloud computing turns IT services into *utilities*. Such a delivery model is made possible by the effective composition of several technologies, which have reached the appropriate maturity level. *Web 2.0* technologies play a central role in making cloud computing an attractive opportunity for building computing systems. They have transformed the Internet into a rich application and service delivery platform, mature enough to serve complex needs. *Service orientation* allows cloud computing to deliver its capabilities with familiar abstractions, while *virtualization* confers on cloud computing the necessary degree of customization, control, and flexibility for building production and enterprise systems.

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

[1] An interesting perspective on the way cloud computing evokes different things to different people can be found in a series of interviews made by Rob Boothby, vice president and platform evangelist of Joyent, at the Web 2.0 Expo in May 2007. Chief executive officers (CEOs), chief technology officers (CTOs), founders of IT companies, and IT analysts were interviewed, and all of them gave their personal perception of the phenomenon, which at that time was starting to spread. The video of the interview can be found on YouTube at the following link: www.youtube.com/watch?v=6PNuQHUiV3Q.

for cloud computing, which provides an open environment where computing, storage, and other services are traded as computing utilities.

1.1.1 The vision of cloud computing

Cloud computing allows anyone with a credit card to provision virtual hardware, runtime environments, and services. These are used for as long as needed, with no up-front commitments required. The entire stack of a computing system is transformed into a collection of utilities, which can be provisioned and composed together to deploy systems in hours rather than days and with virtually no maintenance costs. This opportunity, initially met with skepticism, has now become a practice across several application domains and business sectors (see Figure 1.1). The demand has fast-tracked technical development and enriched the set of services offered, which have also become more sophisticated and cheaper.

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

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

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

These are all possibilities that are introduced with the establishment of a global cloud computing marketplace and by defining effective standards for the unified representation of cloud services as well as the interaction among different cloud technologies. A considerable shift toward cloud computing has already been registered, and its rapid adoption facilitates its consolidation. Moreover, by concentrating the core capabilities of cloud computing into large datacenters, it is possible to reduce or remove the need for any technical infrastructure on the service consumer side. This approach provides opportunities for optimizing datacenter facilities and fully utilizing their

FIGURE 1.1

Cloud computing vision.

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

1.1.2 Defining a cloud

Cloud computing has become a popular buzzword; it has been widely used to refer to different technologies, services, and concepts. It is often associated with virtualized infrastructure or hardware on demand, utility computing, IT outsourcing, platform and software as a service, and many other things that now are the focus of the IT industry. Figure 1.2 depicts the plethora of different notions included in current definitions of cloud computing.

The term *cloud* has historically been used in the telecommunications industry as an abstraction of the network in system diagrams. It then became the symbol of the most popular computer network: the Internet. This meaning also applies to *cloud computing*, which refers to an Internet-centric way of

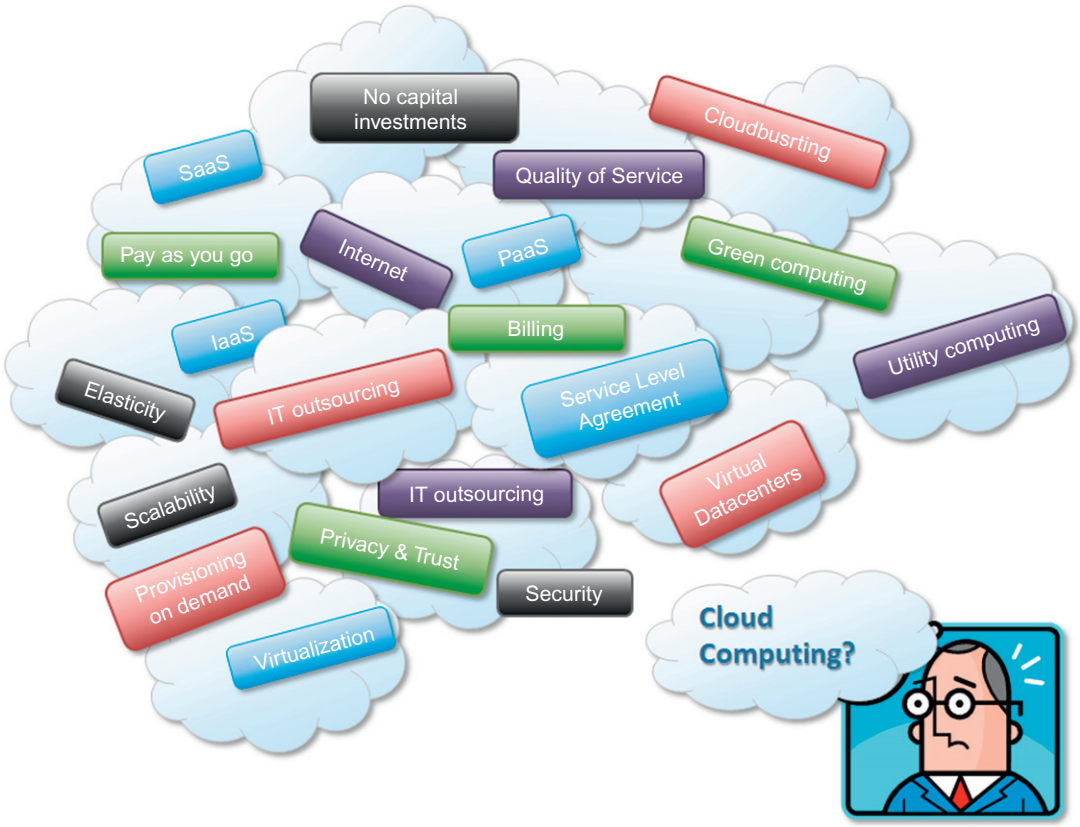

FIGURE 1.2

Cloud computing technologies, concepts, and ideas.

computing. The Internet plays a fundamental role in cloud computing, since it represents either the medium or the platform through which many cloud computing services are delivered and made accessible. This aspect is also reflected in the definition given by Armbrust et al. [28]:

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

This definition describes cloud computing as a phenomenon touching on the entire stack: from the underlying hardware to the high-level software services and applications. It introduces the concept of *everything as a service*, mostly referred as *XaaS*,[2] where the different components of a system—IT infrastructure, development platforms, databases, and so on—can be delivered, measured, and consequently priced as a service. This new approach significantly influences not only the way that we build software but also the way we deploy it, make it accessible, and design our IT infrastructure, and even the way companies allocate the costs for IT needs. The approach fostered by cloud computing is global: it covers both the needs of a single user hosting documents in the cloud and the ones of a CIO deciding to deploy part of or the entire corporate IT infrastructure in the public cloud. This notion of multiple parties using a shared cloud computing environment is highlighted in a definition proposed by the U.S. National Institute of Standards and Technology (NIST):

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

Another important aspect of cloud computing is its utility-oriented approach. More than any other trend in distributed computing, cloud computing focuses on delivering services with a given pricing model, in most cases a "pay-per-use" strategy. It makes it possible to access online storage, rent virtual hardware, or use development platforms and pay only for their effective usage, with no or minimal up-front costs. All these operations can be performed and billed simply by entering the credit card details and accessing the exposed services through a Web browser. This helps us provide a different and more practical characterization of cloud computing. According to Reese [29], we can define three criteria to discriminate whether a service is delivered in the cloud computing style:

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

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

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

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

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

1.1.3 A closer look

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

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

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

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

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

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

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

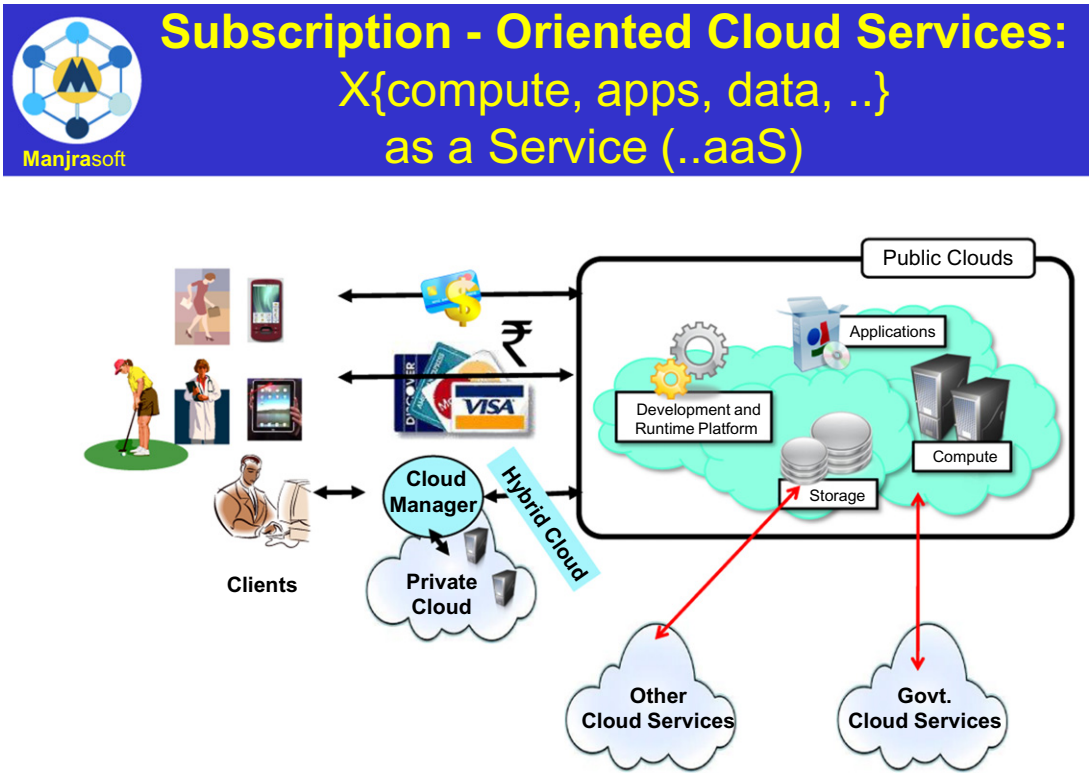

FIGURE 1.3

A bird's-eye view of cloud computing.

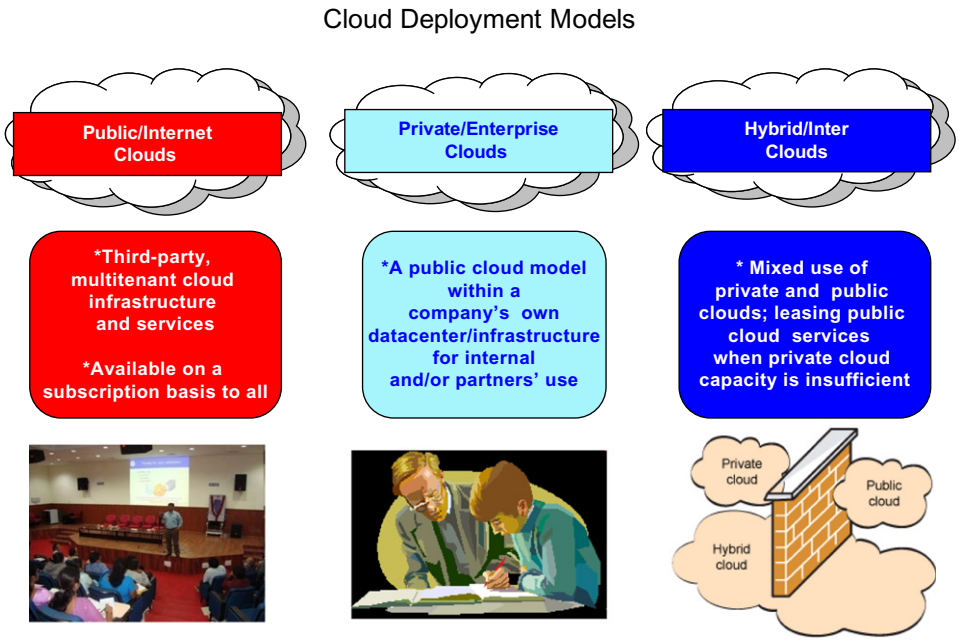

FIGURE 1.4

Major deployment models for cloud computing.

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

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

1.1.4 The cloud computing reference model

A fundamental characteristic of cloud computing is the capability to deliver, on demand, a variety of IT services that are quite diverse from each other. This variety creates different perceptions of what cloud computing is among users. Despite this lack of uniformity, it is possible to classify cloud computing services offerings into three major categories: *Infrastructure-as-a-Service (IaaS)*, *Platform-as-a-Service (PaaS)*, and *Software-as-a-Service (SaaS)*. These categories are related to each other as described in Figure 1.5, which provides an organic view of cloud computing. We refer to this diagram as the *Cloud Computing Reference Model*, and we will use it throughout the

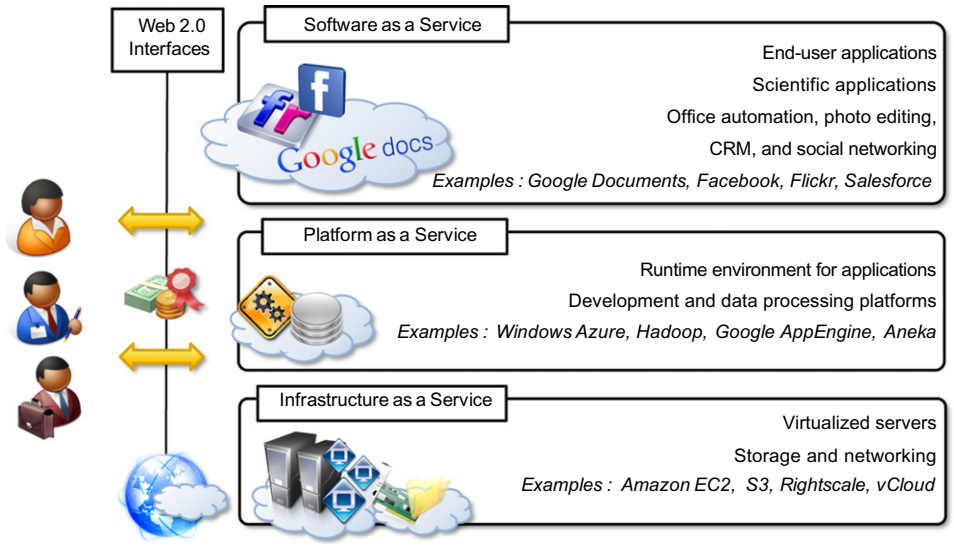

FIGURE 1.5

The Cloud Computing Reference Model.

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

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

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

At the top of the stack, *Software-as-a-Service* solutions provide applications and services on demand. Most of the common functionalities of desktop applications—such as office

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

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

1.1.5 Characteristics and benefits

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

- No up-front commitments
- On-demand access
- Nice pricing
- Simplified application acceleration and scalability
- Efficient resource allocation
- Energy efficiency
- Seamless creation and use of third-party services

The most evident benefit from the use of cloud computing systems and technologies is the increased economical return due to the reduced maintenance costs and *operational costs* related to IT software and infrastructure. This is mainly because IT assets, namely software and infrastructure, are turned into *utility costs*, which are paid for as long as they are used, not paid for up front. Capital costs are costs associated with assets that need to be paid in advance to start a business activity. Before cloud computing, IT infrastructure and software generated capital costs, since they were paid up front so that business start-ups could afford a computing infrastructure, enabling the business activities of the organization. The revenue of the business is then utilized to compensate over time for these costs. Organizations always minimize capital costs, since they are often associated with depreciable values. This is the case of hardware: a server bought today for $1,000 will have a market value less than its original price when it is eventually replaced by new hardware. To make profit, organizations have to compensate for this depreciation created by time, thus reducing the net gain obtained from revenue. Minimizing capital costs, then, is fundamental. Cloud computing transforms IT infrastructure and software into utilities, thus significantly contributing to increasing a company's net gain. Moreover, cloud computing also provides an opportunity for small organizations and start-ups: these do not need large investments to start their business, but they can comfortably grow with it. Finally, maintenance costs are significantly reduced: by renting the

infrastructure and the application services, organizations are no longer responsible for their maintenance. This task is the responsibility of the cloud service provider, who, thanks to economies of scale, can bear the maintenance costs.

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

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

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

1.1.6 Challenges ahead

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

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

addition, issues and challenges concerning the integration of real and virtual infrastructure need to be taken into account from different perspectives, such as security and legislation.

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

Legal issues may also arise. These are specifically tied to the ubiquitous nature of cloud computing, which spreads computing infrastructure across diverse geographical locations. Different legislation about privacy in different countries may potentially create disputes as to the rights that third parties (including government agencies) have to your data. U.S. legislation is known to give extreme powers to government agencies to acquire confidential data when there is the suspicion of operations leading to a threat to national security. European countries are more restrictive and protect the right of privacy. An interesting scenario comes up when a U.S. organization uses cloud services that store their data in Europe. In this case, should this organization be suspected by the government, it would become difficult or even impossible for the U.S. government to take control of the data stored in a cloud datacenter located in Europe.

1.2 Historical developments

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

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

1.2.1 Distributed systems

Clouds are essentially large distributed computing facilities that make available their services to third parties on demand. As a reference, we consider the characterization of a distributed system proposed by Tanenbaum et al. [1]:

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

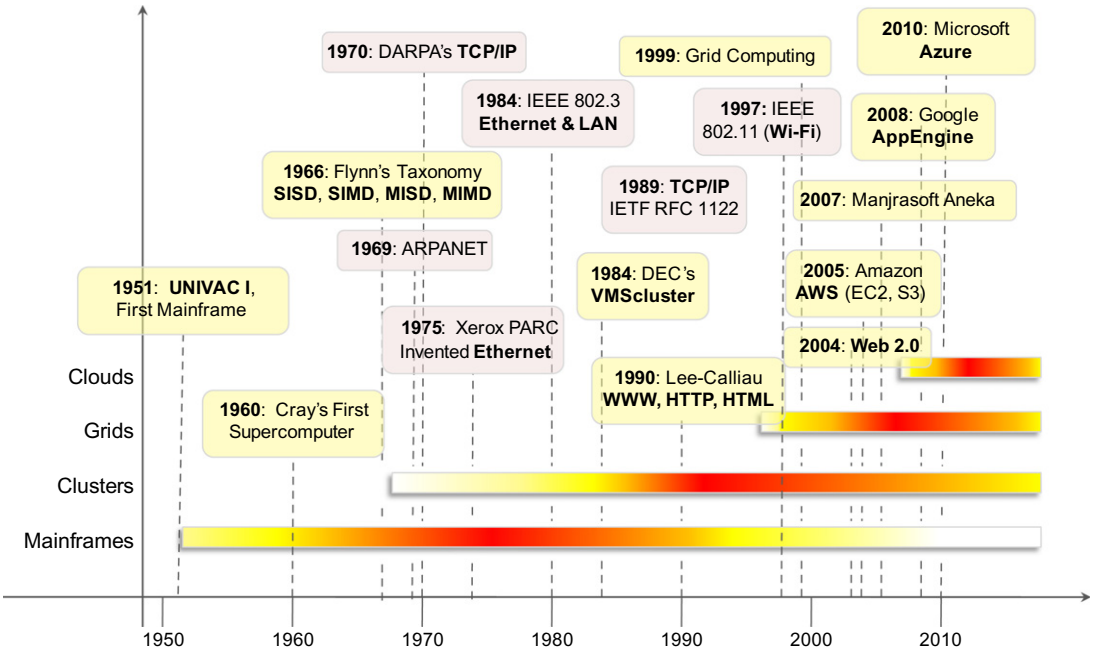

FIGURE 1.6

The evolution of distributed computing technologies, 1950s–2010s.

This is a general definition that includes a variety of computer systems, but it evidences two very important elements characterizing a distributed system: the fact that it is composed of multiple independent components and that these components are perceived as a single entity by users. This is particularly true in the case of cloud computing, in which clouds hide the complex architecture they rely on and provide a single interface to users. The primary purpose of distributed systems is to share resources and utilize them better. This is true in the case of cloud computing, where this concept is taken to the extreme and resources (infrastructure, runtime environments, and services) are rented to users. In fact, one of the driving factors of cloud computing has been the availability of the large computing facilities of IT giants (Amazon, Google) that found that offering their computing capabilities as a service provided opportunities to better utilize their infrastructure. Distributed systems often exhibit other properties such as *heterogeneity*, *openness*, *scalability*, *transparency*, *concurrency*, *continuous availability*, and *independent failures*. To some extent these also characterize clouds, especially in the context of scalability, concurrency, and continuous availability.

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

- *Mainframes*. These were the first examples of large computational facilities leveraging multiple processing units. Mainframes were powerful, highly reliable computers specialized for large

data movement and massive input/output (I/O) operations. They were mostly used by large organizations for bulk data processing tasks such as online transactions, enterprise resource planning, and other operations involving the processing of significant amounts of data. Even though mainframes cannot be considered distributed systems, they offered large computational power by using multiple processors, which were presented as a single entity to users. One of the most attractive features of mainframes was the ability to be highly reliable computers that were "always on" and capable of tolerating failures transparently. No system shutdown was required to replace failed components, and the system could work without interruption. Batch processing was the main application of mainframes. Now their popularity and deployments have reduced, but evolved versions of such systems are still in use for transaction processing (such as online banking, airline ticket booking, supermarket and telcos, and government services).

- *Clusters*. Cluster computing [3][4] started as a low-cost alternative to the use of mainframes and supercomputers. The technology advancement that created faster and more powerful mainframes and supercomputers eventually generated an increased availability of cheap commodity machines as a side effect. These machines could then be connected by a high-bandwidth network and controlled by specific software tools that manage them as a single system. Starting in the 1980s, clusters become the standard technology for parallel and high-performance computing. Built by commodity machines, they were cheaper than mainframes and made high-performance computing available to a large number of groups, including universities and small research labs. Cluster technology contributed considerably to the evolution of tools and frameworks for distributed computing, including Condor [5], Parallel Virtual Machine (PVM) [6], and Message Passing Interface (MPI) [7].[4] One of the attractive features of clusters was that the computational power of commodity machines could be leveraged to solve problems that were previously manageable only on expensive supercomputers. Moreover, clusters could be easily extended if more computational power was required.
- *Grids*. Grid computing [8] appeared in the early 1990s as an evolution of cluster computing. In an analogy to the power grid, grid computing proposed a new approach to access large computational power, huge storage facilities, and a variety of services. Users can "consume" resources in the same way as they use other utilities such as power, gas, and water. Grids initially developed as aggregations of geographically dispersed clusters by means of Internet connections. These clusters belonged to different organizations, and arrangements were made among them to share the computational power. Different from a "large cluster," a computing grid was a dynamic aggregation of heterogeneous computing nodes, and its scale was nationwide or even worldwide. Several developments made possible the diffusion of computing grids: (a) clusters became quite common resources; (b) they were often underutilized; (c) new problems were requiring computational power that went beyond the capability of single clusters; and (d) the improvements in networking and the diffusion of the Internet made possible long-distance, high-bandwidth connectivity. All these elements led to the development of grids, which now serve a multitude of users across the world.

[4] *MPI* is a specification for an API that allows many computers to communicate with one another. It defines a language-independent protocol that supports point-to-point and collective communication. *MPI* has been designed for high performance, scalability, and portability. At present, it is one of the dominant paradigms for developing parallel applications.

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

1.2.2 Virtualization

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

Virtualization is essentially a technology that allows creation of different computing environments. These environments are called *virtual* because they simulate the interface that is expected by a guest. The most common example of virtualization is *hardware virtualization*. This technology allows simulating the hardware interface expected by an operating system. Hardware virtualization allows the coexistence of different software stacks on top of the same hardware. These stacks are contained inside *virtual machine instances*, which operate in complete isolation from each other. High-performance servers can host several virtual machine instances, thus creating the opportunity to have a customized software stack on demand. This is the base technology that enables cloud computing solutions to deliver virtual servers on demand, such as Amazon EC2, RightScale, VMware vCloud, and others. Together with hardware virtualization, *storage* and *network virtualization* complete the range of technologies for the emulation of IT infrastructure.

Virtualization technologies are also used to replicate runtime environments for programs. Applications in the case of *process virtual machines* (which include the foundation of technologies such as Java or .NET), instead of being executed by the operating system, are run by a specific program called a *virtual machine*. This technique allows isolating the execution of applications and providing a finer control on the resource they access. Process virtual machines offer a higher level of abstraction with respect to hardware virtualization, since the guest is only constituted by an application rather than a complete software stack. This approach is used in cloud computing to provide a platform for scaling applications on demand, such as Google AppEngine and Windows Azure.

Having isolated and customizable environments with minor impact on performance is what makes virtualization a attractive technology. Cloud computing is realized through platforms that leverage the basic concepts described above and provides on demand virtualization services to a multitude of users across the globe.

1.2.3 Web 2.0

The Web is the primary interface through which cloud computing delivers its services. At present, the Web encompasses a set of technologies and services that facilitate interactive information sharing, collaboration, user-centered design, and application composition. This evolution has transformed the Web into a rich platform for application development and is known as *Web 2.0.* This term captures a new way in which developers architect applications and deliver services through the Internet and provides new experience for users of these applications and services.

Web 2.0 brings *interactivity* and *flexibility* into Web pages, providing enhanced user experience by gaining Web-based access to all the functions that are normally found in desktop applications. These capabilities are obtained by integrating a collection of standards and technologies such as *XML*, *Asynchronous JavaScript and XML (AJAX)*, *Web Services*, and others. These technologies allow us to build applications leveraging the contribution of users, who now become providers of content. Furthermore, the capillary diffusion of the Internet opens new opportunities and markets for the Web, the services of which can now be accessed from a variety of devices: mobile phones, car dashboards, TV sets, and others. These new scenarios require an increased dynamism for applications, which is another key element of this technology. Web 2.0 applications are extremely dynamic: they improve continuously, and new updates and features are integrated at a constant rate by following the usage trend of the community. There is no need to deploy new software releases on the installed base at the client side. Users can take advantage of the new software features simply by interacting with cloud applications. Lightweight deployment and programming models are very important for effective support of such dynamism. Loose coupling is another fundamental property. New applications can be "synthesized" simply by composing existing services and integrating them, thus providing added value. This way it becomes easier to follow the interests of users. Finally, Web 2.0 applications aim to leverage the "long tail" of Internet users by making themselves available to everyone in terms of either media accessibility or affordability.

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

This idea of the Web as a transport that enables and enhances interaction was introduced in 1999 by Darcy DiNucci[5] and started to become fully realized in 2004. Today it is a mature platform for supporting the needs of cloud computing, which strongly leverages Web 2.0. Applications

[5]In a column for *Design & New Media* magazine, Darci DiNucci describes the Web as follows: "The Web we know now, which loads into a browser window in essentially static screenfulls, is only an embryo of the Web to come. The first glimmerings of Web 2.0 are beginning to appear, and we are just starting to see how that embryo might develop. The Web will be understood not as screenfulls of text and graphics but as a transport mechanism, the ether through which interactivity happens. It will [...] appear on your computer screen, [...] on your TV set [...], your car dashboard [...], your cell phone [...], hand-held game machines [...], maybe even your microwave oven."

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

1.2.4 Service-oriented computing

Service orientation is the core reference model for cloud computing systems. This approach adopts the concept of services as the main building blocks of application and system development. *Service-oriented computing (SOC)* supports the development of rapid, low-cost, flexible, interoperable, and evolvable applications and systems [19].

A *service* is an abstraction representing a self-describing and platform-agnostic component that can perform any function—anything from a simple function to a complex business process. Virtually any piece of code that performs a task can be turned into a service and expose its functionalities through a network-accessible protocol. A service is supposed to be *loosely coupled*, *reusable*, *programming language independent*, and *location transparent*. Loose coupling allows services to serve different scenarios more easily and makes them reusable. Independence from a specific platform increases services accessibility. Thus, a wider range of clients, which can look up services in global registries and consume them in a location-transparent manner, can be served. Services are composed and aggregated into a *service-oriented architecture (SOA)* [27], which is a logical way of organizing software systems to provide end users or other entities distributed over the network with services through published and discoverable interfaces.

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

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

One of the most popular expressions of service orientation is represented by Web Services (WS) [21]. These introduce the concepts of SOC into the World Wide Web, by making it consumable by applications and not only humans. Web services are software components that expose functionalities accessible using a method invocation pattern that goes over the HyperText Transfer Protocol (HTTP). The interface of a Web service can be programmatically inferred by metadata expressed through the *Web Service Description Language (WSDL)* [22]; this is an XML language that defines the characteristics of the service and all the methods, together with parameters, descriptions, and return type, exposed by the service. The interaction with Web services happens through *Simple Object Access Protocol (SOAP)* [23]. This is an XML language that defines how to invoke a Web service method and collect the result. Using SOAP and WSDL over HTTP, Web services become platform independent and accessible to the World Wide Web. The standards and specifications concerning Web services are controlled by the World Wide Web Consortium (W3C). Among the most popular architectures for developing Web services we can note ASP.NET [24] and Axis [25].

The development of systems in terms of distributed services that can be composed together is the major contribution given by SOC to the realization of cloud computing. Web services technologies have provided the right tools to make such composition straightforward and easily integrated with the mainstream World Wide Web (WWW) environment.

1.2.5 Utility-oriented computing

Utility computing is a vision of computing that defines a service-provisioning model for compute services in which resources such as storage, compute power, applications, and infrastructure are packaged and offered on a pay-per-use basis. The idea of providing computing as a *utility* like natural gas, water, power, and telephone connection has a long history but has become a reality today with the advent of cloud computing. Among the earliest forerunners of this vision we can include the American scientist John McCarthy, who, in a speech for the Massachusetts Institute of Technology (MIT) centennial in 1961, observed:

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

The first traces of this service-provisioning model can be found in the mainframe era. IBM and other mainframe providers offered mainframe power to organizations such as banks and government agencies throughout their datacenters. The business model introduced with utility computing brought new requirements and led to improvements in mainframe technology: additional features such as operating systems, process control, and user-metering facilities. The idea of computing as utility remained and extended from the business domain to academia with the advent of cluster computing. Not only businesses but also research institutes became acquainted with the idea of leveraging an external IT infrastructure on demand. Computational science, which was one of the major driving factors for building computing clusters, still required huge compute power for addressing "Grand Challenge" problems, and not all the institutions were able to satisfy their computing needs internally. Access to external clusters still remained a common practice. The capillary diffusion of the Internet and the Web provided the technological means to realize utility computing on a

worldwide scale and through simple interfaces. As already discussed, computing grids provided a planet-scale distributed computing infrastructure that was accessible on demand. Computing grids brought the concept of utility computing to a new level: market orientation [15]. With utility computing accessible on a wider scale, it is easier to provide a trading infrastructure where grid products—storage, computation, and services—are bid for or sold. Moreover, e-commerce technologies [25] provided the infrastructure support for utility computing. In the late 1990s a significant interest in buying any kind of good online spread to the wider public: food, clothes, multimedia products, and online services such as storage space and Web hosting. After the *dot-com bubble*[6] burst, this interest reduced in size, but the phenomenon made the public keener to buy online services. As a result, infrastructures for online payment using credit cards become easily accessible and well proven.

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

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

1.3 Building cloud computing environments

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

1.3.1 Application development

Applications that leverage cloud computing benefit from its capability to dynamically scale on demand. One class of applications that takes the biggest advantage of this feature is that of *Web applications*. Their performance is mostly influenced by the workload generated by varying user demands. With the diffusion of Web 2.0 technologies, the Web has become a platform for developing rich and complex applications, including *enterprise applications* that now leverage the Internet as the preferred channel for service delivery and user interaction. These applications are

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

characterized by complex processes that are triggered by the interaction with users and develop through the interaction between several tiers behind the Web front end. These are the applications that are mostly sensible to inappropriate sizing of infrastructure and service deployment or variability in workload.

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

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

1.3.2 Infrastructure and system development

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

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

Web 2.0 technologies constitute the interface through which cloud computing services are delivered, managed, and provisioned. Besides the interaction with rich interfaces through the Web browser, Web services have become the primary access point to cloud computing systems from a

programmatic standpoint. Therefore, service orientation is the underlying paradigm that defines the architecture of a cloud computing system. Cloud computing is often summarized with the acronym *XaaS—Everything-as-a-Service*—that clearly underlines the central role of service orientation. Despite the absence of a unique standard for accessing the resources serviced by different cloud providers, the commonality of technology smoothes the learning curve and simplifies the integration of cloud computing into existing systems.

Virtualization is another element that plays a fundamental role in cloud computing. This technology is a core feature of the infrastructure used by cloud providers. As discussed before, the virtualization concept is more than 40 years old, but cloud computing introduces new challenges, especially in the management of virtual environments, whether they are abstractions of virtual hardware or a runtime environment. Developers of cloud applications need to be aware of the limitations of the selected virtualization technology and the implications on the volatility of some components of their systems.

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

1.3.3 Computing platforms and technologies

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

1.3.3.1 Amazon web services (AWS)

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

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

1.3.3.2 Google AppEngine

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

1.3.3.3 Microsoft Azure

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

1.3.3.4 Hadoop

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

1.3.3.5 Force.com and Salesforce.com

Force.com is a cloud computing platform for developing social enterprise applications. The platform is the basis for *SalesForce.com*, a Software-as-a-Service solution for customer relationship management. Force.com allows developers to create applications by composing ready-to-use blocks; a complete set of components supporting all the activities of an enterprise are available. It is also possible to develop your own components or integrate those available in *AppExchange* into your applications. The platform provides complete support for developing applications, from the

design of the data layout to the definition of business rules and workflows and the definition of the user interface. The Force.com platform is completely hosted on the cloud and provides complete access to its functionalities and those implemented in the hosted applications through Web services technologies.

1.3.3.6 Manjrasoft Aneka

Manjrasoft Aneka [165] is a cloud application platform for rapid creation of scalable applications and their deployment on various types of clouds in a seamless and elastic manner. It supports a collection of programming abstractions for developing applications and a distributed runtime environment that can be deployed on heterogeneous hardware (clusters, networked desktop computers, and cloud resources). Developers can choose different abstractions to design their application: *tasks*, *distributed threads*, and *map-reduce*. These applications are then executed on the distributed service-oriented runtime environment, which can dynamically integrate additional resource on demand. The service-oriented architecture of the runtime has a great degree of flexibility and simplifies the integration of new features, such as abstraction of a new programming model and associated execution management environment. Services manage most of the activities happening at runtime: scheduling, execution, accounting, billing, storage, and quality of service.

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

SUMMARY

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

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

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

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

Review questions

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

This page intentionally left blank

CHAPTER

Principles of Parallel and Distributed Computing

2

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

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

2.1 Eras of computing

The two fundamental and dominant models of computing are *sequential* and *parallel*. The sequential computing era began in the 1940s; the parallel (and distributed) computing era followed it within a decade (see Figure 2.1). The four key elements of computing developed during these eras are *architectures, compilers, applications*, and *problem-solving environments*.

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

2.2 Parallel vs. distributed computing

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

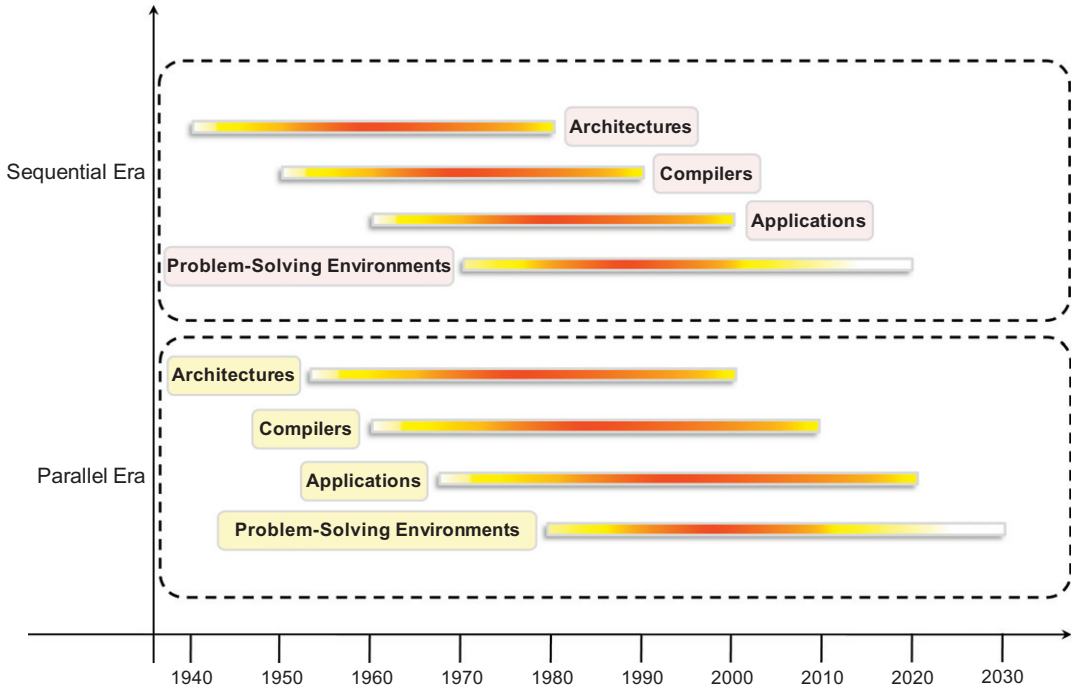

FIGURE 2.1

Eras of computing, 1940s–2030s.

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

The term *distributed computing* encompasses any architecture or system that allows the computation to be broken down into units and executed concurrently on different computing elements, whether these are processors on different nodes, processors on the same computer, or cores within the same processor. Therefore, distributed computing includes a wider range of systems and applications than parallel computing and is often considered a more general term. Even though it is not

a rule, the term *distributed* often implies that the locations of the computing elements are not the same and such elements might be heterogeneous in terms of hardware and software features. Classic examples of distributed computing systems are computing grids or Internet computing systems, which combine together the biggest variety of architectures, systems, and applications in the world.

2.3 Elements of parallel computing

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

2.3.1 What is parallel processing?

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

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

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

- Computational requirements are ever increasing in the areas of both scientific and business computing. The technical computing problems, which require high-speed computational power, are related to life sciences, aerospace, geographical information systems, mechanical design and analysis, and the like.
- Sequential architectures are reaching physical limitations as they are constrained by the speed of light and thermodynamics laws. The speed at which sequential CPUs can operate is reaching saturation point (no more vertical growth), and hence an alternative way to get high computational speed is to connect multiple CPUs (opportunity for horizontal growth).

- Hardware improvements in pipelining, superscalar, and the like are nonscalable and require sophisticated compiler technology. Developing such compiler technology is a difficult task.
- Vector processing works well for certain kinds of problems. It is suitable mostly for scientific problems (involving lots of matrix operations) and graphical processing. It is not useful for other areas, such as databases.
- The technology of parallel processing is mature and can be exploited commercially; there is already significant R&D work on development tools and environments.
- Significant development in networking technology is paving the way for heterogeneous computing.

2.3.2 Hardware architectures for parallel processing

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

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

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

An SISD computing system is a uniprocessor machine capable of executing a single instruction, which operates on a single data stream (see Figure 2.2). In SISD, machine instructions are processed sequentially; hence computers adopting this model are popularly called *sequential computers*. Most conventional computers are built using the SISD model. All the instructions and data to be processed have to be stored in primary memory. The speed of the processing element in the SISD model is limited by the rate at which the computer can transfer information internally. Dominant representative SISD systems are IBM PC, Macintosh, and workstations.

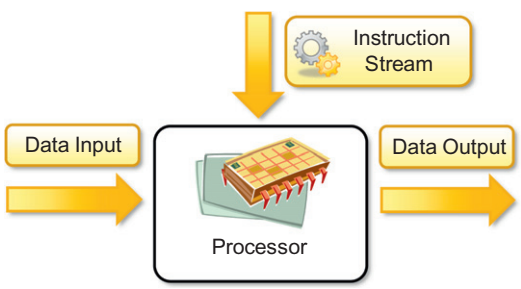

FIGURE 2.2

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

2.3.2.2 Single-instruction, multiple-data (SIMD) systems

An SIMD computing system is a multiprocessor machine capable of executing the same instruction on all the CPUs but operating on different data streams (see Figure 2.3). Machines based on an SIMD model are well suited to scientific computing since they involve lots of vector and matrix operations. For instance, statements such as

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

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

2.3.2.3 Multiple-instruction, single-data (MISD) systems

An MISD computing system is a multiprocessor machine capable of executing different instructions on different PEs but all of them operating on the same data set (see Figure 2.4). For instance, statements such as

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

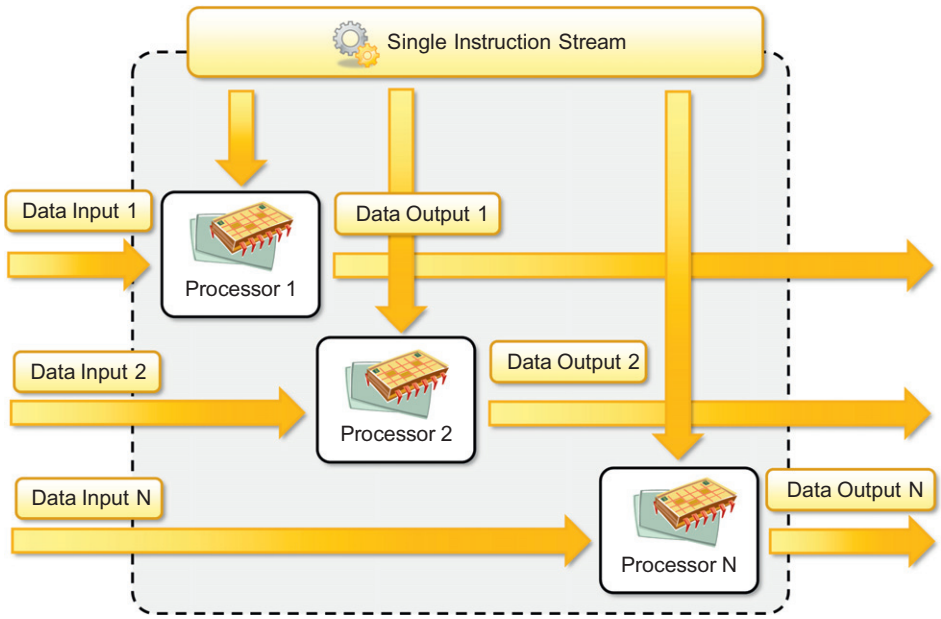

FIGURE 2.3

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

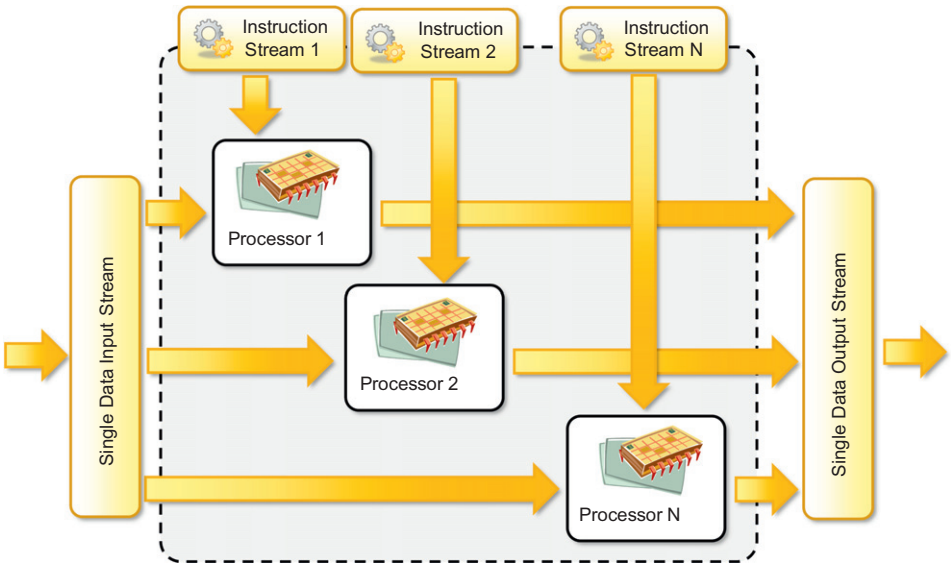

FIGURE 2.4

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

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

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

An MIMD computing system is a multiprocessor machine capable of executing multiple instructions on multiple data sets (see Figure 2.5). Each PE in the MIMD model has separate instruction and data streams; hence machines built using this model are well suited to any kind of application. Unlike SIMD and MISD machines, PEs in MIMD machines work asynchronously.

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

Shared memory MIMD machines

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

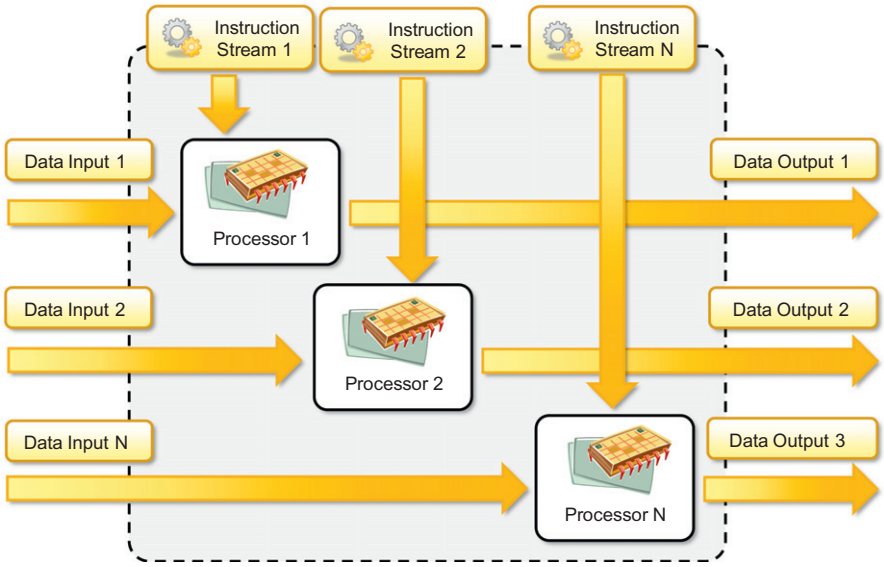

FIGURE 2.5

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

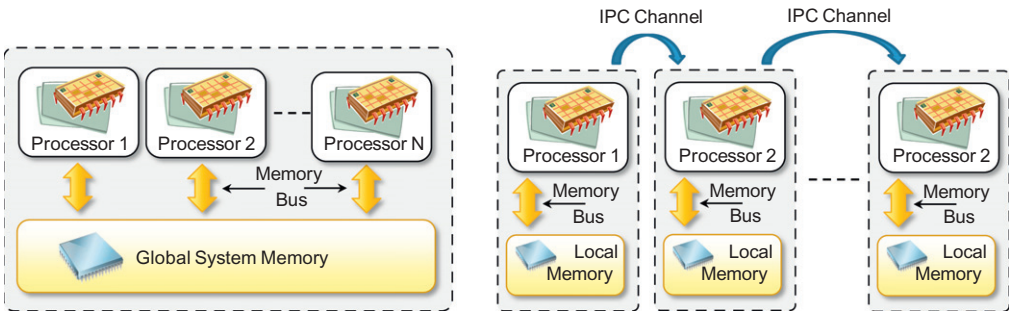

FIGURE 2.6

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

Distributed memory MIMD machines

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

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

2.3.3 Approaches to parallel programming

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

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

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

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

2.3.4 Levels of parallelism

Levels of parallelism are decided based on the lumps of code (grain size) that can be a potential candidate for parallelism. Table 2.1 lists categories of code granularity for parallelism. All these approaches have a common goal: to boost processor efficiency by hiding latency. To conceal latency, there must be another thread ready to run whenever a lengthy operation occurs. The idea is to execute concurrently two or more single-threaded applications, such as compiling, text formatting, database searching, and device simulation.

As shown in the table and depicted in Figure 2.7, parallelism within an application can be detected at several levels:

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

Table 2.1 Levels of Parallelism

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

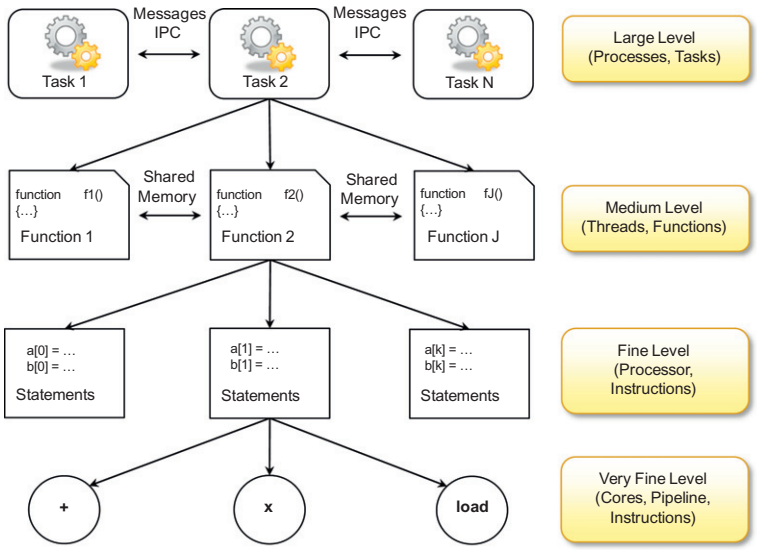

FIGURE 2.7

Levels of parallelism in an application.

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

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

2.3.5 Laws of caution

Now that we have introduced some general aspects of parallel computing in terms of architectures and models, we can make some considerations that have been drawn from experience designing and implementing such systems. These considerations are guidelines that can help us understand

how much benefit an application or a software system can gain from parallelism. In particular, what we need to keep in mind is that parallelism is used to perform multiple activities together so that the system can increase its throughput or its speed. But the relations that control the increment of speed are not linear. For example, for a given n processors, the user expects speed to be increased by n times. This is an ideal situation, but it rarely happens because of the communication overhead.

Here are two important guidelines to take into account:

- Speed of computation is proportional to the square root of system cost; they never increase linearly. Therefore, the faster a system becomes, the more expensive it is to increase its speed (Figure 2.8).
- Speed by a parallel computer increases as the logarithm of the number of processors (i.e., $y = k^*log(N)$). This concept is shown in Figure 2.9.

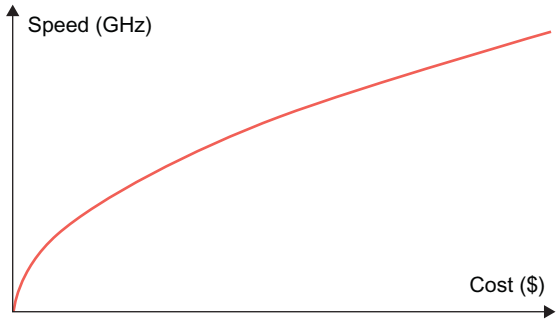

FIGURE 2.8

Cost versus speed.

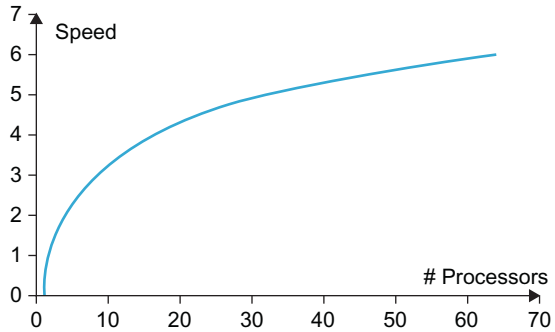

FIGURE 2.9

Number processors versus speed.

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

2.4 Elements of distributed computing

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

2.4.1 General concepts and definitions

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

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

This definition is general enough to include various types of distributed computing systems that are especially focused on unified usage and aggregation of distributed resources. In this chapter, we focus on the architectural models that are used to harness independent computers and present them as a whole coherent system. Communication is another fundamental aspect of distributed computing. Since distributed systems are composed of more than one computer that collaborate together, it is necessary to provide some sort of data and information exchange between them, which generally occurs through the network (Coulouris et al. [2]):

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

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

2.4.2 Components of a distributed system

A distributed system is the result of the interaction of several components that traverse the entire computing stack from hardware to software. It emerges from the collaboration of several elements that—by working together—give users the illusion of a single coherent system. Figure 2.10 provides an overview of the different layers that are involved in providing the services of a distributed system.

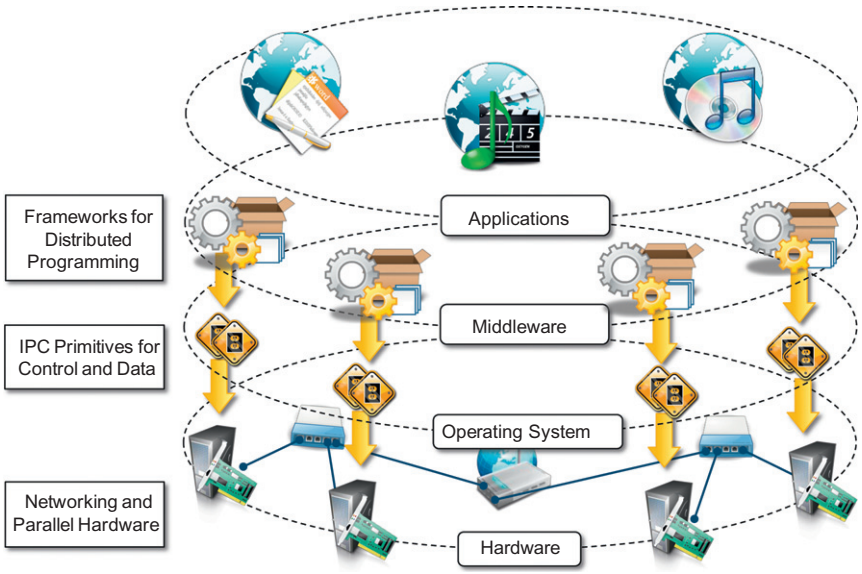

FIGURE 2.10

A layered view of a distributed system.

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

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

The middleware layer leverages such services to build a uniform environment for the development and deployment of distributed applications. This layer supports the programming paradigms for distributed systems, which we will discuss in Chapters 5–7 of this book. By relying on the services offered by the operating system, the middleware develops its own protocols, data formats, and programming language or frameworks for the development of distributed applications. All of them constitute a uniform interface to distributed application developers that is completely independent from the underlying operating system and hides all the heterogeneities of the bottom layers.

The top of the distributed system stack is represented by the applications and services designed and developed to use the middleware. These can serve several purposes and often expose their

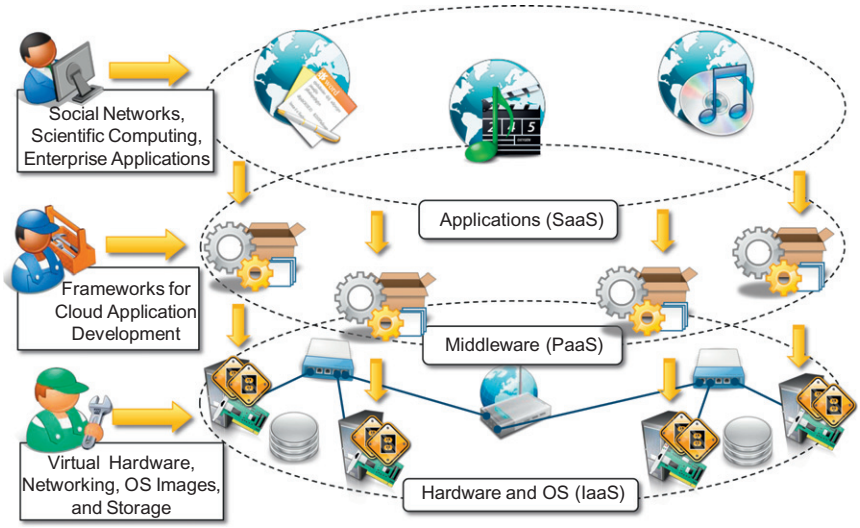

FIGURE 2.11

A cloud computing distributed system.

features in the form of graphical user interfaces (GUIs) accessible locally or through the Internet via a Web browser. For example, in the case of a cloud computing system, the use of Web technologies is strongly preferred, not only to interface distributed applications with the end user but also to provide platform services aimed at building distributed systems. A very good example is constituted by Infrastructure-as-a-Service (IaaS) providers such as Amazon Web Services (AWS), which provide facilities for creating virtual machines, organizing them together into a cluster, and deploying applications and systems on top. Figure 2.11 shows an example of how the general reference architecture of a distributed system is contextualized in the case of a cloud computing system.

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

2.4.3 Architectural styles for distributed computing

Although a distributed system comprises the interaction of several layers, the middleware layer is the one that enables distributed computing, because it provides a coherent and uniform runtime environment for applications. There are many different ways to organize the components that, taken together, constitute such an environment. The interactions among these components and their

responsibilities give structure to the middleware and characterize its type or, in other words, define its architecture. Architectural styles [104] aid in understanding and classifying the organization of software systems in general and distributed computing in particular.

> *Architectural styles are mainly used to determine the vocabulary of components and connectors that are used as instances of the style together with a set of constraints on how they can be combined [105].*

Design patterns [106] help in creating a common knowledge within the community of software engineers and developers as to how to structure the relations of components within an application and understand the internal organization of software applications. Architectural styles do the same for the overall architecture of software systems. In this section, we introduce the most relevant architectural styles for distributed computing and focus on the components and connectors that make each style peculiar. Architectural styles for distributed systems are helpful in understanding the different roles of components in the system and how they are distributed across multiple machines. We organize the architectural styles into two major classes:

- Software architectural styles
- System architectural styles

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

2.4.3.1 Component and connectors

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

2.4.3.2 Software architectural styles

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

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

Data centered architectures

These architectures identify the data as the fundamental element of the software system, and access to shared data is the core characteristic of the data-centered architectures. Therefore, especially

Table 2.2 Software Architectural Styles

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

within the context of distributed and parallel computing systems, integrity of data is the overall goal for such systems.

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

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

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

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

Data-flow architectures

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

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

Pipe-and-Filter Style. The *pipe-and-filter style* is a variation of the previous style for expressing the activity of a software system as sequence of data transformations. Each component of the processing chain is called a *filter,* and the connection between one filter and the next is represented by a data stream. With respect to the batch sequential style, data is processed incrementally and each filter processes the data as soon as it is available on the input stream. As soon as one filter produces a consumable amount of data, the next filter can start its processing. Filters generally do not have state, know the identity of neither the previous nor the next filter, and they are connected with in-memory data structures such as first-in/first-out (FIFO) buffers or other structures. This particular sequencing is called *pipelining* and introduces concurrency in the execution of the filters. A classic example of this architecture is the microprocessor pipeline, whereby multiple instructions are executed at the same time by completing a different phase of each of them. We can identify the phases of the instructions as the filters, whereas the data streams are represented by the registries that are shared within the processors. Another example are the Unix shell pipes (i.e., *cat <filename>| grep<pattern>| wc −l*), where the filters are the single shell programs composed together and the connections are their input and output streams that are chained together. Applications of this architecture can also be found in the compiler design (e.g., the lex/yacc model is based on a pipe of the following phases: *scanning | parsing | semantic analysis | code generation*), image and signal processing, and voice and video streaming.

Data-flow architectures are optimal when the system to be designed embodies a multistage process, which can be clearly identified into a collection of separate components that need to be orchestrated together. Within this reference scenario, components have well-defined interfaces exposing input and output ports, and the connectors are represented by the datastreams between these ports. The main differences between the two subcategories are reported in Table 2.3.

Virtual machine architectures

The virtual machine class of architectural styles is characterized by the presence of an abstract execution environment (generally referred as a *virtual machine*) that simulates features that are not available in the hardware or software. Applications and systems are implemented on top of this layer and become portable over different hardware and software environments as long as there is

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

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

an implementation of the virtual machine they interface with. The general interaction flow for systems implementing this pattern is the following: the program (or the application) defines its operations and state in an abstract format, which is interpreted by the virtual machine engine. The interpretation of a program constitutes its execution. It is quite common in this scenario that the engine maintains an internal representation of the program state. Very popular examples within this category are rule-based systems, interpreters, and command-language processors.

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

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

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

virtual machine closes the gap between the end-user abstractions and the software/hardware environment in which such abstractions are executed.

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

Call & return architectures

This category identifies all systems that are organised into components mostly connected together by method calls. The activity of systems modeled in this way is characterized by a chain of method calls whose overall execution and composition identify the execution of one or more operations. The internal organization of components and their connections may vary. Nonetheless, it is possible to identify three major subcategories, which differentiate by the way the system is structured and how methods are invoked: top-down style, object-oriented style, and layered style.

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

Object-Oriented Style. This architectural style encompasses a wide range of systems that have been designed and implemented by leveraging the abstractions of object-oriented programming (OOP). Systems are specified in terms of classes and implemented in terms of objects. Classes define the type of components by specifying the data that represent their state and the operations that can be done over these data. One of the main advantages over the top-down style is that there is a coupling between data and operations used to manipulate them. Object instances become responsible for hiding their internal state representation and for protecting its integrity while providing operations to other components. This leads to a better decomposition process and more manageable systems. Disadvantages of this style are mainly two: each object needs to know the identity of an object if it wants to invoke operations on it, and shared objects need to be carefully designed in order to ensure the consistency of their state.

Layered Style. The layered system style allows the design and implementation of software systems in terms of layers, which provide a different level of abstraction of the system. Each layer generally operates with at most two layers: the one that provides a lower abstraction level and the one that provides a higher abstraction layer. Specific protocols and interfaces define how adjacent layers interact. It is possible to model such systems as a stack of layers, one for each level of abstraction. Therefore, the components are the layers and the connectors are the interfaces and

protocols used between adjacent layers. A user or client generally interacts with the layer at the highest abstraction, which, in order to carry its activity, interacts and uses the services of the lower layer. This process is repeated (if necessary) until the lowest layer is reached. It is also possible to have the opposite behavior: events and callbacks from the lower layers can trigger the activity of the higher layer and propagate information up through the stack. The advantages of the layered style are that, as happens for the object-oriented style, it supports a modular design of systems and allows us to decompose the system according to different levels of abstractions by encapsulating together all the operations that belong to a specific level. Layers can be replaced as long as they are compliant with the expected protocols and interfaces, thus making the system flexible. The main disadvantage is constituted by the lack of extensibility, since it is not possible to add layers without changing the protocols and the interfaces between layers.[2] This also makes it complex to add operations. Examples of layered architectures are the modern operating system kernels and the International Standards Organization/Open Systems Interconnection (ISO/OSI) or the TCP/IP stack.

Architectural styles based on independent components

This class of architectural style models systems in terms of independent components that have their own life cycles, which interact with each other to perform their activities. There are two major categories within this class—communicating processes and event systems—which differentiate in the way the interaction among components is managed.

Communicating Processes. In this architectural style, components are represented by independent processes that leverage IPC facilities for coordination management. This is an abstraction that is quite suitable to modeling distributed systems that, being distributed over a network of computing nodes, are necessarily composed of several concurrent processes. Each of the processes provides other processes with services and can leverage the services exposed by the other processes. The conceptual organization of these processes and the way in which the communication happens vary according to the specific model used, either peer-to-peer or client/server.[3] Connectors are identified by IPC facilities used by these processes to communicate.

Event Systems. In this architectural style, the components of the system are loosely coupled and connected. In addition to exposing operations for data and state manipulation, each component also publishes (or announces) a collection of events with which other components can register. In general, other components provide a callback that will be executed when the event is activated. During the activity of a component, a specific runtime condition can activate one of the exposed events, thus triggering the execution of the callbacks registered with it. Event activation may be accompanied by contextual information that can be used in the callback to handle the event. This information can be passed as an argument to the callback or by using some shared repository between components. Event-based systems have become quite popular, and support for their implementation is provided either at the API level or the programming language level.[4] The main

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

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

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

advantage of such an architectural style is that it fosters the development of open systems: new modules can be added and easily integrated into the system as long as they have compliant interfaces for registering to the events. This architectural style solves some of the limitations observed for the top-down and object-oriented styles. First, the invocation pattern is implicit, and the connection between the caller and the callee is not hard-coded; this gives a lot of flexibility since addition or removal of a handler to events can be done without changes in the source code of applications. Second, the event source does not need to know the identity of the event handler in order to invoke the callback. The disadvantage of such a style is that it relinquishes control over system computation. When a component triggers an event, it does not know how many event handlers will be invoked and whether there are any registered handlers. This information is available only at runtime and, from a static design point of view, becomes more complex to identify the connections among components and to reason about the correctness of the interactions.

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

2.4.3.3 System architectural styles

System architectural styles cover the physical organization of components and processes over a distributed infrastructure. They provide a set of reference models for the deployment of such systems and help engineers not only have a common vocabulary in describing the physical layout of systems but also quickly identify the major advantages and drawbacks of a given deployment and whether it is applicable for a specific class of applications. In this section, we introduce two fundamental reference styles: *client/server* and *peer-to-peer*.

Client/server

This architecture is very popular in distributed computing and is suitable for a wide variety of applications. As depicted in Figure 2.12, the client/server model features two major components: a *server* and a *client*. These two components interact with each other through a network connection using a given protocol. The communication is unidirectional: The client issues a request to the server, and after processing the request the server returns a response. There could be multiple client components issuing requests to a server that is passively waiting for them. Hence, the important operations in the client-server paradigm are *request*, *accept* (client side), and *listen* and *response* (server side).

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

- *Thin-client model.* In this model, the load of data processing and transformation is put on the server side, and the client has a light implementation that is mostly concerned with retrieving and returning the data it is being asked for, with no considerable further processing.

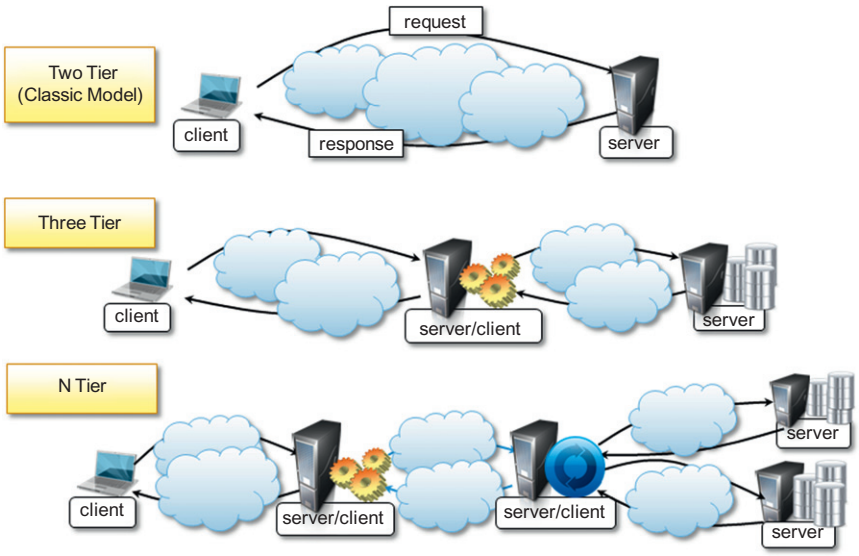

FIGURE 2.12

Client/server architectural styles.

- *Fat-client model.* In this model, the client component is also responsible for processing and transforming the data before returning it to the user, whereas the server features a relatively light implementation that is mostly concerned with the management of access to the data.

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

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

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

dimension of the data to maintain, manage, and access, which might be prohibitive for a single computation node or too large for serving the clients with satisfactory performance.

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

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

Peer-to-peer

The peer-to-peer model, depicted in Figure 2.13, introduces a symmetric architecture in which all the components, called *peers*, play the same role and incorporate both client and server capabilities of the client/server model. More precisely, each peer acts as a *server* when it processes requests from other peers and as a *client* when it issues requests to other peers. With respect to the client/server model that partitions the responsibilities of the IPC between server and clients, the peer-to-peer model attributes the same responsibilities to each component. Therefore, this model is quite suitable for highly decentralized architecture, which can scale better along the dimension of the number of peers. The disadvantage of this approach is that the management of the implementation of algorithms is more complex than in the client/server model.

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

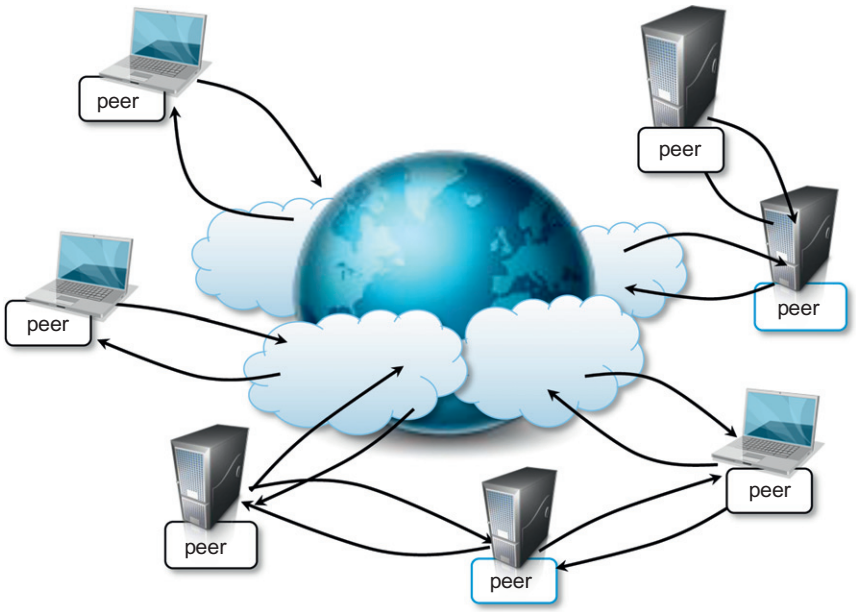

FIGURE 2.13

Peer-to-peer architectural style.

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

2.4.4 Models for interprocess communication

Distributed systems are composed of a collection of concurrent processes interacting with each other by means of a network connection. Therefore, IPC is a fundamental aspect of distributed systems design and implementation. IPC is used to either exchange data and information or coordinate the activity of processes. IPC is what ties together the different components of a distributed system, thus making them act as a single system. There are several different models in which processes can interact with each other; these map to different abstractions for IPC. Among the most relevant that we can mention are shared memory, remote procedure call (RPC), and message passing. At a lower level, IPC is realized through the fundamental tools of network programming. Sockets are the most popular IPC primitive for implementing communication channels between distributed processes.

They facilitate interaction patterns that, at the lower level, mimic the client/server abstraction and are based on a request-reply communication model. Sockets provide the basic capability of transferring a sequence of bytes, which is converted at higher levels into a more meaningful representation (such as procedure parameters or return values or messages). Such a powerful abstraction allows system engineers to concentrate on the logic-coordinating distributed components and the information they exchange rather than the networking details. These two elements identify the model for IPC. In this section, we introduce the most important reference model for architecting the communication among processes.

2.4.4.1 Message-based communication

The abstraction of *message* has played an important role in the evolution of the models and technologies enabling distributed computing. Couloris et al. [2] define a distributed system as "one in which components located at networked computers communicate and coordinate their actions only by passing messages." The term *message*, in this case, identifies any discrete amount of information that is passed from one entity to another. It encompasses any form of data representation that is limited in size and time, whereas this is an invocation to a remote procedure or a serialized object instance or a generic message. Therefore, the term *message-based communication model* can be used to refer to any model for IPC discussed in this section, which does not necessarily rely on the abstraction of data streaming.

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

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

independent from the existence of requests. Examples of distributed object infrastructures are *Common Object Request Broker Architecture (CORBA)*, *Component Object Model* (*COM, DCOM,* and *COM+*), *Java Remote Method Invocation (RMI)*, and *.NET Remoting*.

- *Distributed agents and active objects.* Programming paradigms based on agents and active objects involve by definition the presence of instances, whether they are agents of objects, despite the existence of requests. This means that objects have their own control thread, which allows them to carry out their activity. These models often make explicit use of messages to trigger the execution of methods, and a more complex semantics is attached to the messages.
- *Web services.* Web service technology provides an implementation of the RPC concept over HTTP, thus allowing the interaction of components that are developed with different technologies. A Web service is exposed as a remote object hosted on a Web server, and method invocations are transformed in HTTP requests, opportunely packaged using specific protocols such as *Simple Object Access Protocol (SOAP)* or *Representational State Transfer (REST)*.

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

2.4.4.2 Models for message-based communication

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

Point-to-point message model

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

Publish-and-subscribe message model

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

be available to all the subscribers that registered for the corresponding event. There are two major strategies for dispatching the event to the subscribers:

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

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

Request-reply message model

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

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

2.5 Technologies for distributed computing

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

2.5.1 Remote procedure call

RPC is the fundamental abstraction enabling the execution of procedures on client's request. RPC allows extending the concept of a procedure call beyond the boundaries of a process and a single memory address space. The called procedure and calling procedure may be on the same system or they may be on different systems in a network. The concept of RPC has been discussed since 1976 and completely formalized by Nelson [111] and Birrell [112] in the early 1980s. From there on, it has not changed in its major components. Even though it is a quite old technology, RPC is still used today as a fundamental component for IPC in more complex systems.

Figure 2.14 illustrates the major components that enable an RPC system. The system is based on a client/server model. The server process maintains a registry of all the available procedures that

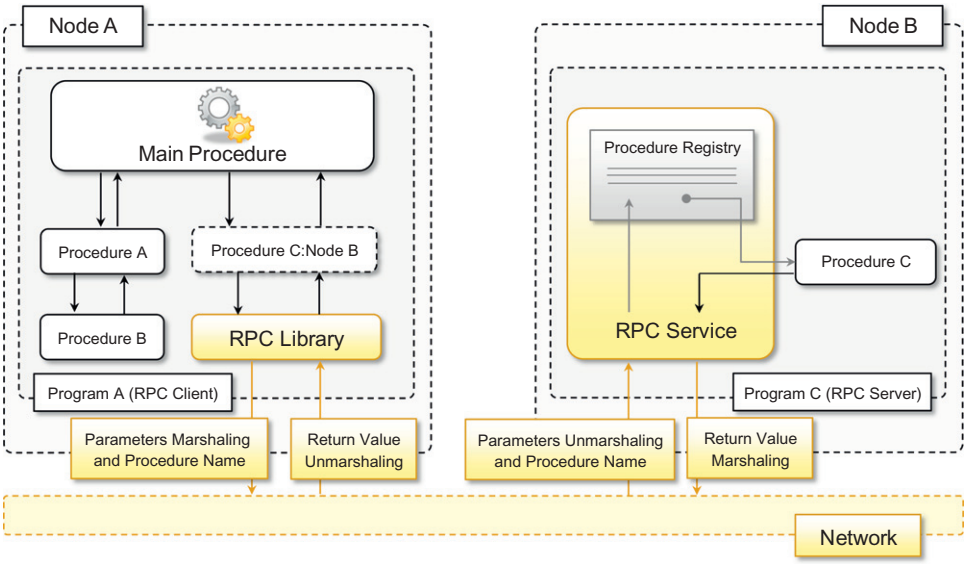

FIGURE 2.14

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

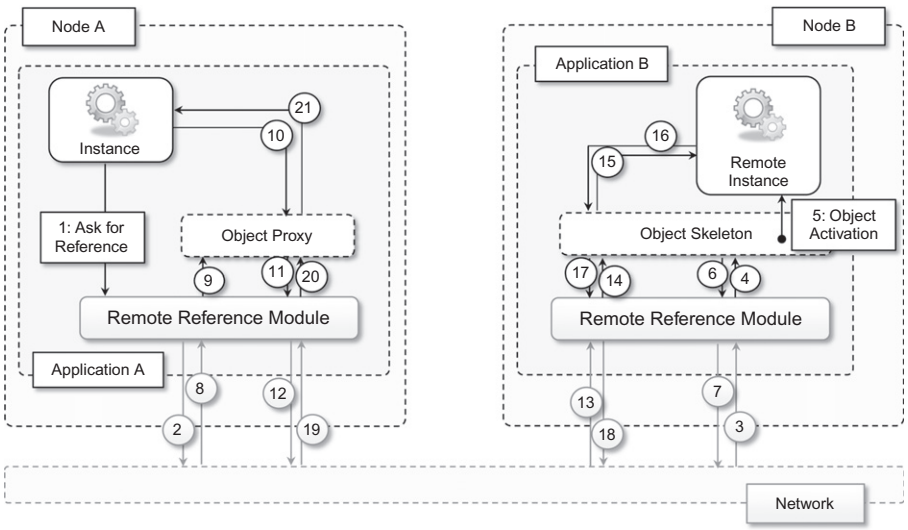

FIGURE 2.15

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

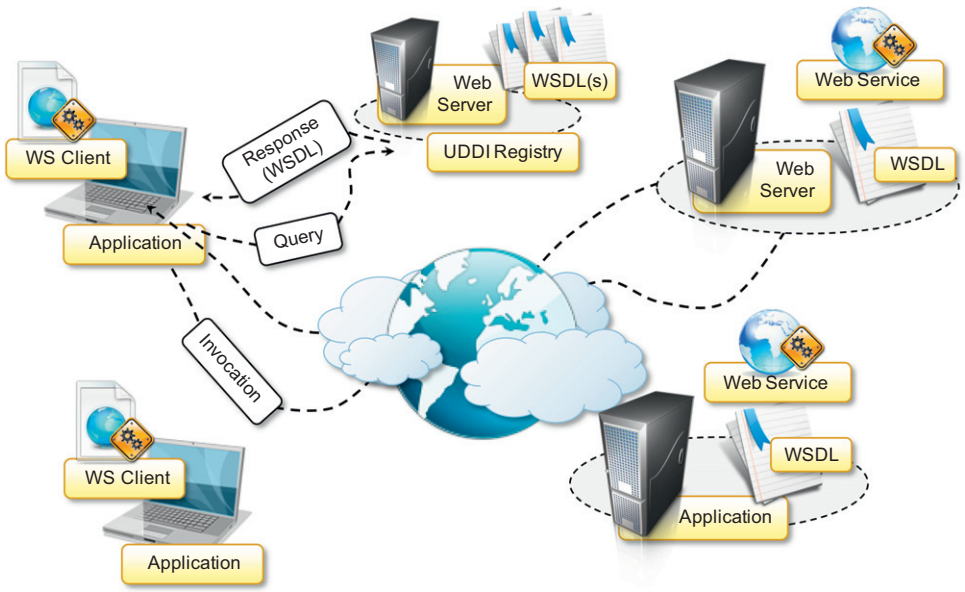

FIGURE 2.16

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

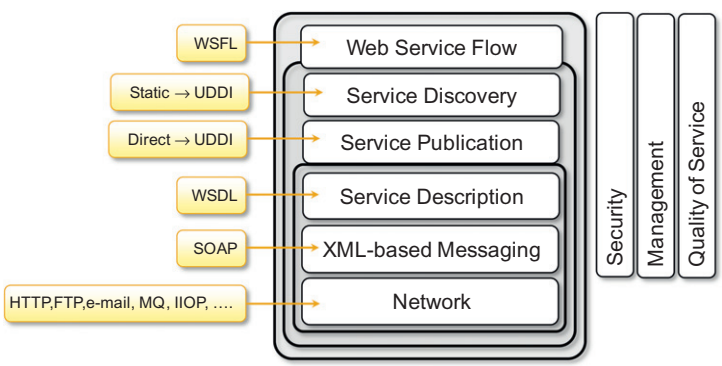

FIGURE 2.17

A Web services technologies stack.

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.

CHAPTER

Virtualization 3

Virtualization technology is one of the fundamental components of cloud computing, especially in regard to infrastructure-based services. Virtualization allows the creation of a secure, customizable, and isolated execution environment for running applications, even if they are untrusted, without affecting other users' applications. The basis of this technology is the ability of a computer program—or a combination of software and hardware—to emulate an executing environment separate from the one that hosts such programs. For example, we can run Windows OS on top of a virtual machine, which itself is running on Linux OS. Virtualization provides a great opportunity to build elastically scalable systems that can provision additional capability with minimum costs. Therefore, virtualization is widely used to deliver customizable computing environments on demand.

This chapter discusses the fundamental concepts of virtualization, its evolution, and various models and technologies used in cloud computing environments.

3.1 Introduction

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

Since its inception, virtualization has been sporadically explored and adopted, but in the last few years there has been a consistent and growing trend to leverage this technology. Virtualization technologies have gained renewed interested recently due to the confluence of several phenomena:

- *Increased performance and computing capacity.* Nowadays, the average end-user desktop PC is powerful enough to meet almost all the needs of everyday computing, with extra capacity that is rarely used. Almost all these PCs have resources enough to host a virtual machine manager and execute a virtual machine with by far acceptable performance. The same consideration applies to the high-end side of the PC market, where supercomputers can provide immense compute power that can accommodate the execution of hundreds or thousands of virtual machines.
- *Underutilized hardware and software resources.* Hardware and software underutilization is occurring due to (1) increased performance and computing capacity, and (2) the effect of

limited or sporadic use of resources. Computers today are so powerful that in most cases only a fraction of their capacity is used by an application or the system. Moreover, if we consider the IT infrastructure of an enterprise, many computers are only partially utilized whereas they could be used without interruption on a 24/7/365 basis. For example, desktop PCs mostly devoted to office automation tasks and used by administrative staff are only used during work hours, remaining completely unused overnight. Using these resources for other purposes after hours could improve the efficiency of the IT infrastructure. To transparently provide such a service, it would be necessary to deploy a completely separate environment, which can be achieved through virtualization.

- *Lack of space.* The continuous need for additional capacity, whether storage or compute power, makes data centers grow quickly. Companies such as Google and Microsoft expand their infrastructures by building data centers as large as football fields that are able to host thousands of nodes. Although this is viable for IT giants, in most cases enterprises cannot afford to build another data center to accommodate additional resource capacity. This condition, along with hardware underutilization, has led to the diffusion of a technique called *server consolidation*,[1] for which virtualization technologies are fundamental.
- *Greening initiatives.* Recently, companies are increasingly looking for ways to reduce the amount of energy they consume and to reduce their carbon footprint. Data centers are one of the major power consumers; they contribute consistently to the impact that a company has on the environment. Maintaining a data center operation not only involves keeping servers on, but a great deal of energy is also consumed in keeping them cool. Infrastructures for cooling have a significant impact on the carbon footprint of a data center. Hence, reducing the number of servers through server consolidation will definitely reduce the impact of cooling and power consumption of a data center. Virtualization technologies can provide an efficient way of consolidating servers.
- *Rise of administrative costs.* Power consumption and cooling costs have now become higher than the cost of IT equipment. Moreover, the increased demand for additional capacity, which translates into more servers in a data center, is also responsible for a significant increment in administrative costs. Computers—in particular, servers—do not operate all on their own, but they require care and feeding from system administrators. Common system administration tasks include hardware monitoring, defective hardware replacement, server setup and updates, server resources monitoring, and backups. These are labor-intensive operations, and the higher the number of servers that have to be managed, the higher the administrative costs. Virtualization can help reduce the number of required servers for a given workload, thus reducing the cost of the administrative personnel.

These can be considered the major causes for the diffusion of hardware virtualization solutions as well as the other kinds of virtualization. The first step toward consistent adoption of virtualization technologies was made with the wide spread of virtual machine-based programming languages: In 1995 Sun released Java, which soon became popular among developers. The ability to integrate small Java applications, called *applets*, made Java a very successful platform, and with the

[1]Server consolidation is a technique for aggregating multiple services and applications originally deployed on different servers on one physical server. Server consolidation allows us to reduce the power consumption of a data center and resolve hardware underutilization.

beginning of the new millennium Java played a significant role in the application server market segment, thus demonstrating that the existing technology was ready to support the execution of managed code for enterprise-class applications. In 2002 Microsoft released the first version of .NET Framework, which was Microsoft's alternative to the Java technology. Based on the same principles as Java, able to support multiple programming languages, and featuring complete integration with other Microsoft technologies, .NET Framework soon became the principal development platform for the Microsoft world and quickly became popular among developers. In 2006, two of the three "official languages" used for development at Google, Java and Python, were based on the virtual machine model. This trend of shifting toward virtualization from a programming language perspective demonstrated an important fact: The technology was ready to support virtualized solutions without a significant performance overhead. This paved the way to another and more radical form of virtualization that now has become a fundamental requisite for any data center management infrastructure.

3.2 Characteristics of virtualized environments

Virtualization is a broad concept that refers to the creation of a virtual version of something, whether hardware, a software environment, storage, or a network. In a virtualized environment there are three major components: *guest*, *host*, and *virtualization layer*. The *guest* represents the system component that interacts with the virtualization layer rather than with the host, as would normally happen. The *host* represents the original environment where the guest is supposed to be managed. The *virtualization layer* is responsible for recreating the same or a different environment where the guest will operate (see Figure 3.1).

Such a general abstraction finds different applications and then implementations of the virtualization technology. The most intuitive and popular is represented by *hardware virtualization*, which also constitutes the original realization of the virtualization concept.[2] In the case of hardware virtualization, the guest is represented by a system image comprising an operating system and installed applications. These are installed on top of virtual hardware that is controlled and managed by the virtualization layer, also called the *virtual machine manager*. The host is instead represented by the physical hardware, and in some cases the operating system, that defines the environment where the virtual machine manager is running. In the case of virtual storage, the guest might be client applications or users that interact with the virtual storage management software deployed on top of the real storage system. The case of virtual networking is also similar: The guest—applications and users—interacts with a virtual network, such as a *virtual private network (VPN)*, which is managed by specific software (VPN client) using the physical network available on the node. VPNs are useful for creating the illusion of being within a different physical network and thus accessing the resources in it, which would otherwise not be available.

[2]Virtualization is a technology that was initially developed during the mainframe era. The IBM CP/CMS mainframes were the first systems to introduce the concept of hardware virtualization and hypervisors. These systems, able to run multiple operating systems at the same time, provided a backward-compatible environment that allowed customers to run previous versions of their applications.

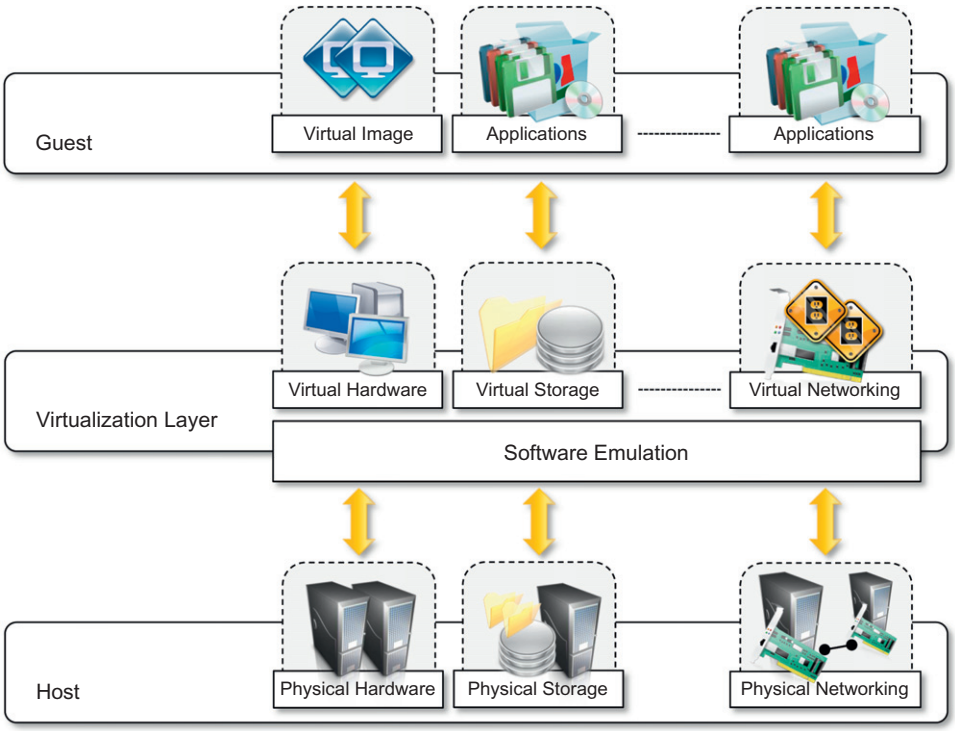

FIGURE 3.1

The virtualization reference model.

The main common characteristic of all these different implementations is the fact that the virtual environment is created by means of a *software program*. The ability to use software to emulate such a wide variety of environments creates a lot of opportunities, previously less attractive because of excessive overhead introduced by the virtualization layer. The technologies of today allow profitable use of virtualization and make it possible to fully exploit the advantages that come with it. Such advantages have always been characteristics of virtualized solutions.

3.2.1 Increased security

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

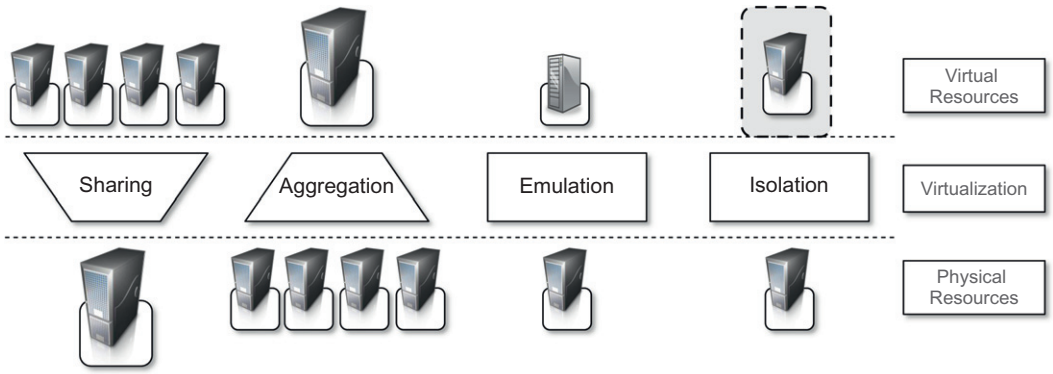

FIGURE 3.2

Functions enabled by managed execution.

information that is contained in the host can be naturally hidden without the need to install complex security policies. Increased security is a requirement when dealing with untrusted code. For example, applets downloaded from the Internet run in a sandboxed[3] version of the *Java Virtual Machine (JVM)*, which provides them with limited access to the hosting operating system resources. Both the JVM and the .NET runtime provide extensive security policies for customizing the execution environment of applications. Hardware virtualization solutions such as VMware Desktop, VirtualBox, and Parallels provide the ability to create a virtual computer with customized virtual hardware on top of which a new operating system can be installed. By default, the file system exposed by the virtual computer is completely separated from the one of the host machine. This becomes the perfect environment for running applications without affecting other users in the environment.

3.2.2 Managed execution

Virtualization of the execution environment not only allows increased security, but a wider range of features also can be implemented. In particular, *sharing*, *aggregation*, *emulation*, and *isolation* are the most relevant features (see Figure 3.2).

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

[3]The term sandbox identifies an isolated execution environment where instructions can be filtered and blocked before being translated and executed in the real execution environment. The expression sandboxed version of the Java Virtual Machine (JVM) refers to a particular configuration of the JVM where, by means of security policy, instructions that are considered potential harmful can be blocked.

- *Aggregation.* Not only is it possible to share physical resource among several guests, but virtualization also allows aggregation, which is the opposite process. A group of separate hosts can be tied together and represented to guests as a single virtual host. This function is naturally implemented in middleware for distributed computing, with a classical example represented by cluster management software, which harnesses the physical resources of a homogeneous group of machines and represents them as a single resource.
- *Emulation.* Guest programs are executed within an environment that is controlled by the virtualization layer, which ultimately is a program. This allows for controlling and tuning the environment that is exposed to guests. For instance, a completely different environment with respect to the host can be emulated, thus allowing the execution of guest programs requiring specific characteristics that are not present in the physical host. This feature becomes very useful for testing purposes, where a specific guest has to be validated against different platforms or architectures and the wide range of options is not easily accessible during development. Again, hardware virtualization solutions are able to provide virtual hardware and emulate a particular kind of device such as *Small Computer System Interface (SCSI)* devices for file I/O, without the hosting machine having such hardware installed. Old and legacy software that does not meet the requirements of current systems can be run on emulated hardware without any need to change the code. This is possible either by emulating the required hardware architecture or within a specific operating system sandbox, such as the MS-DOS mode in Windows 95/98. Another example of emulation is an arcade-game emulator that allows us to play arcade games on a normal personal computer.
- *Isolation.* Virtualization allows providing guests—whether they are operating systems, applications, or other entities—with a completely separate environment, in which they are executed. The guest program performs its activity by interacting with an abstraction layer, which provides access to the underlying resources. Isolation brings several benefits; for example, it allows multiple guests to run on the same host without interfering with each other. Second, it provides a separation between the host and the guest. The virtual machine can filter the activity of the guest and prevent harmful operations against the host.

Besides these characteristics, another important capability enabled by virtualization is *performance tuning*. This feature is a reality at present, given the considerable advances in hardware and software supporting virtualization. It becomes easier to control the performance of the guest by finely tuning the properties of the resources exposed through the virtual environment. This capability provides a means to effectively implement a quality-of-service (QoS) infrastructure that more easily fulfills the service-level agreement (SLA) established for the guest. For instance, software-implementing hardware virtualization solutions can expose to a guest operating system only a fraction of the memory of the host machine or set the maximum frequency of the processor of the virtual machine. Another advantage of managed execution is that sometimes it allows easy capturing of the state of the guest program, persisting it, and resuming its execution. This, for example, allows virtual machine managers such as Xen Hypervisor to stop the execution of a guest operating system, move its virtual image into another machine, and resume its execution in a completely transparent manner. This technique is called *virtual machine migration* and constitutes an important feature in virtualized data centers for optimizing their efficiency in serving application demands.

3.2.3 Portability

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

3.3 Taxonomy of virtualization techniques

Virtualization covers a wide range of emulation techniques that are applied to different areas of computing. A classification of these techniques helps us better understand their characteristics and use (see Figure 3.3).

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

3.3.1 Execution virtualization

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

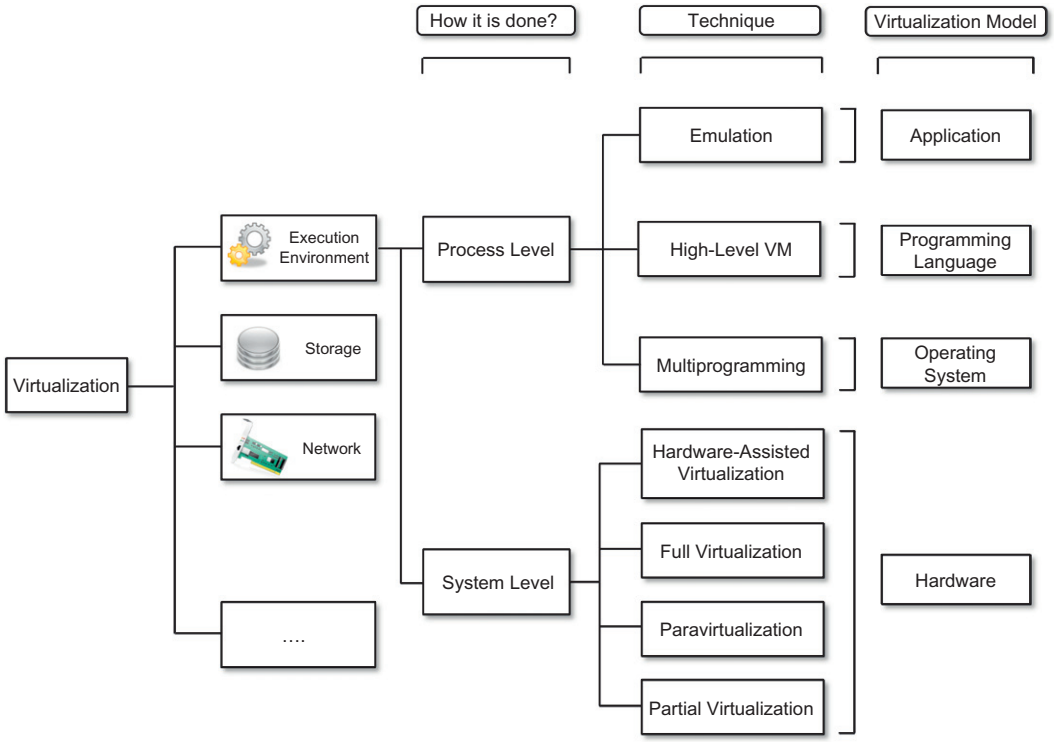

FIGURE 3.3

A taxonomy of virtualization techniques.

3.3.1.1 Machine reference model

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

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

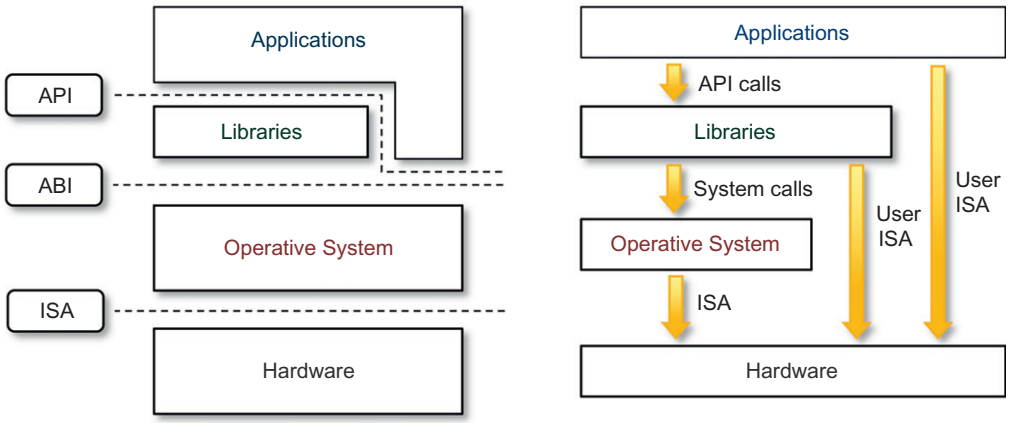

FIGURE 3.4

A machine reference model.

implement the same ABI. The highest level of abstraction is represented by the *application programming interface (API)*, which interfaces applications to libraries and/or the underlying operating system.

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

For this purpose, the instruction set exposed by the hardware has been divided into different security classes that define who can operate with them. The first distinction can be made between *privileged* and *nonprivileged* instructions. Nonprivileged instructions are those instructions that can be used without interfering with other tasks because they do not access shared resources. This category contains, for example, all the floating, fixed-point, and arithmetic instructions. Privileged instructions are those that are executed under specific restrictions and are mostly used for sensitive operations, which expose (*behavior-sensitive*) or modify (*control-sensitive*) the privileged state. For instance, behavior-sensitive instructions are those that operate on the I/O, whereas control-sensitive instructions alter the state of the CPU registers. Some types of architecture feature more than one class of privileged instructions and implement a finer control of how these instructions can be accessed. For instance, a possible implementation features a hierarchy of privileges (see Figure 3.5) in the form of ring-based security: *Ring 0*, *Ring 1*, *Ring 2*, and *Ring 3*; Ring 0 is in the most privileged level and Ring 3 in the least privileged level. Ring 0 is used by the kernel of the OS, rings 1 and 2 are used by the OS-level services, and Ring 3 is used by the user. Recent systems support only two levels, with Ring 0 for supervisor mode and Ring 3 for user mode.

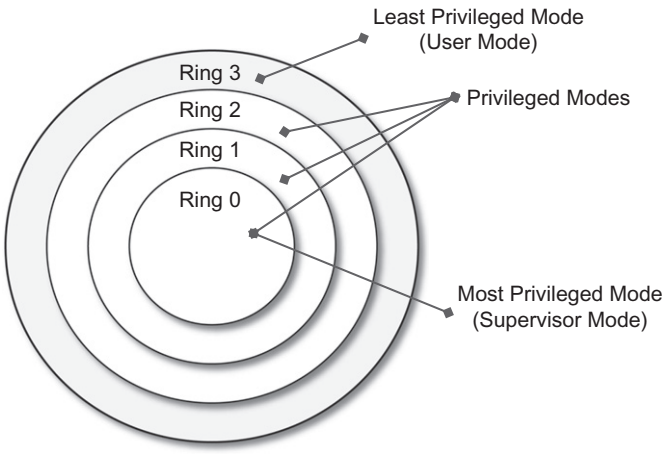

FIGURE 3.5

Security rings and privilege modes.

All the current systems support at least two different execution modes: *supervisor mode* and *user mode*. The first mode denotes an execution mode in which all the instructions (privileged and nonprivileged) can be executed without any restriction. This mode, also called *master mode* or *kernel mode*, is generally used by the operating system (or the hypervisor) to perform sensitive operations on hardware-level resources. In user mode, there are restrictions to control the machine-level resources. If code running in user mode invokes the privileged instructions, hardware interrupts occur and trap the potentially harmful execution of the instruction. Despite this, there might be some instructions that can be invoked as privileged instructions under some conditions and as nonprivileged instructions under other conditions.

The distinction between *user* and *supervisor* mode allows us to understand the role of the *hypervisor* and why it is called that. Conceptually, the hypervisor runs above the supervisor mode, and from here the prefix *hyper-* is used. In reality, hypervisors are run in supervisor mode, and the division between privileged and nonprivileged instructions has posed challenges in designing virtual machine managers. It is expected that all the sensitive instructions will be executed in privileged mode, which requires supervisor mode in order to avoid traps. Without this assumption it is impossible to fully emulate and manage the status of the CPU for guest operating systems. Unfortunately, this is not true for the original ISA, which allows 17 sensitive instructions to be called in user mode. This prevents multiple operating systems managed by a single hypervisor to be isolated from each other, since they are able to access the privileged state of the processor and change it.[4] More recent implementations of ISA (Intel VT and AMD Pacifica) have solved this problem by redesigning such instructions as privileged ones.

By keeping in mind this reference model, it is possible to explore and better understand the various techniques utilized to virtualize execution environments and their relationships to the other components of the system.

[4]It is expected that in a hypervisor-managed environment, all the guest operating system code will be run in user mode in order to prevent it from directly accessing the status of the CPU. If there are sensitive instructions that can be called in user mode (that is, implemented as nonprivileged instructions), it is no longer possible to completely isolate the guest OS.

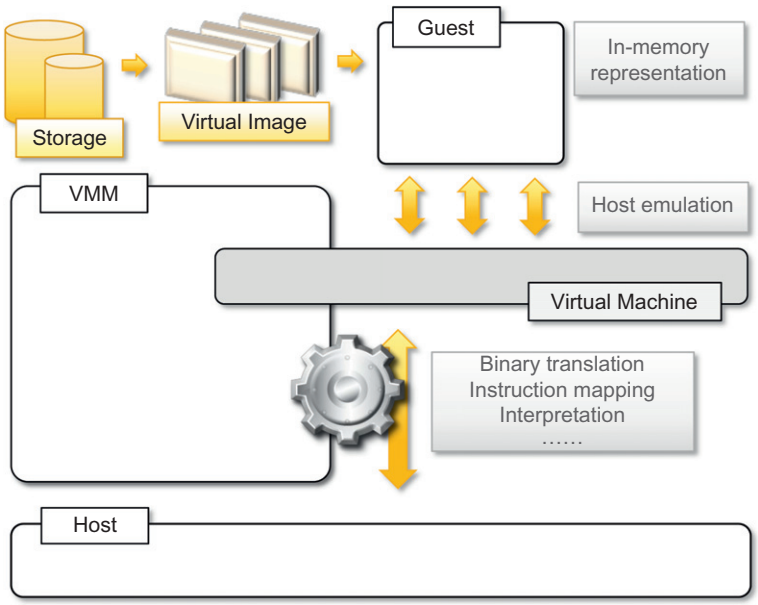

FIGURE 3.6

A hardware virtualization reference model.

3.3.1.2 Hardware-level virtualization

Hardware-level virtualization is a virtualization technique that provides an abstract execution environment in terms of computer hardware on top of which a guest operating system can be run. In this model, the guest is represented by the operating system, the host by the physical computer hardware, the virtual machine by its emulation, and the virtual machine manager by the hypervisor (see Figure 3.6). The hypervisor is generally a program or a combination of software and hardware that allows the abstraction of the underlying physical hardware.

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

Hypervisors

A fundamental element of hardware virtualization is the hypervisor, or virtual machine manager (VMM). It recreates a hardware environment in which guest operating systems are installed. There are two major types of hypervisor: *Type I* and *Type II* (see Figure 3.7).

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

- *Type II* hypervisors require the support of an operating system to provide virtualization services. This means that they are programs managed by the operating system, which interact with it through the ABI and emulate the ISA of virtual hardware for guest operating systems. This type of hypervisor is also called a *hosted virtual machine* since it is hosted within an operating system.

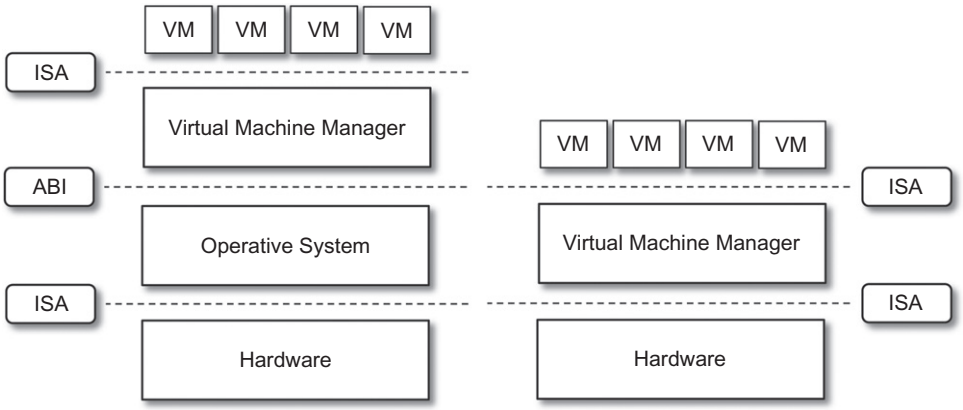

FIGURE 3.7

Hosted (left) and native (right) virtual machines. This figure provides a graphical representation of the two types of hypervisors.

Conceptually, a virtual machine manager is internally organized as described in Figure 3.8. Three main modules, *dispatcher*, *allocator*, and *interpreter*, coordinate their activity in order to emulate the underlying hardware. The dispatcher constitutes the entry point of the monitor and reroutes the instructions issued by the virtual machine instance to one of the two other modules. The allocator is responsible for deciding the system resources to be provided to the VM: whenever a virtual machine tries to execute an instruction that results in changing the machine resources associated with that VM, the allocator is invoked by the dispatcher. The interpreter module consists of interpreter routines. These are executed whenever a virtual machine executes a privileged instruction: a trap is triggered and the corresponding routine is executed.

The design and architecture of a virtual machine manager, together with the underlying hardware design of the host machine, determine the full realization of hardware virtualization, where a guest operating system can be transparently executed on top of a VMM as though it were run on the underlying hardware. The criteria that need to be met by a virtual machine manager to efficiently support virtualization were established by Goldberg and Popek in 1974 [23]. Three properties have to be satisfied:

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

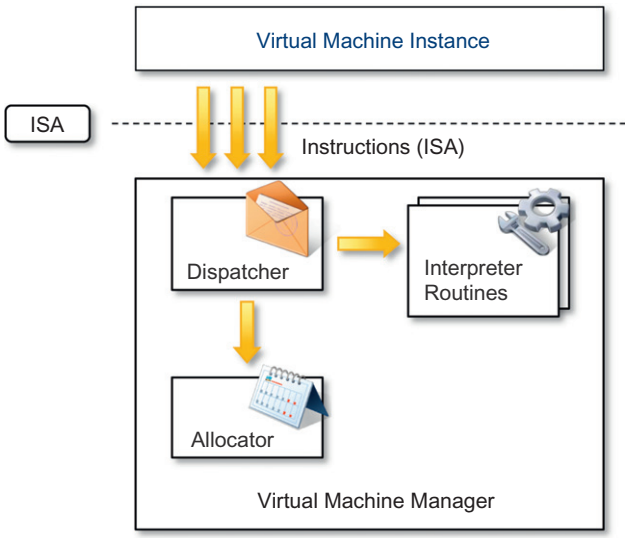

FIGURE 3.8

A hypervisor reference architecture.

- *Efficiency.* A statistically dominant fraction of the machine instructions should be executed without intervention from the virtual machine manager.

The major factor that determines whether these properties are satisfied is represented by the layout of the ISA of the host running a virtual machine manager. Popek and Goldberg provided a classification of the instruction set and proposed three theorems that define the properties that hardware instructions need to satisfy in order to efficiently support virtualization.

THEOREM 3.1

For any conventional third-generation computer, a VMM may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

This theorem establishes that all the instructions that change the configuration of the system resources should generate a trap in user mode and be executed under the control of the virtual machine manager. This allows hypervisors to efficiently control only those instructions that would reveal the presence of an abstraction layer while executing all the rest of the instructions without considerable performance loss. The theorem always guarantees the resource control property when the hypervisor is in the most privileged mode (*Ring 0*). The nonprivileged instructions must be executed without the intervention of the hypervisor. The equivalence property also holds good since the output of the code is the same in both cases because the code is not changed.

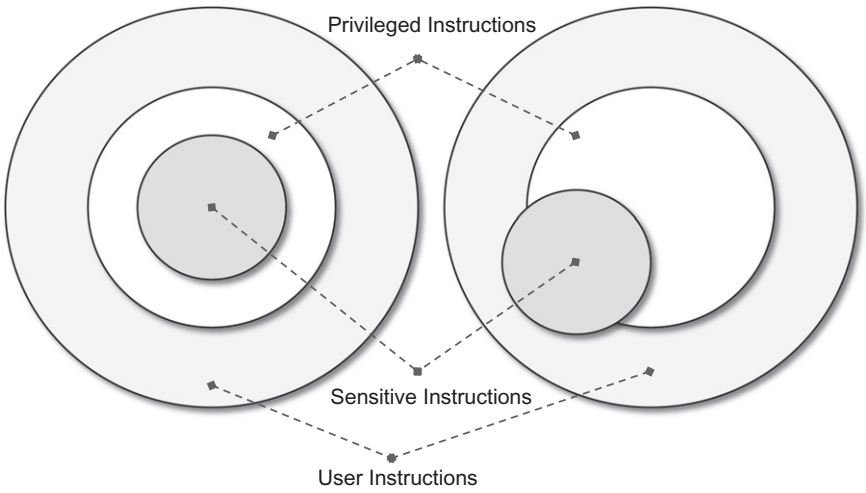

FIGURE 3.9

A virtualizable computer (left) and a nonvirtualizable computer (right).

THEOREM 3.2

A conventional third-generation computer is recursively virtualizable if:

- It is virtualizable and
- A VMM without any timing dependencies can be constructed for it.

Recursive virtualization is the ability to run a virtual machine manager on top of another virtual machine manager. This allows nesting hypervisors as long as the capacity of the underlying resources can accommodate that. Virtualizable hardware is a prerequisite to recursive virtualization.

THEOREM 3.3

A hybrid VMM may be constructed for any conventional third-generation machine in which the set of user-sensitive instructions is a subset of the set of privileged instructions.

There is another term, *hybrid virtual machine (HVM)*, which is less efficient than the virtual machine system. In the case of an HVM, more instructions are interpreted rather than being executed directly. All instructions in virtual supervisor mode are interpreted. Whenever there is an attempt to execute a behavior-sensitive or control-sensitive instruction, HVM controls the execution directly or gains the control via a trap. Here all sensitive instructions are caught by HVM that are simulated.

This reference model represents what we generally consider classic virtualization—that is, the ability to execute a guest operating system in complete isolation. To a greater extent, hardware-level virtualization includes several strategies that differentiate from each other in terms of which kind of support is expected from the underlying hardware, what is actually abstracted from the host, and whether the guest should be modified or not.

Hardware virtualization techniques

Hardware-assisted virtualization. This term refers to a scenario in which the hardware provides architectural support for building a virtual machine manager able to run a guest operating system in complete isolation. This technique was originally introduced in the IBM System/370. At present, examples of hardware-assisted virtualization are the extensions to the x86-64 bit architecture introduced with *Intel VT* (formerly known as *Vanderpool*) and *AMD V* (formerly known as *Pacifica*). These extensions, which differ between the two vendors, are meant to reduce the performance penalties experienced by emulating x86 hardware with hypervisors. Before the introduction of hardware-assisted virtualization, software emulation of x86 hardware was significantly costly from the performance point of view. The reason for this is that by design the x86 architecture did not meet the formal requirements introduced by Popek and Goldberg, and early products were using binary translation to trap some sensitive instructions and provide an emulated version. Products such as VMware Virtual Platform, introduced in 1999 by VMware, which pioneered the field of x86 virtualization, were based on this technique. After 2006, Intel and AMD introduced processor extensions, and a wide range of virtualization solutions took advantage of them*:* Kernel-based Virtual Machine (KVM), VirtualBox, Xen, VMware, Hyper-V, Sun xVM, Parallels, and others.

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

Paravirtualization. This is a not-transparent virtualization solution that allows implementing thin virtual machine managers. Paravirtualization techniques expose a software interface to the virtual machine that is slightly modified from the host and, as a consequence, guests need to be modified. The aim of paravirtualization is to provide the capability to demand the execution of performance-critical operations directly on the host, thus preventing performance losses that would otherwise be experienced in managed execution. This allows a simpler implementation of virtual machine managers that have to simply transfer the execution of these operations, which were hard to virtualize, directly to the host. To take advantage of such an opportunity, guest operating systems

need to be modified and explicitly ported by remapping the performance-critical operations through the virtual machine software interface. This is possible when the source code of the operating system is available, and this is the reason that paravirtualization was mostly explored in the open-source and academic environment. Whereas this technique was initially applied in the IBM VM operating system families, the term *paravirtualization* was introduced in literature in the Denali project [24] at the University of Washington. This technique has been successfully used by Xen for providing virtualization solutions for Linux-based operating systems specifically ported to run on Xen hypervisors. Operating systems that cannot be ported can still take advantage of paravirtualization by using ad hoc device drivers that remap the execution of critical instructions to the paravirtualization APIs exposed by the hypervisor. Xen provides this solution for running Windows-based operating systems on x86 architectures. Other solutions using paravirtualization include VMWare, Parallels, and some solutions for embedded and real-time environments such as TRANGO, Wind River, and XtratuM.

Partial virtualization. Partial virtualization provides a partial emulation of the underlying hardware, thus not allowing the complete execution of the guest operating system in complete isolation. Partial virtualization allows many applications to run transparently, but not all the features of the operating system can be supported, as happens with full virtualization. An example of partial virtualization is address space virtualization used in time-sharing systems; this allows multiple applications and users to run concurrently in a separate memory space, but they still share the same hardware resources (disk, processor, and network). Historically, partial virtualization has been an important milestone for achieving full virtualization, and it was implemented on the experimental IBM M44/44X. Address space virtualization is a common feature of contemporary operating systems.

Operating system-level virtualization

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

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

virtualization. On the other hand, operating system-level virtualization does not expose the same flexibility of hardware virtualization, since all the user space instances must share the same operating system.

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

Examples of operating system-level virtualizations are FreeBSD Jails, IBM Logical Partition (LPAR), SolarisZones and Containers, Parallels Virtuozzo Containers, OpenVZ, iCore Virtual Accounts, Free Virtual Private Server (FreeVPS), and others. The services offered by these technologies differ, and most of them are available on Unix-based systems. Some of them, such as Solaris and OpenVZ, allow for different versions of the same operating system to operate concurrently.

3.3.1.3 Programming language-level virtualization

Programming language-level virtualization is mostly used to achieve ease of deployment of applications, managed execution, and portability across different platforms and operating systems. It consists of a virtual machine executing the byte code of a program, which is the result of the compilation process. Compilers implemented and used this technology to produce a binary format representing the machine code for an abstract architecture. The characteristics of this architecture vary from implementation to implementation. Generally these virtual machines constitute a simplification of the underlying hardware instruction set and provide some high-level instructions that map some of the features of the languages compiled for them. At runtime, the byte code can be either interpreted or compiled on the fly—or *jitted*[5]—against the underlying hardware instruction set.

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

[5]The term *jitted* is an improper use of the *just-in-time (JIT)* acronym as a verb, which has now become common. It refers to a specific execution strategy in which the byte code of a method is compiled against the underlying machine code upon method call—that is, *just in time*. Initial implementations of programming-level virtualization were based on interpretation, which led to considerable slowdowns during execution. The advantage of just-in-time compilation is that the machine code that has been compiled can be reused for executing future calls to the same methods. Virtual machines that implement JIT compilation generally have a method cache that stores the code generated for each method and simply look up this cache before triggering the compilation upon each method call.

.NET Framework. Currently, the Java platform and .NET Framework represent the most popular technologies for enterprise application development.

Both Java and the CLI are *stack-based* virtual machines: The reference model of the abstract architecture is based on an execution stack that is used to perform operations. The byte code generated by compilers for these architectures contains a set of instructions that load operands on the stack, perform some operations with them, and put the result on the stack. Additionally, specific instructions for invoking methods and managing objects and classes are included. Stack-based virtual machines possess the property of being easily interpreted and executed simply by lexical analysis and hence are easily portable over different architectures. An alternative solution is offered by *register-based* virtual machines, in which the reference model is based on registers. This kind of virtual machine is closer to the underlying architecture we use today. An example of a register-based virtual machine is Parrot, a programming-level virtual machine that was originally designed to support the execution of PERL and then generalized to host the execution of dynamic languages.

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

Implementations of this model are also called *high-level virtual machines*, since high-level programming languages are compiled to a conceptual ISA, which is further interpreted or dynamically translated against the specific instruction of the hosting platform.

3.3.1.4 Application-level virtualization

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

also be used to execute program binaries compiled for different hardware architectures. In this case, one of the following strategies can be implemented:

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

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

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

One of the most popular solutions implementing application virtualization is *Wine*, which is a software application allowing Unix-like operating systems to execute programs written for the Microsoft Windows platform. Wine features a software application acting as a container for the guest application and a set of libraries, called *Winelib*, that developers can use to compile applications to be ported on Unix systems. Wine takes its inspiration from a similar product from Sun, *Windows Application Binary Interface (WABI)*, which implements the Win 16 API specifications on Solaris. A similar solution for the Mac OS X environment is CrossOver, which allows running Windows applications directly on the Mac OS X operating system. VMware ThinApp, another product in this area, allows capturing the setup of an installed application and packaging it into an executable image isolated from the hosting operating system.

3.3.2 Other types of virtualization

Other than execution virtualization, other types of virtualization provide an abstract environment to interact with. These mainly cover storage, networking, and client/server interaction.

3.3.2.1 Storage virtualization

Storage virtualization is a system administration practice that allows decoupling the physical organization of the hardware from its logical representation. Using this technique, users do not have to be worried about the specific location of their data, which can be identified using a logical path.

Storage virtualization allows us to harness a wide range of storage facilities and represent them under a single logical file system. There are different techniques for storage virtualization, one of the most popular being network-based virtualization by means of *storage area networks (SANs)*. SANs use a network-accessible device through a large bandwidth connection to provide storage facilities.

3.3.2.2 Network virtualization

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

3.3.2.3 Desktop virtualization

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

In this scenario, an infrastructure supporting hardware virtualization is fundamental to provide access to multiple desktop environments hosted on the same server; a specific desktop environment is stored in a virtual machine image that is loaded and started on demand when a client connects to the desktop environment. This is a typical cloud computing scenario in which the user leverages the virtual infrastructure for performing the daily tasks on his computer. The advantages of desktop virtualization are high availability, persistence, accessibility, and ease of management. As we will discuss in Section 4.5.4 of the next chapter, security issues can prevent the use of this technology. The basic services for remotely accessing a desktop environment are implemented in software components such as Windows Remote Services, VNC, and X Server. Infrastructures for desktop virtualization based on cloud computing solutions include Sun Virtual Desktop Infrastructure (VDI), Parallels Virtual Desktop Infrastructure (VDI), Citrix XenDesktop, and others.

3.3.2.4 Application server virtualization

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

3.4 Virtualization and cloud computing

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

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

Besides being an enabler for computation on demand, virtualization also gives the opportunity to design more efficient computing systems by means of consolidation, which is performed transparently to cloud computing service users. Since virtualization allows us to create isolated and controllable environments, it is possible to serve these environments with the same resource without them interfering with each other. If the underlying resources are capable enough, there will be no evidence of such sharing. This opportunity is particularly attractive when resources are underutilized, because it allows reducing the number of active resources by aggregating virtual machines over a smaller number of resources that become fully utilized. This practice is also known as *server consolidation*, while the movement of virtual machine instances is called *virtual machine migration* (see Figure 3.10). Because virtual machine instances are controllable environments, consolidation can be applied with a minimum impact, either by temporarily stopping its execution and moving its data to the new resources or by performing a finer control and moving the instance while it is running. This second techniques is known as *live migration* and in general is more complex to implement but more efficient since there is no disruption of the activity of the virtual machine instance.[6]

[6]It is important to notice that cloud computing is strongly leveraged for the development of applications that need to scale on demand. In most cases, this is because applications have to process increased workloads or serve more requests, which makes them server applications. In this scenario, it is evident that live migration offers a better solution because it does not create any service interruption during consolidation.

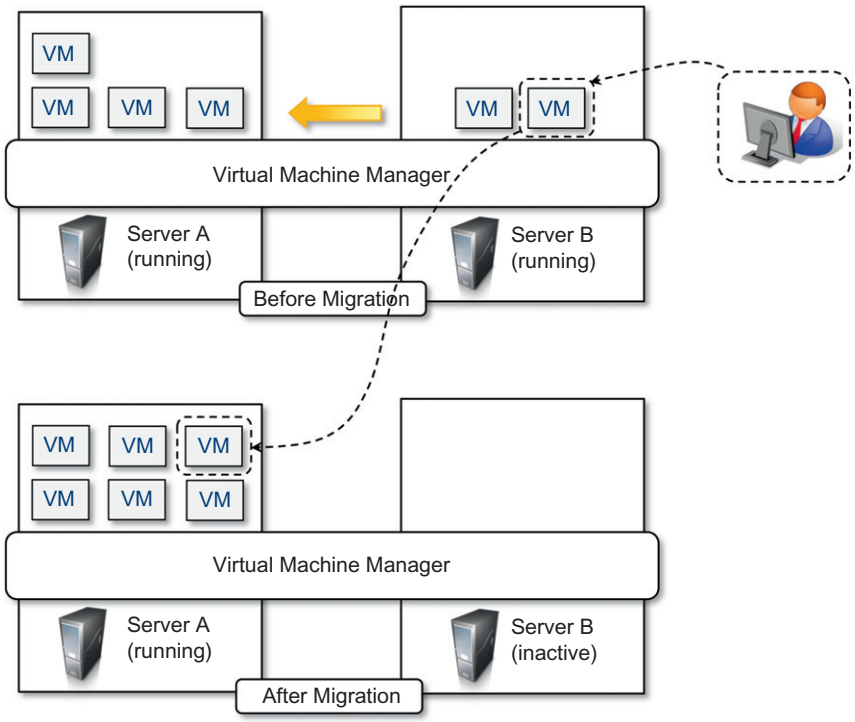

FIGURE 3.10

Live migration and server consolidation.

Server consolidation and virtual machine migration are principally used in the case of hardware virtualization, even though they are also technically possible in the case of programming language virtualization (see Figure 3.9).

Storage virtualization constitutes an interesting opportunity given by virtualization technologies, often complementary to the execution of virtualization. Even in this case, vendors backed by large computing infrastructures featuring huge storage facilities can harness these facilities into a virtual storage service, easily partitionable into slices. These slices can be dynamic and offered as a service. Again, opportunities to secure and protect the hosting infrastructure are available, as are methods for easy accountability of such services.

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

3.5 Pros and cons of virtualization

Virtualization has now become extremely popular and widely used, especially in cloud computing. The primary reason for its wide success is the elimination of technology barriers that prevented virtualization from being an effective and viable solution in the past. The most relevant barrier has been performance. Today, the capillary diffusion of the Internet connection and the advancements in computing technology have made virtualization an interesting opportunity to deliver on-demand IT infrastructure and services. Despite its renewed popularity, this technology has benefits and also drawbacks.

3.5.1 Advantages of virtualization

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

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

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

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

3.5.2 The other side of the coin: disadvantages

Virtualization also has downsides. The most evident is represented by a performance decrease of guest systems as a result of the intermediation performed by the virtualization layer. In addition, suboptimal use of the host because of the abstraction layer introduced by virtualization management software can lead to a very inefficient utilization of the host or a degraded user experience. Less evident, but perhaps more dangerous, are the implications for security, which are mostly due to the ability to emulate a different execution environment.

3.5.2.1 Performance degradation

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

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

- Maintaining the status of virtual processors
- Support of privileged instructions (trap and simulate privileged instructions)
- Support of paging within VM
- Console functions

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

Similar consideration can be made in the case of virtualization technologies at higher levels, such as in the case of programming language virtual machines (Java, .NET, and others). Binary translation and interpretation can slow down the execution of managed applications. Moreover, because their execution is filtered by the runtime environment, access to memory and other physical resources can represent sources of performance degradation.

These concerns are becoming less and less important thanks to technology advancements and the ever-increasing computational power available today. For example, specific techniques for hardware virtualization such as *paravirtualization* can increase the performance of the guest program by offloading most of its execution to the host without any change. In programming-level virtual machines such as the JVM or .NET, compilation to native code is offered as an option when performance is a serious concern.

3.5.2.2 Inefficiency and degraded user experience

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

first version of Java the support for graphic programming was very limited and the look and feel of applications was very poor compared to native applications. These issues have been resolved by providing a new framework called *Swing* for designing the user interface, and further improvements have been done by integrating support for the OpenGL libraries in the software development kit.

3.5.2.3 Security holes and new threats

Virtualization opens the door to a new and unexpected form of *phishing*.[7] The capability of emulating a host in a completely transparent manner led the way to malicious programs that are designed to extract sensitive information from the guest.

In the case of hardware virtualization, malicious programs can preload themselves before the operating system and act as a thin virtual machine manager toward it. The operating system is then controlled and can be manipulated to extract sensitive information of interest to third parties. Examples of these kinds of malware are BluePill and SubVirt. BluePill, malware targeting the AMD processor family, moves the execution of the installed OS within a virtual machine. The original version of SubVirt was developed as a prototype by Microsoft through collaboration with Michigan University. SubVirt infects the guest OS, and when the virtual machine is rebooted, it gains control of the host. The diffusion of such kinds of malware is facilitated by the fact that originally, hardware and CPUs were not manufactured with virtualization in mind. In particular, the existing instruction sets cannot be simply changed or updated to suit the needs of virtualization. Recently, both Intel and AMD have introduced hardware support for virtualization with Intel VT and AMD Pacifica, respectively.

The same considerations can be made for programming-level virtual machines: Modified versions of the runtime environment can access sensitive information or monitor the memory locations utilized by guest applications while these are executed. To make this possible, the original version of the runtime environment needs to be replaced by the modified one, which can generally happen if the malware is run within an administrative context or a security hole of the host operating system is exploited.

3.6 Technology examples

A wide range of virtualization technology is available especially for virtualizing computing environments. In this section, we discuss the most relevant technologies and approaches utilized in the field. Cloud-specific solutions are discussed in the next chapter.

[7] *Phishing* is a term that identifies a malicious practice aimed at capturing sensitive user information, such as usernames and passwords, by recreating an environment identical in functionalities and appearance to the one that manages this information. Phishing most commonly occurs on the Web, where the user is redirected to a malicious website that is a replica of the original and the purpose of which is to collect the information to impersonate the user on the original Website (e.g., a bank site) and access the user's confidential data.

3.6.1 Xen: paravirtualization

Xen is an open-source initiative implementing a virtualization platform based on paravirtualization. Initially developed by a group of researchers at the University of Cambridge in the United Kingdom, Xen now has a large open-source community backing it. Citrix also offers it as a commercial solution, XenSource. Xen-based technology is used for either desktop virtualization or server virtualization, and recently it has also been used to provide cloud computing solutions by means of Xen Cloud Platform (XCP). At the basis of all these solutions is the Xen Hypervisor, which constitutes the core technology of Xen. Recently Xen has been advanced to support full virtualization using hardware-assisted virtualization.

Xen is the most popular implementation of *paravirtualization*, which, in contrast with full virtualization, allows high-performance execution of guest operating systems. This is made possible by eliminating the performance loss while executing instructions that require special management. This is done by modifying portions of the guest operating systems run by Xen with reference to the execution of such instructions. Therefore it is not a transparent solution for implementing virtualization. This is particularly true for x86, which is the most popular architecture on commodity machines and servers.

Figure 3.11 describes the architecture of Xen and its mapping onto a classic x86 privilege model. A Xen-based system is managed by the *Xen hypervisor*, which runs in the highest privileged mode and controls the access of guest operating system to the underlying hardware. Guest

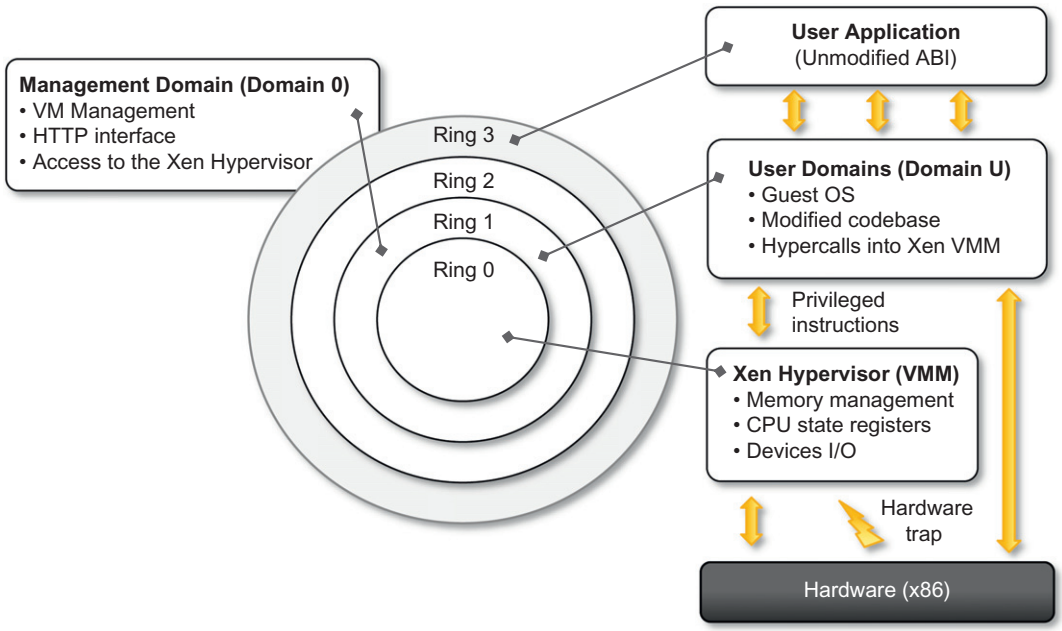

FIGURE 3.11

Xen architecture and guest OS management.

operating systems are executed within *domains*, which represent virtual machine instances. Moreover, specific control software, which has privileged access to the host and controls all the other guest operating systems, is executed in a special domain called *Domain 0*. This is the first one that is loaded once the virtual machine manager has completely booted, and it hosts a HyperText Transfer Protocol (HTTP) server that serves requests for virtual machine creation, configuration, and termination. This component constitutes the embryonic version of a distributed virtual machine manager, which is an essential component of cloud computing systems providing Infrastructure-as-a-Service (IaaS) solutions.

Many of the x86 implementations support four different security levels, called *rings*, where Ring 0 represent the level with the highest privileges and Ring 3 the level with the lowest ones. Almost all the most popular operating systems, except OS/2, utilize only two levels: Ring 0 for the kernel code, and Ring 3 for user application and nonprivileged OS code. This provides the opportunity for Xen to implement virtualization by executing the hypervisor in Ring 0, Domain 0, and all the other domains running guest operating systems—generally referred to as *Domain U*—in Ring 1, while the user applications are run in Ring 3. This allows Xen to maintain the ABI unchanged, thus allowing an easy switch to Xen-virtualized solutions from an application point of view. Because of the structure of the x86 instruction set, some instructions allow code executing in Ring 3 to jump into Ring 0 (kernel mode). Such operation is performed at the hardware level and therefore within a virtualized environment will result in a *trap* or *silent fault*, thus preventing the normal operations of the guest operating system, since this is now running in Ring 1. This condition is generally triggered by a subset of the system calls. To avoid this situation, operating systems need to be changed in their implementation, and the sensitive system calls need to be reimplemented with *hypercalls*, which are specific calls exposed by the virtual machine interface of Xen. With the use of hypercalls, the Xen hypervisor is able to catch the execution of all the sensitive instructions, manage them, and return the control to the guest operating system by means of a supplied handler.

Paravirtualization needs the operating system codebase to be modified, and hence not all operating systems can be used as guests in a Xen-based environment. More precisely, this condition holds in a scenario where it is not possible to leverage hardware-assisted virtualization, which allows running the hypervisor in Ring -1 and the guest operating system in Ring 0. Therefore, Xen exhibits some limitations in the case of legacy hardware and legacy operating systems. In fact, these cannot be modified to be run in Ring 1 safely since their codebase is not accessible and, at the same time, the underlying hardware does not provide any support to run the hypervisor in a more privileged mode than Ring 0. Open-source operating systems such as Linux can be easily modified, since their code is publicly available and Xen provides full support for their virtualization, whereas components of the Windows family are generally not supported by Xen unless hardware-assisted virtualization is available. It can be observed that the problem is now becoming less and less crucial since both new releases of operating systems are designed to be virtualization aware and the new hardware supports x86 virtualization.

3.6.2 VMware: full virtualization

VMware's technology is based on the concept of *full virtualization*, where the underlying hardware is replicated and made available to the guest operating system, which runs unaware of such abstraction layers and does not need to be modified. VMware implements full virtualization either in the

desktop environment, by means of *Type II* hypervisors, or in the server environment, by means of *Type I* hypervisors. In both cases, full virtualization is made possible by means of *direct execution* (for nonsensitive instructions) and *binary translation* (for sensitive instructions), thus allowing the virtualization of architecture such as x86.

Besides these two core solutions, VMware provides additional tools and software that simplify the use of virtualization technology either in a desktop environment, with tools enhancing the integration of virtual guests with the host, or in a server environment, with solutions for building and managing virtual computing infrastructures.

3.6.2.1 Full virtualization and binary translation

VMware is well known for the capability to virtualize x86 architectures, which runs unmodified on top of their hypervisors. With the new generation of hardware architectures and the introduction of *hardware-assisted virtualization* (Intel VT-x and AMD V) in 2006, full virtualization is made possible with hardware support, but before that date, the use of *dynamic binary translation* was the only solution that allowed running x86 guest operating systems unmodified in a virtualized environment.

As discussed before, x86 architecture design does not satisfy the first theorem of virtualization, since the set of sensitive instructions is not a subset of the privileged instructions. This causes a different behavior when such instructions are not executed in Ring 0, which is the normal case in a virtualization scenario where the guest OS is run in Ring 1. Generally, a trap is generated and the way it is managed differentiates the solutions in which virtualization is implemented for x86 hardware. In the case of dynamic binary translation, the trap triggers the translation of the offending instructions into an equivalent set of instructions that achieves the same goal without generating exceptions. Moreover, to improve performance, the equivalent set of instruction is cached so that translation is no longer necessary for further occurrences of the same instructions. Figure 3.12 gives an idea of the process.

This approach has both advantages and disadvantages. The major advantage is that guests can run unmodified in a virtualized environment, which is a crucial feature for operating systems for which source code is not available. This is the case, for example, of operating systems in the Windows family. Binary translation is a more portable solution for full virtualization. On the other hand, translating instructions at runtime introduces an additional overhead that is not present in other approaches (paravirtualization or hardware-assisted virtualization). Even though such disadvantage exists, binary translation is applied to only a subset of the instruction set, whereas the others are managed through direct execution on the underlying hardware. This somehow reduces the impact on performance of binary translation.

CPU virtualization is only a component of a fully virtualized hardware environment. VMware achieves full virtualization by providing virtual representation of memory and I/O devices. Memory virtualization constitutes another challenge of virtualized environments and can deeply impact performance without the appropriate hardware support. The main reason is the presence of a *memory management unit (MMU)*, which needs to be emulated as part of the virtual hardware. Especially in the case of *hosted hypervisors* (Type II), where the virtual MMU and the host-OS MMU are traversed sequentially before getting to the physical memory page, the impact on performance can be significant. To avoid nested translation, the *translation look-aside buffer (TLB)* in the virtual MMU directly maps physical pages, and the performance slowdown only occurs in case of a TLB miss.

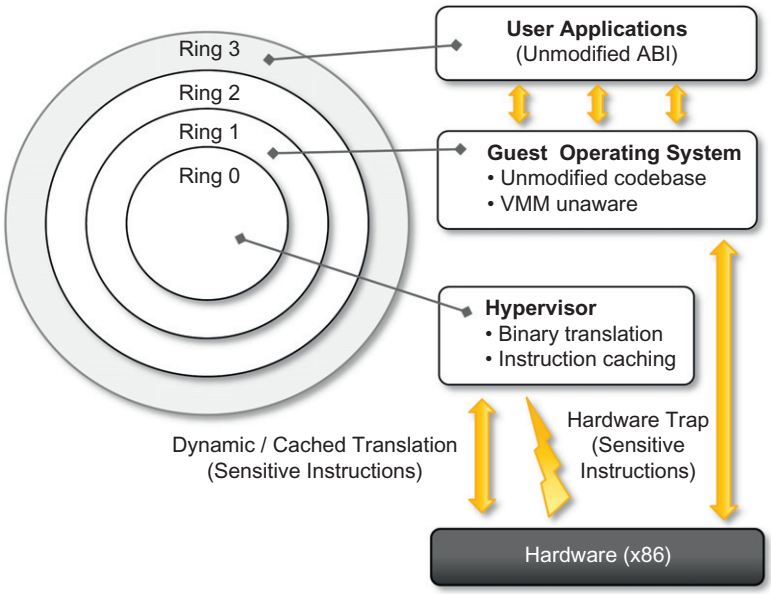

FIGURE 3.12

A full virtualization reference model.

Finally, VMware also provides full virtualization of I/O devices such as network controllers and other peripherals such as keyboard, mouse, disks, and universal serial bus (USB) controllers.

3.6.2.2 Virtualization solutions

VMware is a pioneer in virtualization technology and offers a collection of virtualization solutions covering the entire range of the market, from desktop computing to enterprise computing and infrastructure virtualization.

End-user (desktop) virtualization

VMware supports virtualization of operating system environments and single applications on end-user computers. The first option is the most popular and allows installing a different operating systems and applications in a completely isolated environment from the hosting operating system. Specific VMware software—*VMware Workstation*, for Windows operating systems, and *VMware Fusion*, for Mac OS X environments—is installed in the host operating system to create virtual machines and manage their execution. Besides the creation of an isolated computing environment, the two products allow a guest operating system to leverage the resources of the host machine (USB devices, folder sharing, and integration with the graphical user interface (GUI) of the host operating system). Figure 3.13 provides an overview of the architecture of these systems.

The virtualization environment is created by an application installed in guest operating systems, which provides those operating systems with full hardware virtualization of the underlying

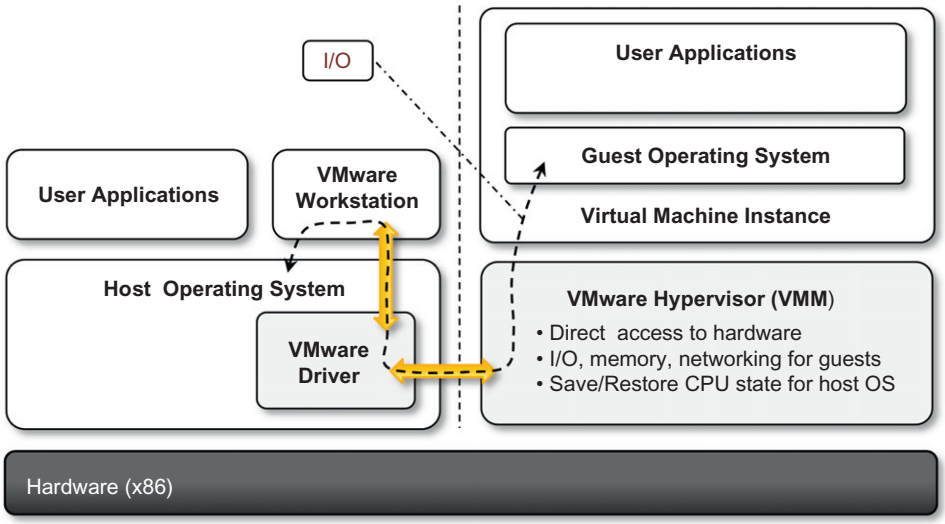

FIGURE 3.13

VMware workstation architecture.

hardware. This is done by installing a specific driver in the host operating system that provides two main services:

- It deploys a virtual machine manager that can run in privileged mode.
- It provides hooks for the VMware application to process specific I/O requests eventually by relaying such requests to the host operating system via system calls.

Using this architecture—also called *Hosted Virtual Machine Architecture*—it is possible to both isolate virtual machine instances within the memory space of a single application and provide reasonable performance, since the intervention of the VMware application is required only for instructions, such as device I/O, that require binary translation. Instructions that can be directly executed are managed by the virtual machine manager, which takes control of the CPU and the MMU and alternates its activity with the host OS. Virtual machine images are saved in a collection of files on the host file system, and both VMware Workstation and VMware Fusion allow creation of new images, pause their execution, create snapshots, and undo operations by rolling back to a previous state of the virtual machine.

Other solutions related to the virtualization of end-user computing environments include VMware Player, VMware ACE, and VMware ThinApp. VMware Player is a reduced version of VMware Workstation that allows creating and playing virtual machines in a Windows or Linux operating environment. VMware ACE, a similar product to VMware Workstation, creates policy-wrapped virtual machines for deploying secure corporate virtual environments on end-user computers. VMware ThinApp is a solution for application virtualization. It provides an isolated environment for applications in order to avoid conflicts due to versioning and incompatible applications. It detects all the changes to the operating environment made by the installation of a specific application and stores them together with the application binary into a package that can be run with VMware ThinApp.

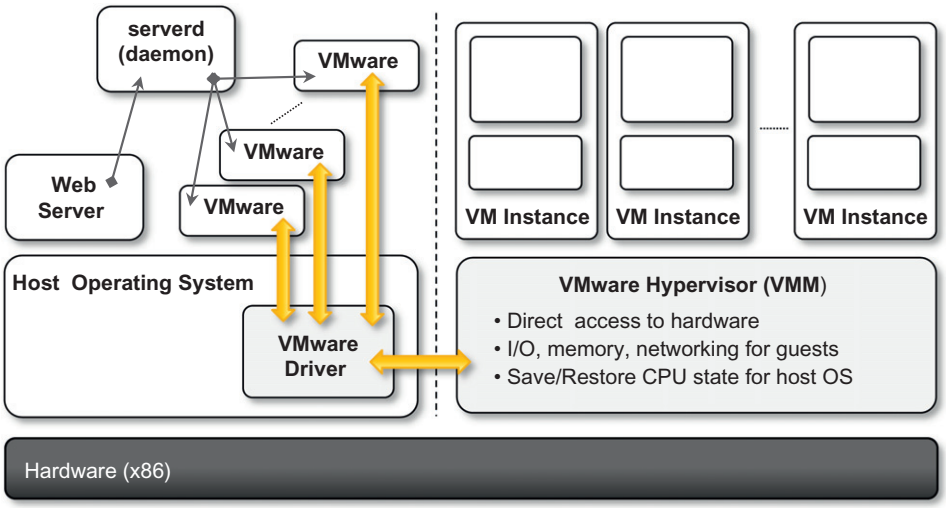

FIGURE 3.14

VMware GSX server architecture.

Server virtualization

VMware provided solutions for server virtualization with different approaches over time. Initial support for server virtualization was provided by VMware GSX server, which replicates the approach used for end-user computers and introduces remote management and scripting capabilities. The architecture of VMware GSX Server is depicted in Figure 3.14.

The architecture is mostly designed to serve the virtualization of Web servers. A daemon process, called *serverd*, controls and manages VMware application processes. These applications are then connected to the virtual machine instances by means of the VMware driver installed on the host operating system. Virtual machine instances are managed by the VMM as described previously. User requests for virtual machine management and provisioning are routed from the Web server through the VMM by means of *serverd*.

VMware ESX Server and its enhanced version, VMWare ESXi Server, are examples of the hypervisor-based approach. Both can be installed on bare metal servers and provide services for virtual machine management. The two solutions provide the same services but differ in the internal architecture, more specifically in the organization of the hypervisor kernel. VMware ESX embeds a modified version of a Linux operating system, which provides access through a service console to hypervisor. VMware ESXi implements a very thin OS layer and replaces the service console with interfaces and services for remote management, thus considerably reducing the hypervisor code size and memory footprint.

The architecture of VMware ESXi is displayed in Figure 3.15. The base of the infrastructure is the VMkernel, which is a thin Portable Operating System Interface (POSIX) compliant operating system that provides the minimal functionality for processes and thread management, file system, I/O stacks, and resource scheduling. The kernel is accessible through specific APIs called User world API. These

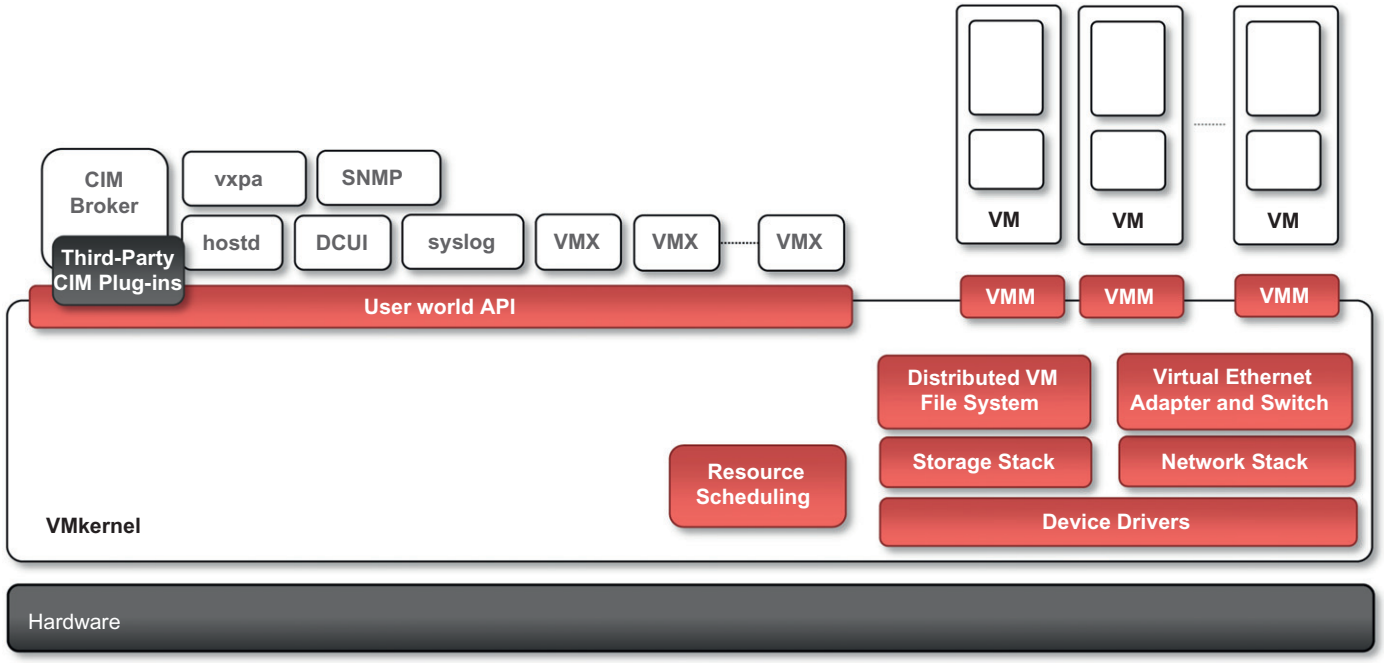

FIGURE 3.15

VMware ESXi server architecture.

APIs are utilized by all the agents that provide supporting activities for the management of virtual machines. Remote management of an ESXi server is provided by the CIM Broker, a system agent that acts as a gateway to the VMkernel for clients by using the *Common Information Model (CIM)*[8] protocol. The ESXi installation can also be managed locally by a *Direct Client User Interface (DCUI)*, which provides a BIOS-like interface for the management of local users.

Infrastructure virtualization and cloud computing solutions

VMware provides a set of products covering the entire stack of cloud computing, from infrastructure management to Software-as-a-Service solutions hosted in the cloud. Figure 3.16 gives an overview of the different solutions offered and how they relate to each other.

ESX and ESXi constitute the building blocks of the solution for virtual infrastructure management: A pool of virtualized servers is tied together and remotely managed as a whole by VMware vSphere. As a virtualization platform it provides a set of basic services besides virtual compute services: Virtual file system, virtual storage, and virtual network constitute the core of the infrastructure; application services, such as virtual machine migration, storage migration, data recovery, and

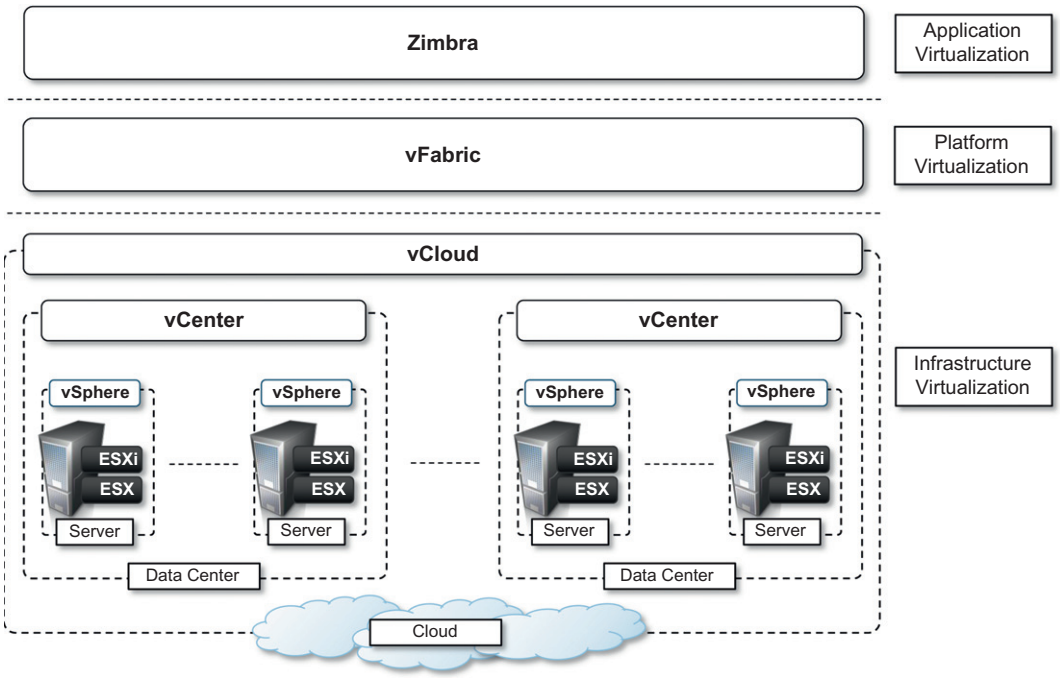

FIGURE 3.16

VMware Cloud Solution stack.

[8]Common Information Model (CIM) is a Distributed Management Task Force standard for defining management information for systems, applications, and services. See http://dmtf.org/standards/cim.

security zones, complete the services offered by vSphere. The management of the infrastructure is operated by VMware vCenter, which provides centralized administration and management of vSphere installations in a data center environment. A collection of virtualized data centers are turned into a Infrastructure-as-a-Service cloud by VMware vCloud, which allows service providers to make available to end users virtual computing environments on demand on a pay-per-use basis. A Web portal provides access to the provisioning services of vCloud, and end users can self-provision virtual machines by choosing from available templates and setting up virtual networks among virtual instances.

VMware also provides a solution for application development in the cloud with VMware vFabric, which is a set of components that facilitate the development of scalable Web applications on top of a virtualized infrastructure. vFabric is a collection of components for application monitoring, scalable data management, and scalable execution and provisioning of Java Web applications.

Finally, at the top of the cloud computing stack, VMware provides Zimbra, a solution for office automation, messaging, and collaboration that is completely hosted in the cloud and accessible from anywhere. This is an SaaS solution that integrates various features into a single software platform providing email and collaboration management.

3.6.2.3 Observations

Initially starting with a solution for fully virtualized x86 hardware, VMware has grown over time and now provides a complete offering for virtualizing hardware, infrastructure, applications, and services, thus covering every segment of the cloud computing market. Even though full x86 virtualization is the core technology of VMware, over time paravirtualization features have been integrated into some of the solutions offered by the vendor, especially after the introduction of hardware-assisted virtualization. For instance, the implementation of some device emulations and the VMware Tools suite that allows enhanced integration with the guest and the host operating environment. Also, VMware has strongly contributed to the development and standardization of a vendor-independent *Virtual Machine Interface (VMI)*, which allows for a general and host-agnostic approach to paravirtualization.

3.6.3 Microsoft Hyper-V

Hyper-V is an infrastructure virtualization solution developed by Microsoft for server virtualization. As the name recalls, it uses a hypervisor-based approach to hardware virtualization, which leverages several techniques to support a variety of guest operating systems. Hyper-V is currently shipped as a component of Windows Server 2008 R2 that installs the hypervisor as a role within the server.

3.6.3.1 Architecture

Hyper-V supports multiple and concurrent execution of guest operating systems by means of *partitions*. A partition is a completely isolated environment in which an operating system is installed and run.

Figure 3.17 provides an overview of the architecture of Hyper-V. Despite its straightforward installation as a component of the host operating system, Hyper-V takes control of the hardware, and the host operating system becomes a virtual machine instance with special privileges, called

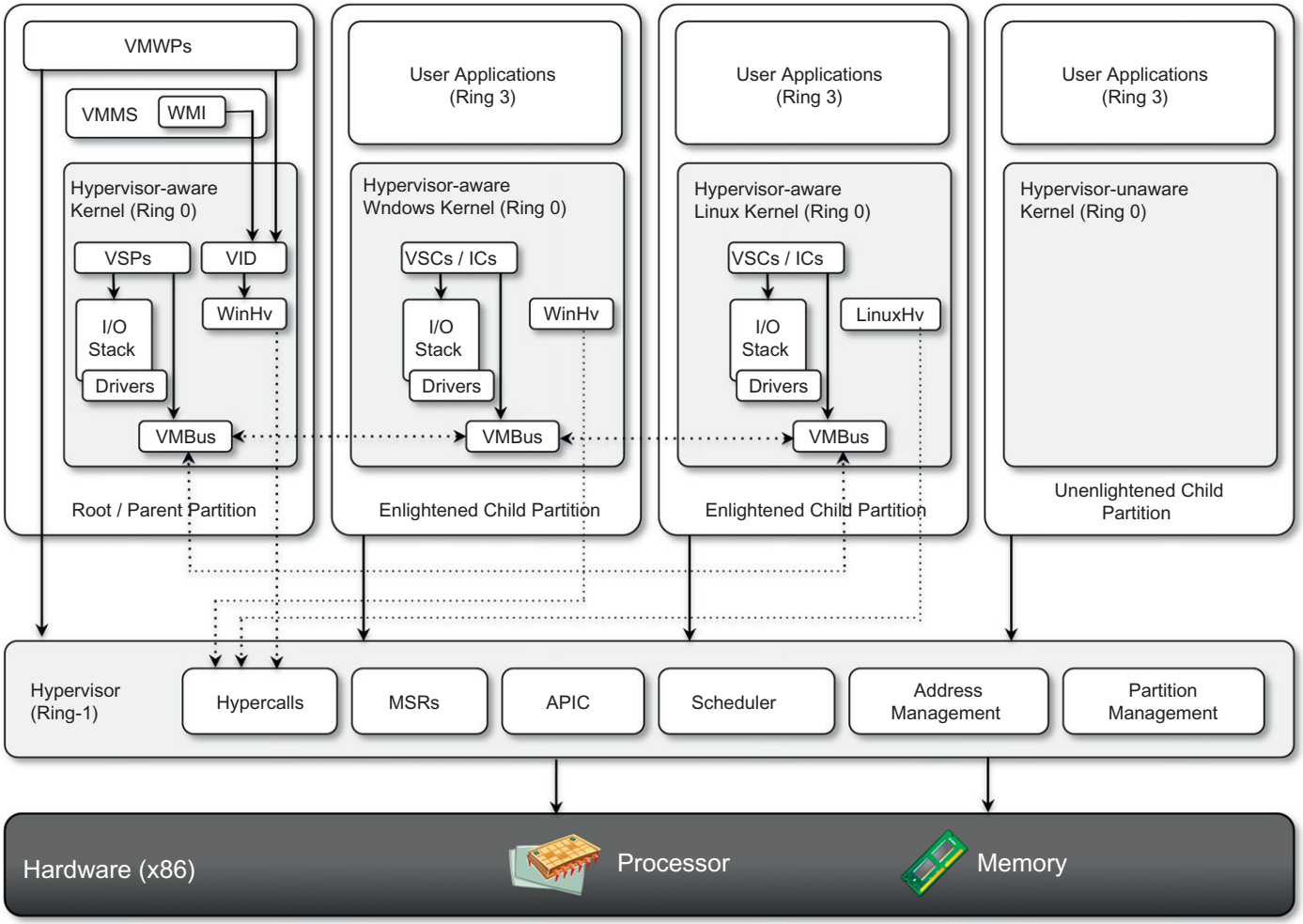

FIGURE 3.17

Microsoft Hyper-V architecture.

the *parent partition.* The parent partition (also called the *root partition*) is the only one that has direct access to the hardware. It runs the virtualization stack, hosts all the drivers required to configure guest operating systems, and creates *child partitions* through the hypervisor. Child partitions are used to host guest operating systems and do not have access to the underlying hardware, but their interaction with it is controlled by either the parent partition or the hypervisor itself.

Hypervisor

The hypervisor is the component that directly manages the underlying hardware (processors and memory). It is logically defined by the following components:

- *Hypercalls interface.* This is the entry point for all the partitions for the execution of sensitive instructions. This is an implementation of the paravirtualization approach already discussed with Xen. This interface is used by drivers in the partitioned operating system to contact the hypervisor using the standard Windows calling convention. The parent partition also uses this interface to create child partitions.
- *Memory service routines (MSRs).* These are the set of functionalities that control the memory and its access from partitions. By leveraging hardware-assisted virtualization, the hypervisor uses the *Input/Output Memory Management Unit* (*I/O MMU* or *IOMMU*) to fast-track access to devices from partitions by translating virtual memory addresses.
- *Advanced programmable interrupt controller (APIC).* This component represents the interrupt controller, which manages the signals coming from the underlying hardware when some event occurs (timer expired, I/O ready, exceptions and traps). Each virtual processor is equipped with a *synthetic interrupt controller (SynIC)*, which constitutes an extension of the local APIC. The hypervisor is responsible of dispatching, when appropriate, the physical interrupts to the synthetic interrupt controllers.
- *Scheduler.* This component schedules the virtual processors to run on available physical processors. The scheduling is controlled by policies that are set by the parent partition.
- *Address manager.* This component is used to manage the virtual network addresses that are allocated to each guest operating system.
- *Partition manager.* This component is in charge of performing partition creation, finalization, destruction, enumeration, and configurations. Its services are available through the hypercalls interface API previously discussed.

The hypervisor runs in Ring -1 and therefore requires corresponding hardware technology that enables such a condition. By executing in this highly privileged mode, the hypervisor can support legacy operating systems that have been designed for x86 hardware. Operating systems of newer generations can take advantage of the new specific architecture of Hyper-V especially for the I/O operations performed by child partitions.

Enlightened I/O and synthetic devices

Enlightened I/O provides an optimized way to perform I/O operations, allowing guest operating systems to leverage an interpartition communication channel rather than traversing the hardware emulation stack provided by the hypervisor. This option is only available to guest operating systems that are hypervisor aware. Enlightened I/O leverages VMBus, an interpartition communication

channel that is used to exchange data between partitions (child and parent) and is utilized mostly for the implementation of virtual device drivers for guest operating systems.

The architecture of Enlightened I/O is described in Figure 3.17. There are three fundamental components: *VMBus*, *Virtual Service Providers (VSPs)*, and *Virtual Service Clients (VSCs)*. VMBus implements the channel and defines the protocol for communication between partitions. VSPs are kernel-level drivers that are deployed in the parent partition and provide access to the corresponding hardware devices. These interact with VSCs, which represent the virtual device drivers (also called *synthetic drivers*) seen by the guest operating systems in the child partitions. Operating systems supported by Hyper-V utilize this preferred communication channel to perform I/O for storage, networking, graphics, and input subsystems. This also results in enhanced performance in child-to-child I/O as a result of virtual networks between guest operating systems. Legacy operating systems, which are not hypervisor aware, can still be run by Hyper-V but rely on device driver emulation, which is managed by the hypervisor and is less efficient.

Parent partition

The parent partition executes the host operating system and implements the virtualization stack that complements the activity of the hypervisor in running guest operating systems. This partition always hosts an instance of the Windows Server 2008 R2, which manages the virtualization stack made available to the child partitions. This partition is the only one that directly accesses device drivers and mediates the access to them by child partitions by hosting the VSPs.

The parent partition is also the one that manages the creation, execution, and destruction of child partitions. It does so by means of the *Virtualization Infrastructure Driver (VID)*, which controls access to the hypervisor and allows the management of virtual processors and memory. For each child partition created, a Virtual Machine Worker Process (VMWP) is instantiated in the parent partition, which manages the child partitions by interacting with the hypervisor through the VID. Virtual Machine Management services are also accessible remotely through a WMI[9] provider that allows remote hosts to access the VID.

Child partitions

Child partitions are used to execute guest operating systems. These are isolated environments that allow secure and controlled execution of guests. Two types of child partition exist, they differ on whether the guest operating system is supported by Hyper-V or not. These are called *Enlightened* and *Unenlightened* partitions, respectively. The first ones can benefit from Enlightened I/O; the other ones are executed by leveraging hardware emulation from the hypervisor.

3.6.3.2 Cloud computing and infrastructure management

Hyper-V constitutes the basic building block of Microsoft virtualization infrastructure. Other components contribute to creating a fully featured platform for server virtualization.

To increase the performance of virtualized environments, a new version of Windows Server 2008, called *Windows Server Core*, has been released. This is a specific version of the operating

[9] *WMI* stands for *Windows Management Instrumentation*. This is a specification used in the Windows environment to provide access to the underlying hardware. The specification is based on providers that give authorized clients access to a specific subsystem of the hardware.

system with a reduced set of features and a smaller footprint. In particular, Windows Server Core has been designed by removing those features, which are not required in a server environment, such as the GUI component and other bulky components such as the .NET Framework and all the applications developed on top of it (for example, PowerShell). This design decision has both advantages and disadvantages. On the plus side, it allows for reduced maintenance (i.e., fewer software patches), reduced attack surface, reduced management, and less disk space. On the negative side, the embedded features are reduced. Still, there is the opportunity to leverage all the "removed features" by means of remote management from a fully featured Windows installation. For instance, administrators can use the PowerShell to remotely manage the Windows Server Core installation through WMI.

Another component that provides advanced management of virtual machines is *System Center Virtual Machine Manager (SCVMM) 2008*. This is a component of the Microsoft System Center suite, which brings into the suite the virtual infrastructure management capabilities from an IT lifecycle point of view. Essentially, SCVMM complements the basic features offered by Hyper-V with management capabilities, including:

- Management portal for the creation and management of virtual instances
- Virtual to Virtual (V2V) and Physical to Virtual (P2V) conversions
- Delegated administration
- Library functionality and deep PowerShell integration
- Intelligent placement of virtual machines in the managed environment
- Host capacity management

SCVMM has also been designed to work with other virtualization platforms such as VMware vSphere (ESX servers) but benefits most from the virtual infrastructure management implemented with Hyper-V.

3.6.3.3 Observations

Compared with Xen and VMware, Hyper-V is a hybrid solution because it leverages both paravirtualization techniques and full hardware virtualization.

The basic architecture of the hypervisor is based on paravirtualized architecture. The hypervisor exposes its services to the guest operating systems by means of hypercalls. Also, paravirtualized kernels can leverage VMBus for fast I/O operations. Moreover, partitions are conceptually similar to domains in Xen: The parent partition maps Domain 0, while child partitions map Domains U. The only difference is that the Xen hypervisor is installed on bare hardware and filters all the access to the underlying hardware, whereas Hyper-V is installed as a role in the existing operating system, and the way it interacts with partitions is quite similar to the strategy implemented by VMware, as we discussed.

The approach adopted by Hyper-V has both advantages and disadvantages. The advantages reside in a flexible virtualization platform supporting a wide range of guest operating systems. The disadvantages are represented by both hardware and software requirements. Hyper-V is compatible only with Windows Server 2008 and newer Windows Server platforms running on a x64 architecture. Moreover, it requires a 64-bit processor supporting hardware-assisted virtualization and data execution prevention. Finally, as noted above, Hyper-V is a role that can be installed on a existing operating system, while vSphere and Xen can be installed on the bare hardware.

SUMMARY

The term *virtualization* is a large umbrella under which a variety of technologies and concepts are classified. The common root of all the forms of virtualization is the ability to provide the illusion of a specific environment, whether a runtime environment, a storage facility, a network connection, or a remote desktop, by using some kind of emulation or abstraction layer. All these concepts play a fundamental role in building cloud computing infrastructure and services in which hardware, IT infrastructure, applications, and services are delivered on demand through the Internet or more generally via a network connection.

Review questions

1. What is virtualization and what are its benefits?
2. What are the characteristics of virtualized environments?
3. Discuss classification or taxonomy of virtualization at different levels.
4. Discuss the machine reference model of execution virtualization.
5. What are hardware virtualization techniques?
6. List and discuss different types of virtualization.
7. What are the benefits of virtualization in the context of cloud computing?
8. What are the disadvantages of virtualization?
9. What is Xen? Discuss its elements for virtualization.
10. Discuss the reference model of full virtualization.
11. Discuss the architecture of Hyper-V. Discuss its use in cloud computing.

This page intentionally left blank

CHAPTER

Cloud Computing Architecture

4

The term *cloud computing* is a wide umbrella encompassing many different things; lately it has become a buzzword that is easily misused to revamp existing technologies and ideas for the public. What makes cloud computing so interesting to IT stakeholders and research practitioners? How does it introduce innovation into the field of distributed computing? This chapter addresses all these questions and characterizes the phenomenon. It provides a reference model that serves as a basis for discussion of cloud computing technologies.

4.1 Introduction

Utility-oriented data centers are the first outcome of cloud computing, and they serve as the infrastructure through which the services are implemented and delivered. Any cloud service, whether virtual hardware, development platform, or application software, relies on a distributed infrastructure owned by the provider or rented from a third party. As noted in the previous definition, the characterization of a cloud is quite general: It can be implemented using a datacenter, a collection of clusters, or a heterogeneous distributed system composed of desktop PCs, workstations, and servers. Commonly, clouds are built by relying on one or more datacenters. In most cases hardware resources are virtualized to provide isolation of workloads and to best exploit the infrastructure. According to the specific service delivered to the end user, different layers can be stacked on top of the virtual infrastructure: a virtual machine manager, a development platform, or a specific application middleware.

As noted in earlier chapters, the cloud computing paradigm emerged as a result of the convergence of various existing models, technologies, and concepts that changed the way we deliver and use IT services. A broad definition of the phenomenon could be as follows:

> *Cloud computing is a utility-oriented and Internet-centric way of delivering IT services on demand. These services cover the entire computing stack: from the hardware infrastructure packaged as a set of virtual machines to software services such as development platforms and distributed applications.*

This definition captures the most important and fundamental aspects of cloud computing. We now discuss a reference model that aids in categorization of cloud technologies, applications, and services.

4.2 The cloud reference model

Cloud computing supports any IT service that can be consumed as a utility and delivered through a network, most likely the Internet. Such characterization includes quite different aspects: infrastructure, development platforms, application and services.

4.2.1 Architecture

It is possible to organize all the concrete realizations of cloud computing into a layered view covering the entire stack (see Figure 4.1), from hardware appliances to software systems. Cloud resources are harnessed to offer "computing horsepower" required for providing services. Often, this layer is implemented using a datacenter in which hundreds and thousands of nodes are stacked together. Cloud infrastructure can be heterogeneous in nature because a variety of resources, such as clusters and even networked PCs, can be used to build it. Moreover, database systems and other storage services can also be part of the infrastructure.

The physical infrastructure is managed by the core middleware, the objectives of which are to provide an appropriate runtime environment for applications and to best utilize resources. At the bottom of the stack, virtualization technologies are used to guarantee runtime environment customization, application isolation, sandboxing, and quality of service. Hardware virtualization is most commonly used at this level. Hypervisors manage the pool of resources and expose the distributed infrastructure as a collection of virtual machines. By using virtual machine technology it is possible to finely partition the hardware resources such as CPU and memory and to virtualize specific devices, thus meeting the requirements of users and applications. This solution is generally paired

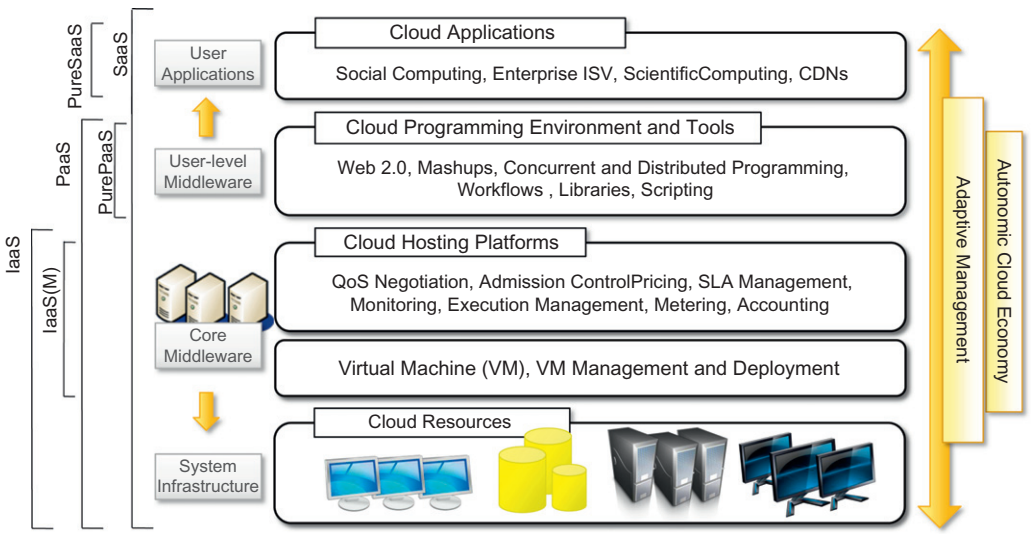

FIGURE 4.1

The cloud computing architecture.

with storage and network virtualization strategies, which allow the infrastructure to be completely virtualized and controlled. According to the specific service offered to end users, other virtualization techniques can be used; for example, programming-level virtualization helps in creating a portable runtime environment where applications can be run and controlled. This scenario generally implies that applications hosted in the cloud be developed with a specific technology or a programming language, such as Java, .NET, or Python. In this case, the user does not have to build its system from bare metal. Infrastructure management is the key function of core middleware, which supports capabilities such as negotiation of the quality of service, admission control, execution management and monitoring, accounting, and billing.

The combination of cloud hosting platforms and resources is generally classified as a *Infrastructure-as-a-Service (IaaS)* solution. We can organize the different examples of IaaS into two categories: Some of them provide both the management layer and the physical infrastructure; others provide only the management layer (*IaaS (M)*). In this second case, the management layer is often integrated with other IaaS solutions that provide physical infrastructure and adds value to them.

IaaS solutions are suitable for designing the system infrastructure but provide limited services to build applications. Such service is provided by cloud programming environments and tools, which form a new layer for offering users a development platform for applications. The range of tools include Web-based interfaces, command-line tools, and frameworks for concurrent and distributed programming. In this scenario, users develop their applications specifically for the cloud by using the API exposed at the user-level middleware. For this reason, this approach is also known as *Platform-as-a-Service (PaaS)* because the service offered to the user is a development platform rather than an infrastructure. PaaS solutions generally include the infrastructure as well, which is bundled as part of the service provided to users. In the case of *Pure PaaS*, only the user-level middleware is offered, and it has to be complemented with a virtual or physical infrastructure.

The top layer of the reference model depicted in Figure 4.1 contains services delivered at the application level. These are mostly referred to as *Software-as-a-Service (SaaS)*. In most cases these are Web-based applications that rely on the cloud to provide service to end users. The horsepower of the cloud provided by IaaS and PaaS solutions allows independent software vendors to deliver their application services over the Internet. Other applications belonging to this layer are those that strongly leverage the Internet for their core functionalities that rely on the cloud to sustain a larger number of users; this is the case of gaming portals and, in general, social networking websites.

As a vision, any service offered in the cloud computing style should be able to adaptively change and expose an autonomic behavior, in particular for its availability and performance. As a reference model, it is then expected to have an adaptive management layer in charge of elastically scaling on demand. SaaS implementations should feature such behavior automatically, whereas PaaS and IaaS generally provide this functionality as a part of the API exposed to users.

The reference model described in Figure 4.1 also introduces the concept of *everything as a Service (XaaS)*. This is one of the most important elements of cloud computing: Cloud services from different providers can be combined to provide a completely integrated solution covering all the computing stack of a system. IaaS providers can offer the bare metal in terms of virtual machines where PaaS solutions are deployed. When there is no need for a PaaS layer, it is possible to directly customize the virtual infrastructure with the software stack needed to run applications. This is the case of virtual Web farms: a distributed system composed of Web servers, database

servers, and load balancers on top of which prepackaged software is installed to run Web applications. This possibility has made cloud computing an interesting option for reducing startups' capital investment in IT, allowing them to quickly commercialize their ideas and grow their infrastructure according to their revenues.

Table 4.1 summarizes the characteristics of the three major categories used to classify cloud computing solutions. In the following section, we briefly discuss these characteristics along with some references to practical implementations.

4.2.2 Infrastructure- and hardware-as-a-service

Infrastructure- and Hardware-as-a-Service (IaaS/HaaS) solutions are the most popular and developed market segment of cloud computing. They deliver customizable infrastructure on demand. The available options within the IaaS offering umbrella range from single servers to entire infrastructures, including network devices, load balancers, and database and Web servers.

The main technology used to deliver and implement these solutions is hardware virtualization: one or more virtual machines opportunely configured and interconnected define the distributed system on top of which applications are installed and deployed. Virtual machines also constitute the atomic components that are deployed and priced according to the specific features of the virtual hardware: memory, number of processors, and disk storage. IaaS/HaaS solutions bring all the benefits of hardware virtualization: workload partitioning, application isolation, sandboxing, and hardware tuning. From the perspective of the service provider, IaaS/HaaS allows better exploiting the IT infrastructure and provides a more secure environment where executing third party applications. From the perspective of the customer it reduces the administration and maintenance cost as well as the capital costs allocated to purchase hardware. At the same time, users can take advantage of the full customization offered by virtualization to deploy their infrastructure in the cloud; in most cases virtual machines come with only the selected operating system installed and the system can be

Table 4.1 Cloud Computing Services Classification

Category	Characteristics	Product Type	Vendors and Products
SaaS	Customers are provided with applications that are accessible anytime and from anywhere.	Web applications and services (Web 2.0)	SalesForce.com (CRM) Clarizen.com (project management) Google Apps
PaaS	Customers are provided with a platform for developing applications hosted in the cloud.	Programming APIs and frameworks Deployment systems	Google AppEngine Microsoft Azure Manjrasoft Aneka Data Synapse
IaaS/HaaS	Customers are provided with virtualized hardware and storage on top of which they can build their infrastructure.	Virtual machine management infrastructure Storage management Network management	Amazon EC2 and S3 GoGrid Nirvanix

configured with all the required packages and applications. Other solutions provide prepackaged system images that already contain the software stack required for the most common uses: Web servers, database servers, or LAMP[1] stacks. Besides the basic virtual machine management capabilities, additional services can be provided, generally including the following: SLA resource-based allocation, workload management, support for infrastructure design through advanced Web interfaces, and the ability to integrate third-party IaaS solutions.

Figure 4.2 provides an overall view of the components forming an Infrastructure-as-a-Service solution. It is possible to distinguish three principal layers: the *physical infrastructure*, the *software management infrastructure*, and the *user interface*. At the top layer the user interface provides access to the services exposed by the software management infrastructure. Such an interface is

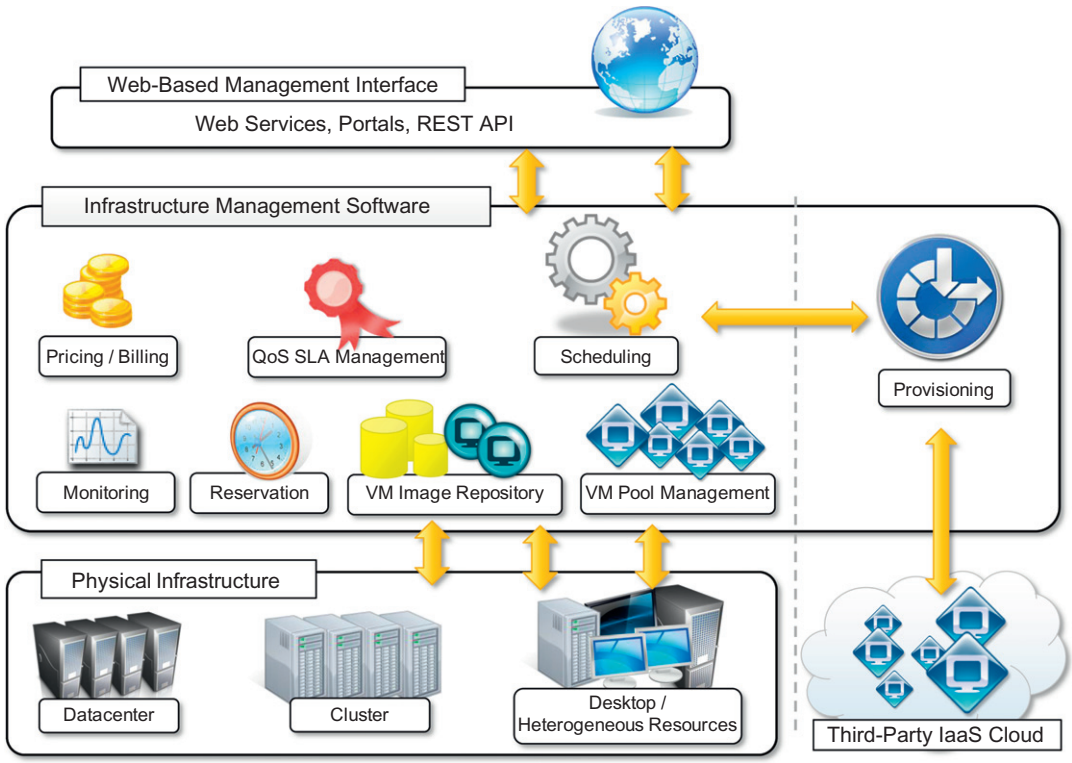

FIGURE 4.2

Infrastructure-as-a-Service reference implementation.

[1] *LAMP* is an acronym for *Linux Apache MySql and PHP* and identifies a specific server configuration running the Linux operating system, featuring Apache as Web server, MySQL as database server, and PHP: Hypertext Preprocessor (PHP) as server-side scripting technology for developing Web applications. LAMP stacks are the most common packaged solutions for quickly deploying Web applications.

generally based on Web 2.0 technologies: Web services, RESTful APIs, and mash-ups. These technologies allow either applications or final users to access the services exposed by the underlying infrastructure. Web 2.0 applications allow developing full-featured management consoles completely hosted in a browser or a Web page. Web services and RESTful APIs allow programs to interact with the service without human intervention, thus providing complete integration within a software system. The core features of an IaaS solution are implemented in the infrastructure management software layer. In particular, management of the virtual machines is the most important function performed by this layer. A central role is played by the scheduler, which is in charge of allocating the execution of virtual machine instances. The scheduler interacts with the other components that perform a variety of tasks:

- The *pricing and billing* component takes care of the cost of executing each virtual machine instance and maintains data that will be used to charge the user.
- The *monitoring* component tracks the execution of each virtual machine instance and maintains data required for reporting and analyzing the performance of the system.
- The *reservation* component stores the information of all the virtual machine instances that have been executed or that will be executed in the future.
- If support for QoS-based execution is provided, a *QoS/SLA management* component will maintain a repository of all the SLAs made with the users; together with the monitoring component, this component is used to ensure that a given virtual machine instance is executed with the desired quality of service.
- The *VM repository* component provides a catalog of virtual machine images that users can use to create virtual instances. Some implementations also allow users to upload their specific virtual machine images.
- A *VM pool manager* component is responsible for keeping track of all the live instances.
- Finally, if the system supports the integration of additional resources belonging to a third-party IaaS provider, a *provisioning* component interacts with the scheduler to provide a virtual machine instance that is external to the local physical infrastructure directly managed by the pool.

The bottom layer is composed of the physical infrastructure, on top of which the management layer operates. As previously discussed, the infrastructure can be of different types; the specific infrastructure used depends on the specific use of the cloud. A service provider will most likely use a massive datacenter containing hundreds or thousands of nodes. A cloud infrastructure developed in house, in a small or medium-sized enterprise or within a university department, will most likely rely on a cluster. At the bottom of the scale it is also possible to consider a heterogeneous environment where different types of resources—PCs, workstations, and clusters—can be aggregated. This case mostly represents an evolution of desktop grids where any available computing resource (such as PCs and workstations that are idle outside of working hours) is harnessed to provide a huge compute power. From an architectural point of view, the physical layer also includes the virtual resources that are rented from external IaaS providers.

In the case of complete IaaS solutions, all three levels are offered as service. This is generally the case with public clouds vendors such as Amazon, GoGrid, Joyent, Rightscale, Terremark, Rackspace, ElasticHosts, and Flexiscale, which own large datacenters and give access to their computing infrastructures using an IaaS approach. Other solutions instead cover only the user interface

and the infrastructure software management layers. They need to provide credentials to access third-party IaaS providers or to own a private infrastructure in which the management software is installed. This is the case with Enomaly, Elastra, Eucalyptus, OpenNebula, and specific IaaS (M) solutions from VMware, IBM, and Microsoft.

The proposed architecture only represents a reference model for IaaS implementations. It has been used to provide general insight into the most common features of this approach for providing cloud computing services and the operations commonly implemented at this level. Different solutions can feature additional services or even not provide support for some of the features discussed here. Finally, the reference architecture applies to IaaS implementations that provide computing resources, especially for the scheduling component. If storage is the main service provided, it is still possible to distinguish these three layers. The role of infrastructure management software is not to keep track and manage the execution of virtual machines but to provide access to large infrastructures and implement storage virtualization solutions on top of the physical layer.

4.2.3 Platform as a service

Platform-as-a-Service (PaaS) solutions provide a development and deployment platform for running applications in the cloud. They constitute the middleware on top of which applications are built. A general overview of the features characterizing the PaaS approach is given in Figure 4.3.

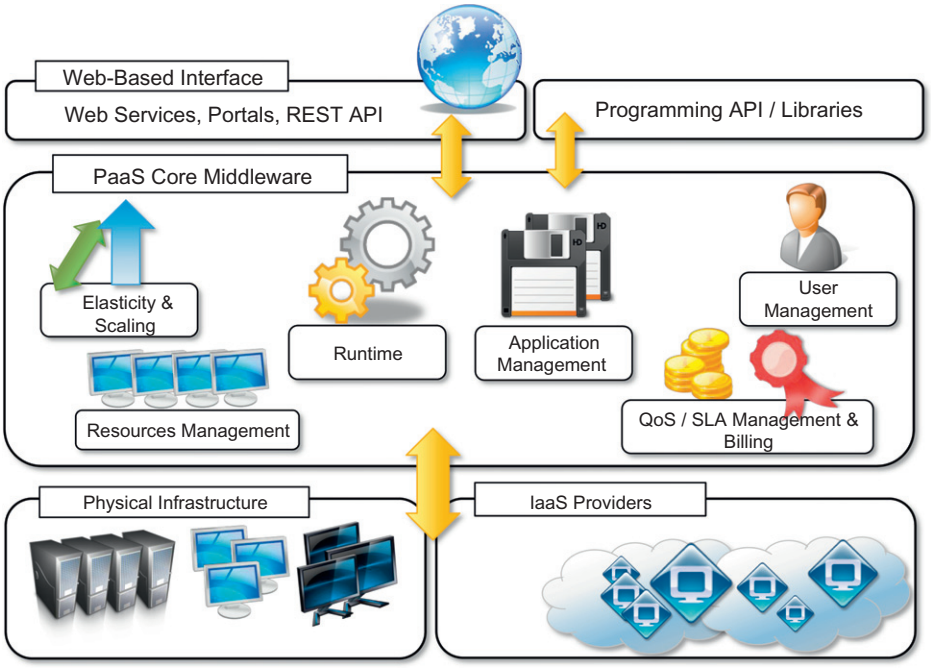

FIGURE 4.3

The Platform-as-a-Service reference model.

Application management is the core functionality of the middleware. PaaS implementations provide applications with a runtime environment and do not expose any service for managing the underlying infrastructure. They automate the process of deploying applications to the infrastructure, configuring application components, provisioning and configuring supporting technologies such as load balancers and databases, and managing system change based on policies set by the user. Developers design their systems in terms of applications and are not concerned with hardware (physical or virtual), operating systems, and other low-level services. The core middleware is in charge of managing the resources and scaling applications on demand or automatically, according to the commitments made with users. From a user point of view, the core middleware exposes interfaces that allow programming and deploying applications on the cloud. These can be in the form of a Web-based interface or in the form of programming APIs and libraries.

The specific development model decided for applications determines the interface exposed to the user. Some implementations provide a completely Web-based interface hosted in the cloud and offering a variety of services. It is possible to find integrated developed environments based on 4GL and visual programming concepts, or rapid prototyping environments where applications are built by assembling mash-ups and user-defined components and successively customized. Other implementations of the PaaS model provide a complete object model for representing an application and provide a programming language-based approach. This approach generally offers more flexibility and opportunities but incurs longer development cycles. Developers generally have the full power of programming languages such as Java, .NET, Python, or Ruby, with some restrictions to provide better scalability and security. In this case the traditional development environments can be used to design and develop applications, which are then deployed on the cloud by using the APIs exposed by the PaaS provider. Specific components can be offered together with the development libraries for better exploiting the services offered by the PaaS environment. Sometimes a local runtime environment that simulates the conditions of the cloud is given to users for testing their applications before deployment. This environment can be restricted in terms of features, and it is generally not optimized for scaling.

PaaS solutions can offer middleware for developing applications together with the infrastructure or simply provide users with the software that is installed on the user premises. In the first case, the PaaS provider also owns large datacenters where applications are executed; in the second case, referred to in this book as *Pure PaaS*, the middleware constitutes the core value of the offering. It is also possible to have vendors that deliver both middleware and infrastructure and ship only the middleware for private installations.

Table 4.2 provides a classification of the most popular PaaS implementations. It is possible to organize the various solutions into three wide categories: *PaaS-I*, *PaaS-II*, and *PaaS-III*. The first category identifies PaaS implementations that completely follow the cloud computing style for application development and deployment. They offer an integrated development environment hosted within the Web browser where applications are designed, developed, composed, and deployed. This is the case of Force.com and Longjump. Both deliver as platforms the combination of middleware and infrastructure. In the second class we can list all those solutions that are focused on providing a scalable infrastructure for Web application, mostly websites. In this case, developers generally use the providers' APIs, which are built on top of industrial runtimes, to develop

Table 4.2 Platform-as-a-Service Offering Classification

Category	Description	Product Type	Vendors and Products
PaaS-I	Runtime environment with Web-hosted application development platform. Rapid application prototyping.	Middleware + Infrastructure Middleware + Infrastructure	Force.com Longjump
PaaS-II	Runtime environment for scaling Web applications. The runtime could be enhanced by additional components that provide scaling capabilities.	Middleware + Infrastructure Middleware Middleware + Infrastructure Middleware + Infrastructure Middleware + Infrastructure Middleware	Google AppEngine AppScale Heroku Engine Yard Joyent Smart Platform GigaSpaces XAP
PaaS-III	Middleware and programming model for developing distributed applications in the cloud.	Middleware + Infrastructure Middleware Middleware Middleware Middleware Middleware	Microsoft Azure DataSynapse Cloud IQ Manjrasof Aneka Apprenda SaaSGrid GigaSpaces DataGrid

applications. Google AppEngine is the most popular product in this category. It provides a scalable runtime based on the Java and Python programming languages, which have been modified for providing a secure runtime environment and enriched with additional APIs and components to support scalability. AppScale, an open-source implementation of Google AppEngine, provides interface-compatible middleware that has to be installed on a physical infrastructure. Joyent Smart Platform provides a similar approach to Google AppEngine. A different approach is taken by Heroku and Engine Yard, which provide scalability support for Ruby- and Ruby on Rails-based Websites. In this case developers design and create their applications with the traditional methods and then deploy them by uploading to the provider's platform.

The third category consists of all those solutions that provide a cloud programming platform for any kind of application, not only Web applications. Among these, the most popular is Microsoft Windows Azure, which provides a comprehensive framework for building service-oriented cloud applications on top of the .NET technology, hosted on Microsoft's datacenters. Other solutions in the same category, such as Manjrasoft Aneka, Apprenda SaaSGrid, Appistry Cloud IQ Platform, DataSynapse, and GigaSpaces DataGrid, provide only middleware with different services. Table 4.2 shows only a few options available in the Platform-as-a-Service market segment.

The PaaS umbrella encompasses a variety of solutions for developing and hosting applications in the cloud. Despite this heterogeneity, it is possible to identify some criteria that are expected to

be found in any implementation. As noted by Sam Charrington, product manager at Appistry.com,[2] there are some essential characteristics that identify a PaaS solution:

- *Runtime framework*. This framework represents the "software stack" of the PaaS model and the most intuitive aspect that comes to people's minds when they refer to PaaS solutions. The runtime framework executes end-user code according to the policies set by the user and the provider.
- *Abstraction*. PaaS solutions are distinguished by the higher level of abstraction that they provide. Whereas in the case of IaaS solutions the focus is on delivering "raw" access to virtual or physical infrastructure, in the case of PaaS the focus is on the applications the cloud must support. This means that PaaS solutions offer a way to deploy and manage applications on the cloud rather than a bunch of virtual machines on top of which the IT infrastructure is built and configured.
- *Automation*. PaaS environments automate the process of deploying applications to the infrastructure, scaling them by provisioning additional resources when needed. This process is performed automatically and according to the SLA made between the customers and the provider. This feature is normally not native in IaaS solutions, which only provide ways to provision more resources.
- *Cloud services*. PaaS offerings provide developers and architects with services and APIs, helping them to simplify the creation and delivery of elastic and highly available cloud applications. These services are the key differentiators among competing PaaS solutions and generally include specific components for developing applications, advanced services for application monitoring, management, and reporting.

Another essential component for a PaaS-based approach is the ability to integrate third-party cloud services offered from other vendors by leveraging service-oriented architecture. Such integration should happen through standard interfaces and protocols. This opportunity makes the development of applications more agile and able to evolve according to the needs of customers and users. Many of the PaaS offerings provide this facility, which is naturally built into the framework they leverage to provide a cloud computing solution.

One of the major concerns of leveraging PaaS solutions for implementing applications is *vendor lock-in*. Differently from IaaS solutions, which deliver bare virtual servers that can be fully customized in terms of the software stack installed, PaaS environments deliver a platform for developing applications, which exposes a well-defined set of APIs and, in most cases, binds the application to the specific runtime of the PaaS provider. Even though a platform-based approach strongly simplifies the development and deployment cycle of applications, it poses the risk of making these applications completely dependent on the provider. Such dependency can become a significant obstacle in retargeting the application to another environment and runtime if the commitments made with the provider cease. The impact of the vendor lock-in on applications obviously varies according to the various solutions. Some of them, such as Force.com, rely on a proprietary runtime framework, which makes the retargeting process very difficult. Others, such as Google AppEngine and Microsoft Azure, rely on industry-standard runtimes but utilize private data storage facilities and

[2]The full detail of this analysis can be found in the Cloud-pulse blog post available at the following address: http://Cloudpulseblog.com/2010/02/the-essential-characteristics-of-paas.

computing infrastructure. In this case it is possible to find alternatives based on PaaS solutions implementing the same interfaces, with perhaps different performance. Others, such as Appistry Cloud IQ Platform, Heroku, and Engine Yard, completely rely on open standards, thus making the migration of applications easier.

Finally, from a financial standpoint, although IaaS solutions allow shifting the capital cost into operational costs through outsourcing, PaaS solutions can cut the cost across development, deployment, and management of applications. It helps management reduce the risk of ever-changing technologies by offloading the cost of upgrading the technology to the PaaS provider. This happens transparently for the consumers of this model, who can concentrate their effort on the core value of their business. The PaaS approach, when bundled with underlying IaaS solutions, helps even small start-up companies quickly offer customers integrated solutions on a hosted platform at a very minimal cost. These opportunities make the PaaS offering a viable option that targets different market segments.

4.2.4 Software as a service

Software-as-a-Service (SaaS) is a software delivery model that provides access to applications through the Internet as a Web-based service. It provides a means to free users from complex hardware and software management by offloading such tasks to third parties, which build applications accessible to multiple users through a Web browser. In this scenario, customers neither need install anything on their premises nor have to pay considerable up-front costs to purchase the software and the required licenses. They simply access the application website, enter their credentials and billing details, and can instantly use the application, which, in most of the cases, can be further customized for their needs. On the provider side, the specific details and features of each customer's application are maintained in the infrastructure and made available on demand.

The SaaS model is appealing for applications serving a wide range of users and that can be adapted to specific needs with little further customization. This requirement characterizes SaaS as a "one-to-many" software delivery model, whereby an application is shared across multiple users. This is the case of CRM[3] and ERP[4] applications that constitute common needs for almost all enterprises, from small to medium-sized and large business. Every enterprise will have the same requirements for the basic features concerning CRM and ERP; different needs can be satisfied with further customization. This scenario facilitates the development of software platforms that provide a general set of features and support specialization and ease of integration of new components. Moreover, it constitutes the perfect candidate for hosted solutions, since the applications delivered to the user are the same, and the applications themselves provide users with the means to shape the

[3]CRM is an acronym for *customer relationship management* and identifies concerns related to interactions with customers and prospect sales. CRM solutions are software systems that simplify the process of managing customers and identifying sales strategies.

[4]ERP, an acronym for *enterprise resource planning*, generally refers to an integrated computer-based system used to manage internal and external resources, including tangible assets, materials, and financial and human resources. ERP software provides an integrated view of the enterprise and facilitates the management of the information flows between business functions and resources.

applications according to user needs. As a result, SaaS applications are naturally multitenant. *Multitenancy*, which is a feature of SaaS compared to traditional packaged software, allows providers to centralize and sustain the effort of managing large hardware infrastructures, maintaining and upgrading applications transparently to the users, and optimizing resources by sharing the costs among the large user base. On the customer side, such costs constitute a minimal fraction of the usage fee paid for the software.

As noted previously (see Section 1.2), the concept of software as a service preceded cloud computing, starting to circulate at the end of the 1990s, when it began to gain marketplace acceptance [31]. The acronym SaaS was then coined in 2001 by the *Software Information & Industry Association (SIIA)* [32] with the following connotation:

> *In the software as a service model, the application, or service, is deployed from a centralized datacenter across a network—Internet, Intranet, LAN, or VPN—providing access and use on a recurring fee basis. Users "rent," "subscribe to," "are assigned," or "are granted access to" the applications from a central provider. Business models vary according to the level to which the software is streamlined, to lower price and increase efficiency, or value-added through customization to further improve digitized business processes.*

The analysis carried out by SIIA was mainly oriented to cover application service providers (ASPs) and all their variations, which capture the concept of software applications consumed as a service in a broader sense. ASPs already had some of the core characteristics of SaaS:

- The product sold to customer is *application access*.
- The application is centrally managed.
- The service delivered is *one-to-many*.
- The service delivered is an integrated solution *delivered on the contract*, which means provided as promised.

Initially ASPs offered hosting solutions for packaged applications, which were served to multiple customers. Successively, other options, such as Web-based integration of third-party application services, started to gain interest and a new range of opportunities open up to independent software vendors and service providers. These opportunities eventually evolved into a more flexible model to deliver applications as a service: the SaaS model. ASPs provided access to packaged software solutions that addressed the needs of a variety of customers. Initially this approach was affordable for service providers, but it later became inconvenient when the cost of customizations and specializations increased. The SaaS approach introduces a more flexible way of delivering application services that are fully customizable by the user by integrating new services, injecting their own components, and designing the application and information workflows. Such a new approach has also been possible with the support of Web 2.0 technologies, which allowed turning the Web browser into a full-featured interface, able even to support application composition and development.

How is cloud computing related to SaaS? According to the classification of services shown in Figure 4.1, the SaaS approach lays on top of the cloud computing stack. It fits into the cloud computing vision expressed by the *XaaS* acronym, Everything-as-a-Service; and with SaaS, applications

are delivered as a service. Initially the SaaS model was of interest only for lead users and early adopters. The benefits delivered at that stage were the following:

- Software cost reduction and total cost of ownership (TCO) were paramount
- Service-level improvements
- Rapid implementation
- Standalone and configurable applications
- Rudimentary application and data integration
- Subscription and pay-as-you-go (PAYG) pricing

With the advent of cloud computing there has been an increasing acceptance of SaaS as a viable software delivery model. This led to transition into SaaS 2.0 [40], which does not introduce a new technology but transforms the way in which SaaS is used.

In particular, SaaS 2.0 is focused on providing a more robust infrastructure and application platforms driven by SLAs. Rather than being characterized as a more rapid implementation and deployment environment, SaaS 2.0 will focus on the rapid achievement of business objectives. This is why such evolution does not introduce any new technology: The existing technologies are composed together in order to achieve business goals efficiently. Fundamental to this perspective is the ability to leverage existing solutions and integrate value-added business services. The existing SaaS infrastructures not only allow the development and customization of applications, but they also facilitate the integration of services that are exposed by other parties. SaaS applications are then the result of the interconnection and the synergy of different applications and components that together provide customers with added value. This approach dramatically changes the software ecosystem of the SaaS market, which is no longer monopolized by a few vendors but is now a fully interconnected network of service providers, clustered around some "big hubs" that deliver the application to the customer. In this scenario, each single component integrated into the SaaS application becomes responsible to the user for ensuring the attached SLA and at the same time could be priced differently. Customers can then choose how to specialize their applications by deciding which components and services they want to integrate.

Software-as-a-Service applications can serve different needs. CRM, ERP, and social networking applications are definitely the most popular ones. SalesForce.com is probably the most successful and popular example of a CRM service. It provides a wide range of services for applications: customer relationship and human resource management, enterprise resource planning, and many other features. SalesForce.com builds on top of the Force.com platform, which provides a fully featured environment for building applications. It offers either a programming language or a visual environment to arrange components together for building applications. In addition to the basic features provided, the integration with third-party-made applications enriches SalesForce.com's value. In particular, through AppExchange customers can publish, search, and integrate new services and features into their existing applications. This makes SalesForce.com applications completely extensible and customizable. Similar solutions are offered by NetSuite and RightNow. NetSuite is an integrated software business suite featuring financials, CRM, inventory, and ecommerce functionalities integrated all together. RightNow is customer experience-centered SaaS application that integrates together different features, from chat to Web communities, to support the common activity of an enterprise.

Another important class of popular SaaS applications comprises social networking applications such as Facebook and professional networking sites such as LinkedIn. Other than providing the basic features of networking, they allow incorporating and extending their capabilities by integrating third-party applications. These can be developed as plug-ins for the hosting platform, as happens for Facebook, and made available to users, who can select which applications they want to add to their profile. As a result, the integrated applications get full access to the network of contacts and users' profile data. The nature of these applications can be of different types: office automation components, games, or integration with other existing services.

Office automation applications are also an important representative for SaaS applications: Google Documents and Zoho Office are examples of Web-based applications that aim to address all user needs for documents, spreadsheets, and presentation management. They offer a Web-based interface for creating, managing, and modifying documents that can be easily shared among users and made accessible from anywhere.

It is important to note the role of SaaS solution enablers, which provide an environment in which to integrate third-party services and share information with others. A quite successful example is Box.net, an SaaS application providing users with a Web space and profile that can be enriched and extended with third-party applications such as office automation, integration with CRM-based solutions, social Websites, and photo editing.

4.3 Types of clouds

Clouds constitute the primary outcome of cloud computing. They are a type of parallel and distributed system harnessing physical and virtual computers presented as a unified computing resource. Clouds build the infrastructure on top of which services are implemented and delivered to customers. Such infrastructures can be of different types and provide useful information about the nature and the services offered by the cloud. A more useful classification is given according to the administrative domain of a cloud: It identifies the boundaries within which cloud computing services are implemented, provides hints on the underlying infrastructure adopted to support such services, and qualifies them. It is then possible to differentiate four different types of cloud:

- *Public clouds*. The cloud is open to the wider public.
- *Private clouds*. The cloud is implemented within the private premises of an institution and generally made accessible to the members of the institution or a subset of them.
- *Hybrid or heterogeneous clouds*. The cloud is a combination of the two previous solutions and most likely identifies a private cloud that has been augmented with resources or services hosted in a public cloud.
- *Community clouds*. The cloud is characterized by a multi-administrative domain involving different deployment models (public, private, and hybrid), and it is specifically designed to address the needs of a specific industry.

Almost all the implementations of clouds can be classified in this categorization. In the following sections, we provide brief characterizations of these clouds.

4.3.1 Public clouds

Public clouds constitute the first expression of cloud computing. They are a realization of the canonical view of cloud computing in which the services offered are made available to anyone, from anywhere, and at any time through the Internet. From a structural point of view they are a distributed system, most likely composed of one or more datacenters connected together, on top of which the specific services offered by the cloud are implemented. Any customer can easily sign in with the cloud provider, enter her credential and billing details, and use the services offered.

Historically, public clouds were the first class of cloud that were implemented and offered. They offer solutions for minimizing IT infrastructure costs and serve as a viable option for handling peak loads on the local infrastructure. They have become an interesting option for small enterprises, which are able to start their businesses without large up-front investments by completely relying on public infrastructure for their IT needs. What made attractive public clouds compared to the reshaping of the private premises and the purchase of hardware and software was the ability to grow or shrink according to the needs of the related business. By renting the infrastructure or subscribing to application services, customers were able to dynamically upsize or downsize their IT according to the demands of their business. Currently, public clouds are used both to completely replace the IT infrastructure of enterprises and to extend it when it is required.

A fundamental characteristic of public clouds is multitenancy. A public cloud is meant to serve a multitude of users, not a single customer. Any customer requires a virtual computing environment that is separated, and most likely isolated, from other users. This is a fundamental requirement to provide effective monitoring of user activities and guarantee the desired performance and the other QoS attributes negotiated with users. QoS management is a very important aspect of public clouds. Hence, a significant portion of the software infrastructure is devoted to monitoring the cloud resources, to bill them according to the contract made with the user, and to keep a complete history of cloud usage for each customer. These features are fundamental to public clouds because they help providers offer services to users with full accountability.

A public cloud can offer any kind of service: infrastructure, platform, or applications. For example, Amazon EC2 is a public cloud that provides infrastructure as a service; Google AppEngine is a public cloud that provides an application development platform as a service; and SalesForce.com is a public cloud that provides software as a service. What makes public clouds peculiar is the way they are consumed: They are available to everyone and are generally architected to support a large quantity of users. What characterizes them is their natural ability to scale on demand and sustain peak loads.

From an architectural point of view there is no restriction concerning the type of distributed system implemented to support public clouds. Most likely, one or more datacenters constitute the physical infrastructure on top of which the services are implemented and delivered. Public clouds can be composed of geographically dispersed datacenters to share the load of users and better serve them according to their locations. For example, Amazon Web Services has datacenters installed in the United States, Europe, Singapore, and Australia; they allow their customers to choose between three different regions: *us-west-1*, *us-east-1*, or *eu-west-1*. Such regions are priced differently and are further divided into availability zones, which map to specific datacenters. According to the specific class of services delivered by the cloud, a different software stack is installed to manage the infrastructure: virtual machine managers, distributed middleware, or distributed applications.

4.3.2 Private clouds

Public clouds are appealing and provide a viable option to cut IT costs and reduce capital expenses, but they are not applicable in all scenarios. For example, a very common critique to the use of cloud computing in its canonical implementation is the *loss of control*. In the case of public clouds, the provider is in control of the infrastructure and, eventually, of the customers' core logic and sensitive data. Even though there could be regulatory procedure in place that guarantees fair management and respect of the customer's privacy, this condition can still be perceived as a threat or as an unacceptable risk that some organizations are not willing to take. In particular, institutions such as government and military agencies will not consider public clouds as an option for processing or storing their sensitive data. The risk of a breach in the security infrastructure of the provider could expose such information to others; this could simply be considered unacceptable.

In other cases, the loss of control of where your virtual IT infrastructure resides could open the way to other problematic situations. More precisely, the geographical location of a datacenter generally determines the regulations that are applied to management of digital information. As a result, according to the specific location of data, some sensitive information can be made accessible to government agencies or even considered outside the law if processed with specific cryptographic techniques. For example, the USA PATRIOT Act[5] provides its government and other agencies with virtually limitless powers to access information, including that belonging to any company that stores information in the U.S. territory. Finally, existing enterprises that have large computing infrastructures or large installed bases of software do not simply want to switch to public clouds, but they use the existing IT resources and optimize their revenue. All these aspects make the use of a public computing infrastructure not always possible. Yet the general idea supported by the cloud computing vision can still be attractive. More specifically, having an infrastructure able to deliver IT services on demand can still be a winning solution, even when implemented within the private premises of an institution. This idea led to the diffusion of private clouds, which are similar to public clouds, but their resource-provisioning model is limited within the boundaries of an organization.

Private clouds are virtual distributed systems that rely on a private infrastructure and provide internal users with dynamic provisioning of computing resources. Instead of a pay-as-you-go model as in public clouds, there could be other schemes in place, taking into account the usage of the cloud and proportionally billing the different departments or sections of an enterprise. Private clouds have the advantage of keeping the core business operations in-house by relying on the existing IT infrastructure and reducing the burden of maintaining it once the cloud has been set up. In this scenario, security concerns are less critical, since sensitive information does not flow out of the private infrastructure. Moreover, existing IT resources can be better utilized because the private cloud can provide services to a different range of users. Another interesting opportunity that comes with private clouds is the possibility of testing applications and systems at a comparatively lower

[5]The USA PATRIOT Act is a statute enacted by the U.S. government that increases the ability of law enforcement agencies to search telephone, email, medical, financial, and other records and eases restrictions on foreign intelligence gathering within the United States. The full text of the act is available at the Website of the Library of the Congress at the following address: http://thomas.loc.gov/cgi-bin/bdquery/z?d107:hr03162: (accessed April 20, 2010).

price rather than public clouds before deploying them on the public virtual infrastructure. A Forrester report [34] on the benefits of delivering in-house cloud computing solutions for enterprises highlighted some of the key advantages of using a private cloud computing infrastructure:

- *Customer information protection*. Despite assurances by the public cloud leaders about security, few provide satisfactory disclosure or have long enough histories with their cloud offerings to provide warranties about the specific level of security put in place on their systems. In-house security is easier to maintain and rely on.
- *Infrastructure ensuring SLAs*. Quality of service implies specific operations such as appropriate clustering and failover, data replication, system monitoring and maintenance, and disaster recovery, and other uptime services can be commensurate to the application needs. Although public cloud vendors provide some of these features, not all of them are available as needed.
- *Compliance with standard procedures and operations*. If organizations are subject to third-party compliance standards, specific procedures have to be put in place when deploying and executing applications. This could be not possible in the case of the virtual public infrastructure.

All these aspects make the use of cloud-based infrastructures in private premises an interesting option.

From an architectural point of view, private clouds can be implemented on more heterogeneous hardware: They generally rely on the existing IT infrastructure already deployed on the private premises. This could be a datacenter, a cluster, an enterprise desktop grid, or a combination of them. The physical layer is complemented with infrastructure management software (i.e., IaaS (M); see Section 4.2.2) or a PaaS solution, according to the service delivered to the users of the cloud.

Different options can be adopted to implement private clouds. Figure 4.4 provides a comprehensive view of the solutions together with some reference to the most popular software used to deploy private clouds. At the bottom layer of the software stack, virtual machine technologies such as Xen [35], KVM [36], and VMware serve as the foundations of the cloud. Virtual machine management technologies such as VMware vCloud, Eucalyptus [37], and OpenNebula [38] can be used to control the virtual infrastructure and provide an IaaS solution. VMware vCloud is a proprietary solution, but Eucalyptus provides full compatibility with Amazon Web Services interfaces and supports different virtual machine technologies such as Xen, KVM, and VMware. Like Eucalyptus, OpenNebula is an open-source solution for virtual infrastructure management that supports KVM, Xen, and VMware, which has been designed to easily integrate third-party IaaS providers. Its modular architecture allows extending the software with additional features such as the capability of reserving virtual machine instances by using Haizea [39] as scheduler.

Solutions that rely on the previous virtual machine managers and provide added value are OpenPEX [40] and InterGrid [41]. OpenPEX is Web-based system that allows the reservation of virtual machine instances and is designed to support different back ends (at the moment only the support for Xen is implemented). InterGrid provides added value on top of OpenNebula and Amazon EC2 by allowing the reservation of virtual machine instances and managing multi-administrative domain clouds. PaaS solutions can provide an additional layer and deliver a high-level service for private clouds. Among the options available for private deployment of clouds we can consider DataSynapse, Zimory Pools, Elastra, and Aneka. DataSynapse is a global provider of application virtualization software. By relying on the VMware virtualization technology,

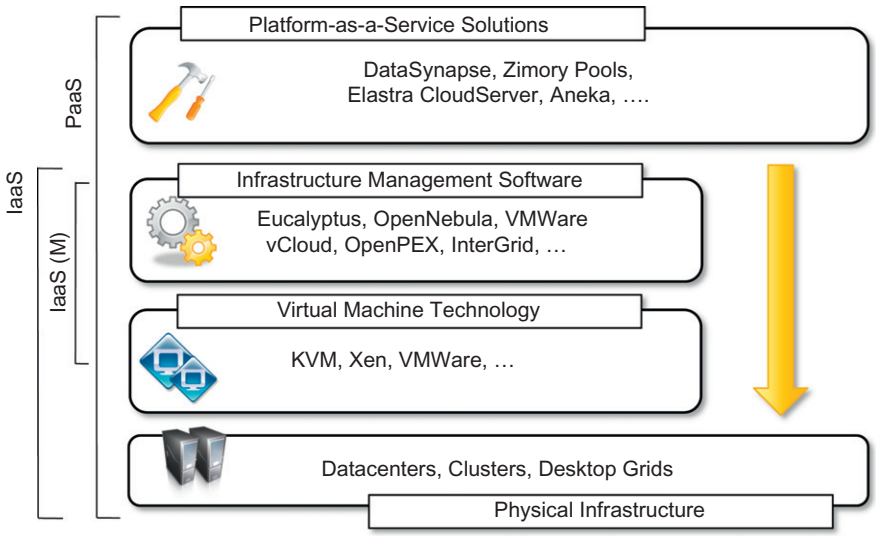

FIGURE 4.4

Private clouds hardware and software stack.

DataSynapse provides a flexible environment for building private clouds on top of datacenters. Elastra Cloud Server is a platform for easily configuring and deploying distributed application infrastructures on clouds. Zimory provides a software infrastructure layer that automates the use of resource pools based on Xen, KVM, and VMware virtualization technologies. It allows creating an internal cloud composed of sparse private and public resources and provides facilities for migrating applications within the existing infrastructure. Aneka is a software development platform that can be used to deploy a cloud infrastructure on top of heterogeneous hardware: datacenters, clusters, and desktop grids. It provides a pluggable service-oriented architecture that's mainly devoted to supporting the execution of distributed applications with different programming models: bag of tasks, MapReduce, and others.

Private clouds can provide in-house solutions for cloud computing, but if compared to public clouds they exhibit more limited capability to scale elastically on demand.

4.3.3 Hybrid clouds

Public clouds are large software and hardware infrastructures that have a capability that is huge enough to serve the needs of multiple users, but they suffer from security threats and administrative pitfalls. Although the option of completely relying on a public virtual infrastructure is appealing for companies that did not incur IT capital costs and have just started considering their IT needs (i.e., start-ups), in most cases the private cloud option prevails because of the existing IT infrastructure.

Private clouds are the perfect solution when it is necessary to keep the processing of information within an enterprise's premises or it is necessary to use the existing hardware and software infrastructure. One of the major drawbacks of private deployments is the inability to scale on demand and to efficiently address peak loads. In this case, it is important to leverage capabilities of public clouds as needed. Hence, a hybrid solution could be an interesting opportunity for taking advantage of the best of the private and public worlds. This led to the development and diffusion of hybrid clouds.

Hybrid clouds allow enterprises to exploit existing IT infrastructures, maintain sensitive information within the premises, and naturally grow and shrink by provisioning external resources and releasing them when they're no longer needed. Security concerns are then only limited to the public portion of the cloud that can be used to perform operations with less stringent constraints but that are still part of the system workload. Figure 4.5 provides a general overview of a hybrid cloud: It is a heterogeneous distributed system resulting from a private cloud that integrates additional services or resources from one or more public clouds. For this reason they are also called *heterogeneous clouds*. As depicted in the diagram, dynamic provisioning is a fundamental component in this scenario. Hybrid clouds address scalability issues by leveraging external resources for exceeding

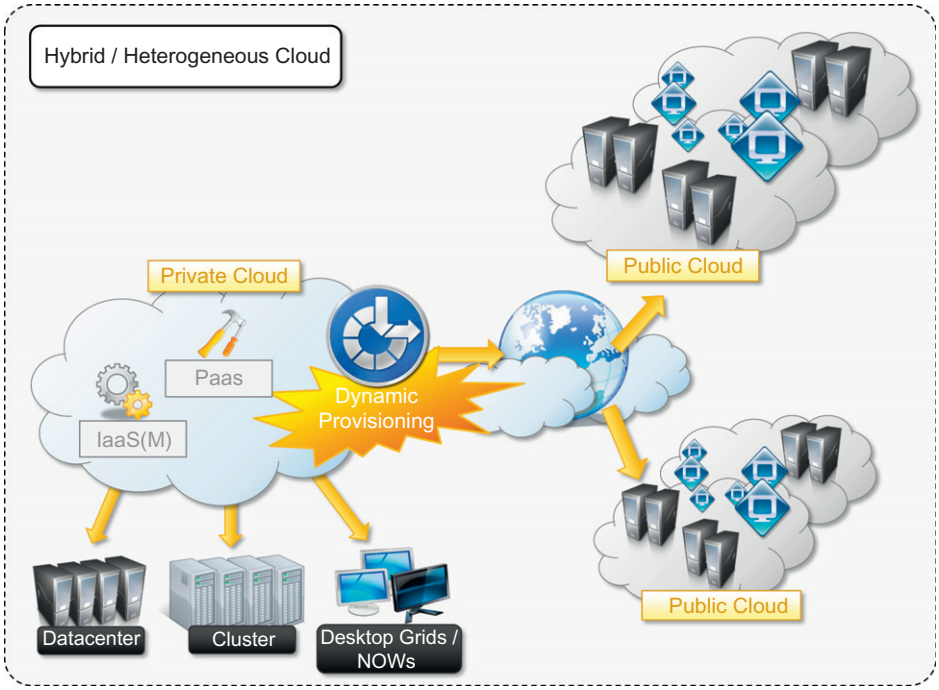

FIGURE 4.5

Hybrid/heterogeneous cloud overview.

capacity demand. These resources or services are temporarily leased for the time required and then released. This practice is also known as *cloudbursting*.[6]

Whereas the concept of hybrid cloud is general, it mostly applies to IT infrastructure rather than software services. Service-oriented computing already introduces the concept of integration of paid software services with existing application deployed in the private premises. In an IaaS scenario, *dynamic provisioning* refers to the ability to acquire on demand virtual machines in order to increase the capability of the resulting distributed system and then release them. Infrastructure management software and PaaS solutions are the building blocks for deploying and managing hybrid clouds. In particular, with respect to private clouds, dynamic provisioning introduces a more complex scheduling algorithm and policies, the goal of which is also to optimize the budget spent to rent public resources.

Infrastructure management software such as OpenNebula already exposes the capability of integrating resources from public clouds such as Amazon EC2. In this case the virtual machine obtained from the public infrastructure is managed as all the other virtual machine instances maintained locally. What is missing is then an advanced scheduling engine that's able to differentiate these resources and provide smart allocations by taking into account the budget available to extend the existing infrastructure. In the case of OpenNebula, advanced schedulers such as Haizea can be integrated to provide cost-based scheduling. A different approach is taken by InterGrid. This is essentially a distributed scheduling engine that manages the allocation of virtual machines in a collection of peer networks. Such networks can be represented by a local cluster, a gateway to a public cloud, or a combination of the two. Once a request is submitted to one of the InterGrid gateways, it is served by possibly allocating virtual instances in all the peered networks, and the allocation of requests is performed by taking into account the user budget and the peering arrangements between networks.

Dynamic provisioning is most commonly implemented in PaaS solutions that support hybrid clouds. As previously discussed, one of the fundamental components of PaaS middleware is the mapping of distributed applications onto the cloud infrastructure. In this scenario, the role of dynamic provisioning becomes fundamental to ensuring the execution of applications under the QoS agreed on with the user. For example, Aneka provides a provisioning service that leverages different IaaS providers for scaling the existing cloud infrastructure [42]. The provisioning service cooperates with the scheduler, which is in charge of guaranteeing a specific QoS for applications. In particular, each user application has a budget attached, and the scheduler uses that budget to optimize the execution of the application by renting virtual nodes if needed. Other PaaS implementations support the deployment of hybrid clouds and provide dynamic provisioning capabilities. Among those discussed for the implementation and management of private clouds we can cite Elastra CloudServer and Zimory Pools.

[6]According to the Cloud Computing Wiki, the term *cloudburst* has a double meaning; it also refers to the "failure of a cloud computing environment due to the inability to handle a spike in demand" (http://sites.google.com/site/Cloudcomputingwiki/Home/Cloud-computing-vocabulary). In this book, we always refer to the dynamic provisioning of resources from public clouds when mentioning this term.

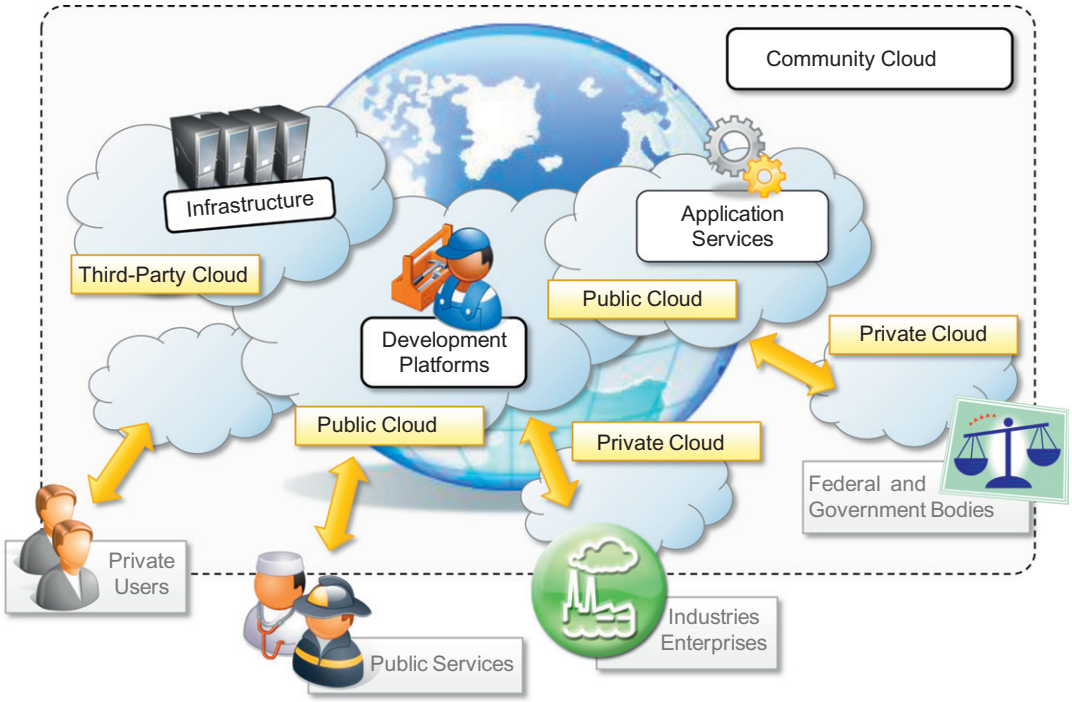

FIGURE 4.6

A community cloud.

4.3.4 Community clouds

Community clouds are distributed systems created by integrating the services of different clouds to address the specific needs of an industry, a community, or a business sector. The National Institute of Standards and Technologies (NIST) [43] characterizes community clouds as follows:

> *The infrastructure is shared by several organizations and supports a specific community that has shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be managed by the organizations or a third party and may exist on premise or off premise.*

Figure 4.6 provides a general view of the usage scenario of community clouds, together with reference architecture. The users of a specific community cloud fall into a well-identified community, sharing the same concerns or needs; they can be government bodies, industries, or even simple users, but all of them focus on the same issues for their interaction with the cloud. This is a different scenario than public clouds, which serve a multitude of users with different needs. Community clouds are also different from private clouds, where the services are generally delivered within the institution that owns the cloud.

From an architectural point of view, a community cloud is most likely implemented over multiple administrative domains. This means that different organizations such as government bodies,

private enterprises, research organizations, and even public virtual infrastructure providers contribute with their resources to build the cloud infrastructure.

Candidate sectors for community clouds are as follows:

- *Media industry*. In the media industry, companies are looking for low-cost, agile, and simple solutions to improve the efficiency of content production. Most media productions involve an extended ecosystem of partners. In particular, the creation of digital content is the outcome of a collaborative process that includes movement of large data, massive compute-intensive rendering tasks, and complex workflow executions. Community clouds can provide a shared environment where services can facilitate business-to-business collaboration and offer the horsepower in terms of aggregate bandwidth, CPU, and storage required to efficiently support media production.
- *Healthcare industry*. In the healthcare industry, there are different scenarios in which community clouds could be of use. In particular, community clouds can provide a global platform on which to share information and knowledge without revealing sensitive data maintained within the private infrastructure. The naturally hybrid deployment model of community clouds can easily support the storing of patient-related data in a private cloud while using the shared infrastructure for noncritical services and automating processes within hospitals.
- *Energy and other core industries*. In these sectors, community clouds can bundle the comprehensive set of solutions that together vertically address management, deployment, and orchestration of services and operations. Since these industries involve different providers, vendors, and organizations, a community cloud can provide the right type of infrastructure to create an open and fair market.
- *Public sector*. Legal and political restrictions in the public sector can limit the adoption of public cloud offerings. Moreover, governmental processes involve several institutions and agencies and are aimed at providing strategic solutions at local, national, and international administrative levels. They involve business-to-administration, citizen-to-administration, and possibly business-to-business processes. Some examples include invoice approval, infrastructure planning, and public hearings. A community cloud can constitute the optimal venue to provide a distributed environment in which to create a communication platform for performing such operations.
- *Scientific research*. Science clouds are an interesting example of community clouds. In this case, the common interest driving different organizations sharing a large distributed infrastructure is scientific computing.

The term *community cloud* can also identify a more specific type of cloud that arises from concern over the controls of vendors in cloud computing and that aspire to combine the principles of *digital ecosystems*[7] [44] with the case study of cloud computing. A community cloud is formed by harnessing the underutilized resources of user machines [45] and providing an infrastructure in

[7]*Digital ecosystems* are distributed, adaptive, and open sociotechnical systems with properties of self-organization, scalability, and sustainability inspired by natural ecosystems. The primary aim of digital ecosystems is to sustain the regional development of small and medium-sized enterprises (SMEs).

which each can be at the same time a consumer, a producer, or a coordinator of the services offered by the cloud. The benefits of these community clouds are the following:

- *Openness*. By removing the dependency on cloud vendors, community clouds are open systems in which fair competition between different solutions can happen.
- *Community*. Being based on a collective that provides resources and services, the infrastructure turns out to be more scalable because the system can grow simply by expanding its user base.
- *Graceful failures*. Since there is no single provider or vendor in control of the infrastructure, there is no single point of failure.
- *Convenience and control*. Within a community cloud there is no conflict between convenience and control because the cloud is shared and owned by the community, which makes all the decisions through a collective democratic process.
- *Environmental sustainability*. The community cloud is supposed to have a smaller carbon footprint because it harnesses underutilized resources. Moreover, these clouds tend to be more organic by growing and shrinking in a symbiotic relationship to support the demand of the community, which in turn sustains it.

This is an alternative vision of a community cloud, focusing more on the social aspect of the clouds that are formed as an aggregation of resources of community members. The idea of a heterogeneous infrastructure built to serve the needs of a community of people is also reflected in the previous definition, but in that case the attention is focused on the commonality of interests that aggregates the users of the cloud into a community. In both cases, the concept of community is fundamental.

4.4 Economics of the cloud

The main drivers of cloud computing are economy of scale and simplicity of software delivery and its operation. In fact, the biggest benefit of this phenomenon is financial: the *pay-as-you-go* model offered by cloud providers. In particular, cloud computing allows:

- Reducing the capital costs associated to the IT infrastructure
- Eliminating the depreciation or lifetime costs associated with IT capital assets
- Replacing software licensing with subscriptions
- Cutting the maintenance and administrative costs of IT resources

A *capital cost* is the cost occurred in purchasing an asset that is useful in the production of goods or the rendering of services. Capital costs are one-time expenses that are generally paid up front and that will contribute over the long term to generate profit. The IT infrastructure and the software are capital assets because enterprises require them to conduct their business. At present it does not matter whether the principal business of an enterprise is related to IT, because the business will definitely have an IT department that is used to automate many of the activities that are performed within the enterprise: payroll, customer relationship management, enterprise resource planning, tracking and inventory of products, and others. Hence, IT resources constitute a capital cost for any kind of enterprise. It is good practice to try to keep capital costs low because they introduce

expenses that will generate profit over time; more than that, since they are associated with material things they are subject to *depreciation* over time, which in the end reduces the profit of the enterprise because such costs are directly subtracted from the enterprise revenues. In the case of IT capital costs, the depreciation costs are represented by the loss of value of the hardware over time and the aging of software products that need to be replaced because new features are required.

Before cloud computing diffused within the enterprise, the budget spent on IT infrastructure and software constituted a significant expense for medium-sized and large enterprises. Many enterprises own a small or medium-sized datacenter that introduces several operational costs in terms of maintenance, electricity, and cooling. Additional operational costs are occurred in maintaining an IT department and an IT support center. Moreover, other costs are triggered by the purchase of potentially expensive software. With cloud computing these costs are significantly reduced or simply disappear according to its penetration. One of the advantages introduced by the cloud computing model is that it shifts the capital costs previously allocated to the purchase of hardware and software into operational costs inducted by renting the infrastructure and paying subscriptions for the use of software. These costs can be better controlled according to the business needs and prosperity of the enterprise. Cloud computing also introduces reductions in administrative and maintenance costs. That is, there is no or limited need for having administrative staff take care of the management of the cloud infrastructure. At the same time, the cost of IT support staff is also reduced. When it comes to depreciation costs, they simply disappear for the enterprise, since in a scenario where all the IT needs are served by the cloud there are no IT capital assets that depreciate over time.

The amount of cost savings that cloud computing can introduce within an enterprise is related to the specific scenario in which cloud services are used and how they contribute to generate a profit for the enterprise. In the case of a small startup, it is possible to completely leverage the cloud for many aspects, such as:

- IT infrastructure
- Software development
- CRM and ERP

In this case it is possible to completely eliminate capital costs because there are no initial IT assets. The situation is completely different in the case of enterprises that already have a considerable amount of IT assets. In this case, cloud computing, especially IaaS-based solutions, can help manage unplanned capital costs that are generated by the needs of the enterprise in the short term. In this case, by leveraging cloud computing, these costs can be turned into operational costs that last as long as there is a need for them. For example, IT infrastructure leasing helps more efficiently manage peak loads without inducing capital expenses. As soon as the increased load does not justify the use of additional resources, these can be released and the costs associated with them disappear. This is the most adopted model of cloud computing because many enterprises already have IT facilities. Another option is to make a slow transition toward cloud-based solutions while the capital IT assets get depreciated and need to be replaced. Between these two cases there is a wide variety of scenarios in which cloud computing could be of help in generating profits for enterprises.

Another important aspect is the elimination of some indirect costs that are generated by IT assets, such as software licensing and support and carbon footprint emissions. With cloud computing, an enterprise uses software applications on a subscription basis, and there is no need for any licensing fee because the software providing the service remains the property of the provider. Leveraging IaaS solutions allows room for datacenter consolidation that in the end could result in a smaller carbon footprint. In some countries such as Australia, the carbon footprint emissions are taxable, so by reducing or completely eliminating such emissions, enterprises can pay less tax.

In terms of the pricing models introduced by cloud computing, we can distinguish three different strategies that are adopted by the providers:

- *Tiered pricing*. In this model, cloud services are offered in several tiers, each of which offers a fixed computing specification and SLA at a specific price per unit of time. This model is used by Amazon for pricing the EC2 service, which makes available different server configurations in terms of computing capacity (CPU type and speed, memory) that have different costs per hour.
- *Per-unit pricing*. This model is more suitable to cases where the principal source of revenue for the cloud provider is determined in terms of units of specific services, such as data transfer and memory allocation. In this scenario customers can configure their systems more efficiently according to the application needs. This model is used, for example, by GoGrid, which makes customers pay according to RAM/hour units for the servers deployed in the GoGrid cloud.
- *Subscription-based pricing*. This is the model used mostly by SaaS providers in which users pay a periodic subscription fee for use of the software or the specific component services that are integrated in their applications.

All of these costs are based on a pay-as-you-go model, which constitutes a more flexible solution for supporting the delivery on demand of IT services. This is what actually makes possible the conversion of IT capital costs into operational costs, since the cost of buying hardware turns into a cost for leasing it and the cost generated by the purchase of software turns into a subscription fee paid for using it.

4.5 Open challenges

Still in its infancy, cloud computing presents many challenges for industry and academia. There is a significant amount of work in academia focused on defining the challenges brought by this phenomenon [46–49]. In this section, we highlight the most important ones: the definition and the formalization of cloud computing, the interoperation between different clouds, the creation of standards, security, scalability, fault tolerance, and organizational aspects.

4.5.1 Cloud definition

As discussed earlier, there have been several attempts made to define cloud computing and to provide a classification of all the services and technologies identified as such. One of the most comprehensive formalizations is noted in the NIST working definition of cloud computing [43]. It **characterizes** cloud computing as on-demand self-service, broad network access, resource-pooling,

rapid elasticity, and measured service; **classifies** services as SaaS, PaaS, and IaaS; and **categorizes** deployment models as public, private, community, and hybrid clouds. The view is in line with our discussion and shared by many IT practitioners and academics.

Despite the general agreement on the NIST definition, there are alternative taxonomies for cloud services. David Linthicum, founder of BlueMountains Labs, provides a more detailed classification,[8] which comprehends 10 different classes and better suits the vision of cloud computing within the enterprise. A different approach has been taken at the University of California, Santa Barbara (UCSB) [50], which departs from the XaaS concept and tries to define an ontology for cloud computing. In their work the concept of a cloud is dissected into five main layers: applications, software environments, software infrastructure, software kernel, and hardware. Each layer addresses the needs of a different class of users within the cloud computing community and most likely builds on the underlying layers. According to the authors, this work constitutes the first effort to provide a more robust interaction model between the different cloud entities on both the functional level and the semantic level.

These characterizations and taxonomies reflect what is meant by cloud computing at the present time, but being in its infancy the phenomenon is constantly evolving, and the same will happen to the attempts to capture the real nature of cloud computing. It is interesting to note that the principal characterization used in this book as a reference for introducing and explaining cloud computing is considered a working definition, which by nature identifies something that continuously changes over time by becoming refined.

4.5.2 Cloud interoperability and standards

Cloud computing is a service-based model for delivering IT infrastructure and applications like utilities such as power, water, and electricity. To fully realize this goal, introducing standards and allowing interoperability between solutions offered by different vendors are objectives of fundamental importance. Vendor lock-in constitutes one of the major strategic barriers against the seamless adoption of cloud computing at all stages. In particular there is major fear on the part of enterprises in which IT constitutes the significant part of their revenues. Vendor lock-in can prevent a customer from switching to another competitor's solution, or when this is possible, it happens at considerable conversion cost and requires significant amounts of time. This can occur either because the customer wants to find a more suitable solution for customer needs or because the vendor is no longer able to provide the required service. The presence of standards that are actually implemented and adopted in the cloud computing community could give room for interoperability and then lessen the risks resulting from vendor lock-in.

The current state of standards and interoperability in cloud computing resembles the early Internet era, when there was no common agreement on the protocols and technologies used and each organization had its own network. Yet the first steps toward a standardization process have been made, and a few organizations, such as the Cloud Computing Interoperability Forum (CCIF),[9]

[8]David Linthicum, Cloud Computing Ontology Framework; http://Cloudcomputing.sys-con.com/node/811519.
[9]www.Cloudforum.org.

the Open Cloud Consortium,[10] and the DMTF Cloud Standards Incubator,[11] are leading the path. Another interesting initiative is the Open Cloud Manifesto,[12] which embodies the point of view of various stakeholders on the benefits of open standards in the field.

The standardization efforts are mostly concerned with the lower level of the cloud computing architecture, which is the most popular and developed. In particular, in the IaaS market, the use of a proprietary virtual machine format constitutes the major reasons for the vendor lock-in, and efforts to provide virtual machine image compatibility between IaaS vendors can possibly improve the level of interoperability among them. The Open Virtualization Format (OVF) [51] is an attempt to provide a common format for storing the information and metadata describing a virtual machine image. Even though the OVF provides a full specification for packaging and distributing virtual machine images in completely platform-independent fashion, it is supported by few vendors that use it to import static virtual machine images. The challenge is providing standards for supporting the migration of running instances, thus allowing the real ability of switching from one infrastructure vendor to another in a completely transparent manner.

Another direction in which standards try to move is devising a general reference architecture for cloud computing systems and providing a standard interface through which one can interact with them. At the moment the compatibility between different solutions is quite restricted, and the lack of a common set of APIs make the interaction with cloud-based solutions vendor specific. In the IaaS market, Amazon Web Services plays a leading role, and other IaaS solutions, mostly open source, provide AWS-compatible APIs, thus constituting themselves as valid alternatives. Even in this case, there is no consistent trend in devising some common APIs for interfacing with IaaS (and, in general, XaaS), and this constitutes one of the areas in which a considerable improvement can be made in the future.

4.5.3 Scalability and fault tolerance

The ability to scale on demand constitutes one of the most attractive features of cloud computing. Clouds allow scaling beyond the limits of the existing in-house IT resources, whether they are infrastructure (compute and storage) or applications services. To implement such a capability, the cloud middleware has to be designed with the principle of scalability along different dimensions in mind—for example, performance, size, and load. The cloud middleware manages a huge number of resource and users, which rely on the cloud to obtain the horsepower that they cannot obtain within the premises without bearing considerable administrative and maintenance costs. These costs are a reality for whomever develops, manages, and maintains the cloud middleware and offers the service to customers. In this scenario, the ability to tolerate failure becomes fundamental, sometimes even more important than providing an extremely efficient and optimized system. Hence, the challenge in this case is designing highly scalable and fault-tolerant systems that are easy to manage and at the same time provide competitive performance.

[10] www.opencloudconsortium.org.
[11] www.dmtf.org/about/cloud-incubator.
[12] www.opencloudmanifesto.org.

4.5.4 Security, trust, and privacy

Security, trust, and privacy issues are major obstacles for massive adoption of cloud computing. The traditional cryptographic technologies are used to prevent data tampering and access to sensitive information. The massive use of virtualization technologies exposes the existing system to new threats, which previously were not considered applicable. For example, it might be possible that applications hosted in the cloud can process sensitive information; such information can be stored within a cloud storage facility using the most advanced technology in cryptography to protect data and then be considered safe from any attempt to access it without the required permissions. Although these data are processed in memory, they must necessarily be decrypted by the legitimate application, but since the application is hosted in a managed virtual environment it becomes accessible to the virtual machine manager that by program is designed to access the memory pages of such an application. In this case, what is experienced is a lack of control over the environment in which the application is executed, which is made possible by leveraging the cloud. It then happens that a new way of using existing technologies creates new opportunities for additional threats to the security of applications. The lack of control over their own data and processes also poses severe problems for the trust we give to the cloud service provider and the level of privacy we want to have for our data.

On one side we need to decide whether to trust the provider itself; on the other side, specific regulations can simply prevail over the agreement the provider is willing to establish with us concerning the privacy of the information managed on our behalf. Moreover, cloud services delivered to the end user can be the result of a complex stack of services that are obtained by third parties via the primary cloud service provider. In this case there is a chain of responsibilities in terms of service delivery that can introduce more vulnerability for the secure management of data, the enforcement of privacy rules, and the trust given to the service provider. In particular, when a violation of privacy or illegal access to sensitive information is detected, it could become difficult to identify who is liable for such violations. The challenges in this area are, then, mostly concerned with devising secure and trustable systems from different perspectives: technical, social, and legal.

4.5.5 Organizational aspects

Cloud computing introduces a significant change in the way IT services are consumed and managed. More precisely, storage, compute power, network infrastructure, and applications are delivered as metered services over the Internet. This introduces a billing model that is new within typical enterprise IT departments, which requires a certain level of cultural and organizational process maturity. In particular, a wide acceptance of cloud computing will require a significant change to business processes and organizational boundaries. Some interesting questions arise in considering the role of the IT department in this new scenario. In particular, the following questions have to be considered:

- What is the new role of the IT department in an enterprise that completely or significantly relies on the cloud?
- How will the compliance department perform its activity when there is a considerable lack of control over application workflows?

- What are the implications (political, legal, etc.) for organizations that lose control over some aspects of their services?
- What will be the perception of the end users of such services?

From an organizational point of view, the lack of control over the management of data and processes poses not only security threats but also new problems that previously did not exist. Traditionally, when there was a problem with computer systems, organizations developed strategies and solutions to cope with them, often by relying on local expertise and knowledge. One of the major advantages of moving IT infrastructure and services to the cloud is to reduce or completely remove the costs related to maintenance and support. As a result, users of such infrastructure and services lose a reference to deal with for IT troubleshooting. At the same time, the existing IT staff is required to have a different kind of competency and, in general, fewer skills, thus reducing their value. These are the challenges from an organizational point of view that must be faced and that will significantly change the relationships within the enterprise itself among the various groups of people working together.

SUMMARY

In this chapter we discussed the fundamental characteristics of cloud computing and introduced reference architecture for classifying and organizing cloud services. To best sum up the content of this chapter, we can recall the NIST working definition of cloud computing, which outlines the fundamental aspects of this phenomenon as follows:

- *Five essential characteristics*. In-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service.
- *Three service models*. Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS).
- *Four deployment models*. Public clouds, private clouds, community clouds, and hybrid clouds.

The major driving force for rapid adoption of cloud computing are the economics and the simplicity of software delivery and operation. Cloud computing presents considerable opportunity to increase the profits of enterprises by reducing capital costs of IT assets and transforming them into operational costs. For these reasons we have also discussed the economic and cost models introduced with cloud computing.

Although cloud computing has been rapidly adopted in industry, there are several open research challenges in areas such as management of cloud computing systems, their security, and social and organizational issues. There is significant room for advancement in software infrastructure and models supporting cloud computing.

Review questions

1. What does the acronym *XaaS* stand for?
2. What are the fundamental components introduced in the cloud reference model?

3. What does Infrastructure-as-a-Service refer to?
4. Which are the basic components of an IaaS-based solution for cloud computing?
5. Provide some examples of IaaS implementations.
6. What are the main characteristics of a Platform-as-a-Service solution?
7. Describe the different categories of options available in a PaaS market.
8. What does the acronym SaaS mean? How does it relate to cloud computing?
9. Give the name of some popular Software-as-a-Service solutions.
10. Classify the various types of clouds.
11. Give an example of the public cloud.
12. Which is the most common scenario for a private cloud?
13. What kinds of needs are addressed by heterogeneous clouds?
14. Describe the fundamental features of the economic and business model behind cloud computing.
15. How does cloud computing help to reduce the time to market for applications and to cut down capital expenses?
16. List some of the challenges in cloud computing.

PART

Cloud Application Programming and the Aneka Platform

2

This page intentionally left blank

CHAPTER

Aneka
Cloud Application Platform

5

Aneka is Manjrasoft Pty. Ltd.'s solution for developing, deploying, and managing cloud applications. Aneka consists of a scalable cloud middleware that can be deployed on top of heterogeneous computing resources. It offers an extensible collection of services coordinating the execution of applications, helping administrators monitor the status of the cloud, and providing integration with existing cloud technologies. One of Aneka's key advantages is its extensible set of application programming interfaces (APIs) associated with different types of programming models—such as Task, Thread, and MapReduce—used for developing distributed applications, integrating new capabilities into the cloud, and supporting different types of cloud deployment models: public, private, and hybrid (see Figure 5.1). These features differentiate Aneka from infrastructure management software and characterize it as a platform for developing, deploying, and managing execution of applications on various types of clouds.

This chapter provides a complete overview of the framework by first describing the architecture of the system. It introduces Aneka's components and the fundamental services that make up the Aneka Cloud and discusses some common deployment scenarios.

5.1 Framework overview

Aneka is a software platform for developing cloud computing applications. It allows harnessing of disparate computing resources and managing them into a unique virtual domain—the Aneka Cloud—in which applications are executed. According to the Cloud Computing Reference Model presented in Chapter 1, Aneka is a *pure PaaS* solution for cloud computing. Aneka is a cloud middleware product that can be deployed on a heterogeneous set of resources: a network of computers, a multicore server, datacenters, virtual cloud infrastructures, or a mixture of these. The framework provides both middleware for managing and scaling distributed applications and an extensible set of APIs for developing them.

Figure 5.2 provides a complete overview of the components of the Aneka framework. The core infrastructure of the system provides a uniform layer that allows the framework to be deployed over different platforms and operating systems. The physical and virtual resources representing the bare metal of the cloud are managed by the Aneka container, which is installed on each node and constitutes the basic building block of the middleware. A collection of interconnected containers constitute the Aneka Cloud: a single domain in which services are made available to users and

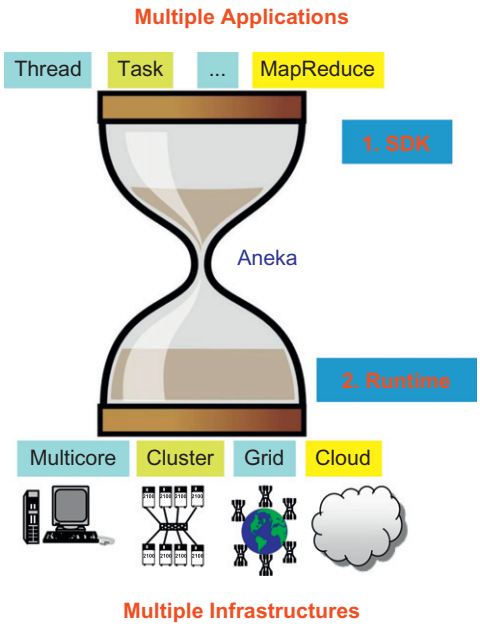

FIGURE 5.1

Aneka's capabilities at a glance.

developers. The container features three different classes of services: *Fabric Services*, *Foundation Services*, and *Execution Services*. These take care of infrastructure management, supporting services for the Aneka Cloud, and application management and execution, respectively. These services are made available to developers and administrators by means of the application management and development layer, which includes interfaces and APIs for developing cloud applications and the management tools and interfaces for controlling Aneka Clouds.

Aneka implements a service-oriented architecture (SOA), and services are the fundamental components of an Aneka Cloud. Services operate at container level and, except for the platform abstraction layer, they provide developers, users, and administrators with all features offered by the framework. Services also constitute the extension and customization point of Aneka Clouds: The infrastructure allows for the integration of new services or replacement of the existing ones with a different implementation. The framework includes the basic services for infrastructure and node management, application execution, accounting, and system monitoring; existing services can be extended and new features can be added to the cloud by dynamically plugging new ones into the container. Such extensible and flexible infrastructure enables Aneka Clouds to support different programming and execution models for applications. A programming model represents a collection of abstractions that developers can use to express distributed applications; the runtime support for a programming model is constituted by a collection of execution and foundation services interacting together to carry out application execution. Thus, the implementation of a new model requires the development of the specific programming abstractions used by application developers and the

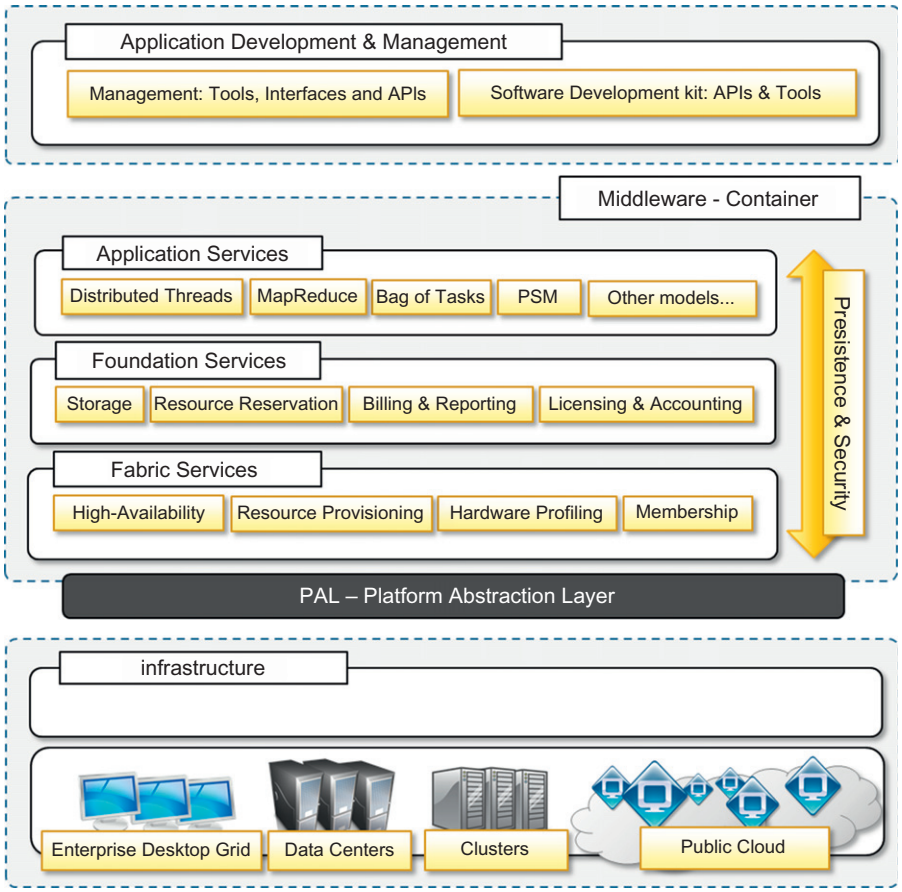

FIGURE 5.2

Aneka framework overview.

services, providing runtime support for them. Programming models are just one aspect of application management and execution. Within an Aneka Cloud environment, there are different aspects involved in providing a scalable and elastic infrastructure and distributed runtime for applications. These services involve:

- *Elasticity and scaling*. By means of the dynamic provisioning service, Aneka supports dynamically upsizing and downsizing of the infrastructure available for applications.
- *Runtime management*. The runtime machinery is responsible for keeping the infrastructure up and running and serves as a hosting environment for services. It is primarily represented by the container and a collection of services that manage service membership and lookup, infrastructure maintenance, and profiling.
- *Resource management*. Aneka is an elastic infrastructure in which resources are added and removed dynamically according to application needs and user requirements. To provide

QoS-based execution, the system not only allows dynamic provisioning but also provides capabilities for reserving nodes for exclusive use by specific applications.

- *Application management.* A specific subset of services is devoted to managing applications. These services include scheduling, execution, monitoring, and storage management.
- *User management.* Aneka is a multitenant distributed environment in which multiple applications, potentially belonging to different users, are executed. The framework provides an extensible user system via which it is possible to define users, groups, and permissions. The services devoted to user management build up the security infrastructure of the system and constitute a fundamental element for accounting management.
- *QoS/SLA management and billing.* Within a cloud environment, application execution is metered and billed. Aneka provides a collection of services that coordinate together to take into account the usage of resources by each application and to bill the owning user accordingly.

All these services are available to specific interfaces and APIs on top of which the software development kit (SDK) and management kit are built. The SDK mainly relates to application development and modeling; it provides developers with APIs to develop applications with the existing programming models and an object model for creating new models. The management kit is mostly focused on interacting with the runtime services for managing the infrastructure, users, and applications. The management kit gives a complete view of Aneka Clouds and allows monitoring Aneka's status, whereas the SDK is more focused on the single application and provides means to control its execution from a single user. Both components are meant to provide an easy-to-use interface via which to interact and manage containers that are the core component of the Aneka framework.

5.2 Anatomy of the Aneka container

The Aneka container constitutes the building blocks of Aneka Clouds and represents the runtime machinery available to services and applications. The container, the unit of deployment in Aneka Clouds, is a lightweight software layer designed to host services and interact with the underlying operating system and hardware. The main role of the container is to provide a lightweight environment in which to deploy services and some basic capabilities such as communication channels through which it interacts with other nodes in the Aneka Cloud. Almost all operations performed within Aneka are carried out by the services managed by the container. The services installed in the Aneka container can be classified into three major categories:

- Fabric Services
- Foundation Services
- Application Services

The services stack resides on top of the *Platform Abstraction Layer (PAL)*, representing the interface to the underlying operating system and hardware. It provides a uniform view of the software and hardware environment in which the container is running. Persistence and security traverse all the services stack to provide a secure and reliable infrastructure. In the following sections we discuss the components of these layers in more detail.

5.2.1 From the ground up: the platform abstraction layer

The core infrastructure of the system is based on the .NET technology and allows the Aneka container to be portable over different platforms and operating systems. Any platform featuring an ECMA-334 [52] and ECMA-335 [53] compatible environment can host and run an instance of the Aneka container.

The *Common Language Infrastructure (CLI)*, which is the specification introduced in the ECMA-334 standard, defines a common runtime environment and application model for executing programs but does not provide any interface to access the hardware or to collect performance data from the hosting operating system. Moreover, each operating system has a different file system organization and stores that information differently. The *Platform Abstraction Layer (PAL)* addresses this heterogeneity and provides the container with a uniform interface for accessing the relevant hardware and operating system information, thus allowing the rest of the container to run unmodified on any supported platform.

The PAL is responsible for detecting the supported hosting environment and providing the corresponding implementation to interact with it to support the activity of the container. The PAL provides the following features:

- Uniform and platform-independent implementation interface for accessing the hosting platform
- Uniform access to extended and additional properties of the hosting platform
- Uniform and platform-independent access to remote nodes
- Uniform and platform-independent management interfaces

The PAL is a small layer of software that comprises a detection engine, which automatically configures the container at boot time, with the platform-specific component to access the above information and an implementation of the abstraction layer for the Windows, Linux, and Mac OS X operating systems.

The collectible data that are exposed by the PAL are the following:

- Number of cores, frequency, and CPU usage
- Memory size and usage
- Aggregate available disk space
- Network addresses and devices attached to the node

Moreover, additional custom information can be retrieved by querying the properties of the hardware. The PAL interface provides means for custom implementations to pull additional information by using name-value pairs that can host any kind of information about the hosting platform. For example, these properties can contain additional information about the processor, such as the model and family, or additional data about the process running the container.

5.2.2 Fabric services

Fabric Services define the lowest level of the software stack representing the Aneka Container. They provide access to the resource-provisioning subsystem and to the monitoring facilities implemented in Aneka. Resource-provisioning services are in charge of dynamically providing new nodes on demand by relying on virtualization technologies, while monitoring services allow for hardware profiling and implement a basic monitoring infrastructure that can be used by all the services installed in the container.

5.2.2.1 Profiling and monitoring

Profiling and monitoring services are mostly exposed through the *Heartbeat*, *Monitoring*, and *Reporting Services*. The first makes available the information that is collected through the PAL; the other two implement a generic infrastructure for monitoring the activity of any service in the Aneka Cloud.

The Heartbeat Service periodically collects the dynamic performance information about the node and publishes this information to the membership service in the Aneka Cloud. These data are collected by the index node of the Cloud, which makes them available for services such as reservations and scheduling in order to optimize the use of a heterogeneous infrastructure. As already discussed, basic information about memory, disk space, CPU, and operating system is collected. Moreover, additional data are pulled into the "alive" message, such as information about the installed software in the system and any other useful information. More precisely, the infrastructure has been designed to carry over any type of data that can be expressed by means of text-valued properties. As previously noted, the information published by the Heartbeat Service is mostly concerned with the properties of the node. A specific component, called *Node Resolver*, is in charge of collecting these data and making them available to the Heartbeat Service. Aneka provides different implementations for such component in order to cover a wide variety of hosting environments. A variety of operating systems are supported with different implementations of the PAL, and different node resolvers allow Aneka to capture other types of data that do not strictly depend on the hosting operating system. For example, the retrieval of the public IP of the node is different in the case of physical machines or virtual instances hosted in the infrastructure of an IaaS provider such as EC2 or GoGrid. In virtual deployment, a different node resolver is used so that all other components of the system can work transparently.

The set of built-in services for monitoring and profiling is completed by a generic monitoring infrastructure, which allows any custom service to report its activity. This infrastructure is composed of the Reporting and Monitoring Services. The *Reporting Service* manages the store for monitored data and makes them accessible to other services or external applications for analysis purposes. On each node, an instance of the *Monitoring Service* acts as a gateway to the *Reporting Service* and forwards to it all the monitored data that has been collected on the node. Any service that wants to publish monitoring data can leverage the local monitoring service without knowing the details of the entire infrastructure. Currently several built-in services provide information through this channel:

- The *Membership Catalogue* tracks the performance information of nodes.
- The *Execution Service* monitors several time intervals for the execution of jobs.
- The *Scheduling Service* tracks the state transitions of jobs.
- The *Storage Service* monitors and makes available information about data transfer, such as upload and download times, filenames, and sizes.
- The *Resource Provisioning Service* tracks the provisioning and lifetime information of virtual nodes.

All this information can be stored on a relational database management system (RDBMS) or a flat file and can be further analyzed by specific applications. For example, the Management Console provides a view on such data for administrative purposes.

5.2.2.2 Resource management

Resource management is another fundamental feature of Aneka Clouds. It comprises several tasks: resource membership, resource reservation, and resource provisioning. Aneka provides a collection of services that are in charge of managing resources. These are the *Index Service* (or *Membership Catalogue*), *Reservation Service*, and *Resource Provisioning Service*.

The *Membership Catalogue* is Aneka's fundamental component for resource management; it keeps track of the basic node information for all the nodes that are connected or disconnected. The Membership Catalogue implements the basic services of a directory service, allowing the search for services using attributes such as names and nodes. During container startup, each instance publishes its information to the Membership Catalogue and updates it constantly during its lifetime. Services and external applications can query the membership catalogue to discover the available services and interact with them. To speed up and enhance the performance of queries, the membership catalogue is organized as a distributed database: All the queries that pertain to information maintained locally are resolved locally; otherwise the query is forwarded to the main index node, which has a global knowledge of the entire Cloud. The Membership Catalogue is also the collector of the dynamic performance data of each node, which are then sent to the local monitoring service to be persisted in the long term.

Indexing and categorizing resources are fundamental to resource management. On top of the basic indexing service, provisioning completes the set of features that are available for resource management within Aneka. Deployment of container instances and their configuration are performed by the infrastructure management layer and are not part of the Fabric Services.

Dynamic resource provisioning allows the integration and management of virtual resources leased from IaaS providers into the Aneka Cloud. This service changes the structure of the Aneka Cloud by allowing it to scale up and down according to different needs: handling node failures, ensuring the quality of service for applications, or maintaining a constant performance and throughput of the Cloud. Aneka defines a very flexible infrastructure for resource provisioning whereby it is possible to change the logic that triggers provisioning, support several back-ends, and change the runtime strategy with which a specific back-end is selected for provisioning. The resource-provisioning infrastructure built into Aneka is mainly concentrated in the *Resource Provisioning Service*, which includes all the operations that are needed for provisioning virtual instances. The implementation of the service is based on the idea of *resource pools*. A resource pool abstracts the interaction with a specific IaaS provider by exposing a common interface so that all the pools can be managed uniformly. A resource pool does not necessarily map to an IaaS provider but can be used to expose as dynamic resources a private cloud managed by a Xen Hypervisor or a collection of physical resources that are only used sporadically. The system uses an open protocol, allowing for the use of metadata to provide additional information for describing resource pools and to customize provisioning requests. This infrastructure simplifies the implementation of additional features and the support of different implementations that can be transparently integrated into the existing system.

Resource provisioning is a feature designed to support QoS requirements-driven execution of applications. Therefore, it mostly serves requests coming from the Reservation Service or the Scheduling Services. Despite this, external applications can directly leverage Aneka's resource-provisioning capabilities by dynamically retrieving a client to the service and interacting with the infrastructure to it. This extends the resource-provisioning scenarios that can be handled by Aneka, which can also be used as a virtual machine manager.

5.2.3 Foundation services

Fabric Services are fundamental services of the Aneka Cloud and define the basic infrastructure management features of the system. *Foundation Services* are related to the logical management of the distributed system built on top of the infrastructure and provide supporting services for the execution of distributed applications. All the supported programming models can integrate with and leverage these services to provide advanced and comprehensive application management. These services cover:

- Storage management for applications
- Accounting, billing, and resource pricing
- Resource reservation

Foundation Services provide a uniform approach to managing distributed applications and allow developers to concentrate only on the logic that distinguishes a specific programming model from the others. Together with the Fabric Services, Foundation Services constitute the core of the Aneka middleware. These services are mostly consumed by the execution services and Management Consoles. External applications can leverage the exposed capabilities for providing advanced application management.

5.2.3.1 Storage management

Data management is an important aspect of any distributed system, even in computing clouds. Applications operate on data, which are mostly persisted and moved in the format of files. Hence, any infrastructure that supports the execution of distributed applications needs to provide facilities for file/data transfer management and persistent storage. Aneka offers two different facilities for storage management: a centralized file storage, which is mostly used for the execution of compute-intensive applications, and a distributed file system, which is more suitable for the execution of data-intensive applications. The requirements for the two types of applications are rather different. Compute-intensive applications mostly require powerful processors and do not have high demands in terms of storage, which in many cases is used to store small files that are easily transferred from one node to another. In this scenario, a centralized storage node, or a pool of storage nodes, can constitute an appropriate solution. In contrast, data-intensive applications are characterized by large data files (gigabytes or terabytes), and the processing power required by tasks does not constitute a performance bottleneck. In this scenario, a distributed file system harnessing the storage space of all the nodes belonging to the cloud might be a better and more scalable solution.

Centralized storage is implemented through and managed by Aneka's *Storage Service*. The service constitutes Aneka's data-staging facilities. It provides distributed applications with the basic file transfer facility and abstracts the use of a specific protocol to end users and other components of the system, which are dynamically configured at runtime according to the facilities installed in the cloud. The option that is currently installed by default is normal File Transfer Protocol (FTP).

To support different protocols, the system introduces the concept of a *file channel* that identifies a pair of components: a file channel controller and a file channel handler. The *file channel controller* constitutes the server component of the channel, where files are stored and made available; the *file channel handler* represents the client component, which is used by user applications or other components of the system to upload, download, or browse files. The storage service uses the

configured file channel factory to first create the server component that will manage the storage and then create the client component on demand. User applications that require support for file transfer are automatically configured with the appropriate file channel handler and transparently upload input files or download output files during application execution. In the same way, worker nodes are configured by the infrastructure to retrieve the required files for the execution of the jobs and to upload their results.

An interesting property of the file channel abstraction is the ability to chain two different channels to move files by using two different protocols. Each file in Aneka contains metadata that helps the infrastructure select the appropriate channel for moving the file. For example, an output file whose final location is an S3 bucket can be moved from the worker node to the Storage Service using the internal FTP protocol and then can be staged out on S3 by the FTP channel controller managed by the service. The Storage Service supports the execution of task-based programming such as the *Task* and the *Thread Model* as well as *Parameter Sweep*-based applications.

Storage support for data-intensive applications is provided by means of a distributed file system. The reference model for the distributed file system is the Google File System [54], which features a highly scalable infrastructure based on commodity hardware. The architecture of the file system is based on a master node, which contains a global map of the file system and keeps track of the status of all the storage nodes, and a pool of chunk servers, which provide distributed storage space in which to store files. Files are logically organized into a directory structure but are persisted on the file system using a flat namespace based on a unique ID. Each file is organized as a collection of *chunks* that are all of the same size. File chunks are assigned unique IDs and stored on different servers, eventually replicated to provide high availability and failure tolerance. The model proposed by the Google File System provides optimized support for a specific class of applications that expose the following characteristics:

- Files are huge by traditional standards (multi-gigabytes).
- Files are modified by appending new data rather than rewriting existing data.
- There are two kinds of major workloads: large streaming reads and small random reads.
- It is more important to have a sustained bandwidth than a low latency.

Moreover, given the huge number of commodity machines that the file system harnesses together, failure (process or hardware failure) is the norm rather than an exception. These characteristics strongly influenced the design of the storage, which provides the best performance for applications specifically designed to operate on data as described. Currently, the only programming model that makes use of the distributed file system is *MapReduce* [55], which has been the primary reason for the Google File System implementation. Aneka provides a simple distributed file system (DFS), which relies on the file system services of the Windows operating system.

5.2.3.2 Accounting, billing, and resource pricing

Accounting Services keep track of the status of applications in the Aneka Cloud. The collected information provides a detailed breakdown of the distributed infrastructure usage and is vital for the proper management of resources.

The information collected for accounting is primarily related to infrastructure usage and application execution. A complete history of application execution and storage as well as other resource

utilization parameters is captured and minded by the Accounting Services. This information constitutes the foundation on which users are charged in Aneka.

Billing is another important feature of accounting. Aneka is a multitenant cloud programming platform in which the execution of applications can involve provisioning additional resources from commercial IaaS providers. Aneka Billing Service provides detailed information about each user's usage of resources, with the associated costs. Each resource can be priced differently according to the set of services that are available on the corresponding Aneka container or the installed software in the node. The accounting model provides an integrated view of budget spent for each application, a summary view of the costs associated to a specific user, and the detailed information about the execution cost of each job.

The accounting capabilities are concentrated within the *Accounting Service* and the *Reporting Service*. The former keeps track of the information that is related to application execution, such as the distribution of jobs among the available resources, the timing of each of job, and the associated cost. The latter makes available the information collected from the monitoring services for accounting purposes: storage utilization and CPU performance. This information is primarily consumed by the Management Console.

5.2.3.3 Resource reservation

Aneka's *Resource Reservation* supports the execution of distributed applications and allows for reserving resources for exclusive use by specific applications. Resource reservation is built out of two different kinds of services: Resource Reservation and the Allocation Service. Resource Reservation keeps track of all the reserved time slots in the Aneka Cloud and provides a unified view of the system. The *Allocation Service* is installed on each node that features execution services and manages the database of information regarding the allocated slots on the local node. Applications that need to complete within a given deadline can make a reservation request for a specific number of nodes in a given timeframe. If it is possible to satisfy the request, the Reservation Service will return a reservation identifier as proof of the resource booking. During application execution, such an identifier is used to select the nodes that have been reserved, and they will be used to execute the application. On each reserved node, the execution services will check with the Allocation Service that each job has valid permissions to occupy the execution timeline by verifying the reservation identifier. Even though this is the general reference model for the reservation infrastructure, Aneka allows for different implementations of the service, which mostly vary in the protocol that is used to reserve resources or the parameters that can be specified while making a reservation request. Different protocol and strategies are integrated in a completely transparent manner, and Aneka provides extensible APIs for supporting advanced services. At the moment, the framework supports three different implementations:

- *Basic Reservation*. Features the basic capability to reserve execution slots on nodes and implements the *alternate offers* protocol, which provides alternative options in case the initial reservation requests cannot be satisfied.
- *Libra Reservation.* Represents a variation of the previous implementation that features the ability to price nodes differently according to their hardware capabilities.
- *Relay Reservation.* Constitutes a very thin implementation that allows a resource broker to reserve nodes in Aneka Clouds and control the logic with which these nodes are reserved. This

implementation is useful in integration scenarios in which Aneka operates in an intercloud environment.

Resource reservation is fundamental to ensuring the quality of service that is negotiated for applications. It allows Aneka to have a predictable environment in which applications can complete within the deadline or not be executed at all. The assumptions made by the reservation service for accepting reservation requests are based on the static allocation of such requests to the existing physical (or virtual) infrastructure available at the time of the requests and by taking into account the current and future load. This solution is sensitive to node failures that could make Aneka unable to fulfill the service-level agreement (SLA) made with users. Specific implementations of the service tend to delay the allocation of nodes to reservation requests as late as possible in order to cope with temporary failures or limited outages, but in the case of serious outages in which the remaining available nodes are not able to cover the demand, this strategy is not enough. In this case, resource provisioning can provide an effective solution: Additional nodes can be provisioned from external resource providers in order to cover the outage and meet the SLA defined for applications. The current implementation of the resource reservation infrastructure leverages the provisioning capabilities of the fabric layer when the current availability in the system is not able to address the reservation requests already confirmed. Such behavior solves the problems of both insufficient resources and temporary failures.

5.2.4 Application services

Application Services manage the execution of applications and constitute a layer that differentiates according to the specific programming model used for developing distributed applications on top of Aneka. The types and the number of services that compose this layer for each of the programming models may vary according to the specific needs or features of the selected model. It is possible to identify two major types of activities that are common across all the supported models: scheduling and execution. Aneka defines a reference model for implementing the runtime support for programming models that abstracts these two activities in corresponding services: the *Scheduling Service* and the *Execution Service*. Moreover, it also defines base implementations that can be extended in order to integrate new models.

5.2.4.1 Scheduling

Scheduling Services are in charge of planning the execution of distributed applications on top of Aneka and governing the allocation of jobs composing an application to nodes. They also constitute the integration point with several other Foundation and Fabric Services, such as the Resource Provisioning Service, the Reservation Service, the Accounting Service, and the Reporting Service. Common tasks that are performed by the scheduling component are the following:

- Job to node mapping
- Rescheduling of failed jobs
- Job status monitoring
- Application status monitoring

Aneka does not provide a centralized scheduling engine, but each programming model features its own scheduling service that needs to work in synergy with the existing services of the middleware. As already mentioned, these services mostly belong to the fabric and the foundation layers of the architecture shown in Figure 5.2. The possibility of having different scheduling engines for different models gives great freedom in implementing scheduling and resource allocation strategies but, at the same time, requires a careful design of use of shared resources. In this scenario, common situations that have to be appropriately managed are the following: multiple jobs sent to the same node at the same time; jobs without reservations sent to reserved nodes; and jobs sent to nodes where the required services are not installed. Aneka's Foundation Services provide sufficient information to avoid these cases, but the runtime infrastructure does not feature specific policies to detect these conditions and provide corrective action. The current design philosophy in Aneka is to keep the scheduling engines completely separate from each other and to leverage existing services when needed. As a result, it is possible to enforce that only one job per programming model is run on each node at any given time, but the execution of applications is not mutually exclusive unless Resource Reservation is used.

5.2.4.2 Execution

Execution Services control the execution of single jobs that compose applications. They are in charge of setting up the runtime environment hosting the execution of jobs. As happens for the scheduling services, each programming model has its own requirements, but it is possible to identify some common operations that apply across all the range of supported models:

- Unpacking the jobs received from the scheduler
- Retrieval of input files required for job execution
- Sandboxed execution of jobs
- Submission of output files at the end of execution
- Execution failure management (i.e., capturing sufficient contextual information useful to identify the nature of the failure)
- Performance monitoring
- Packing jobs and sending them back to the scheduler

Execution Services constitute a more self-contained unit with respect to the corresponding scheduling services. They handle less information and are required to integrate themselves only with the Storage Service and the local Allocation and Monitoring Services. Aneka provides a reference implementation of execution services that has built-in integration with all these services, and currently two of the supported programming models specialize on the reference implementation.

Application Services constitute the runtime support of the programming model in the Aneka Cloud. Currently there are several supported models:

- *Task Model*. This model provides the support for the independent "bag of tasks" applications and many computing tasks. In this model, an application is modeled as a collection of tasks that are independent from each other and whose execution can be sequenced in any order.
- *Thread Model*. This model provides an extension to the classical multithreaded programming to a distributed infrastructure and uses the abstraction of *Thread* to wrap a method that is executed remotely.

- *MapReduce Model*. This is an implementation of MapReduce as proposed by Google on top of Aneka.
- *Parameter Sweep Model*. This model is a specialization of the Task Model for applications that can be described by a template task whose instances are created by generating different combinations of parameters, which identify a specific point into the domain of interest.

Other programming models have been developed for internal use and are at an experimental stage. These are the Dataflow Model [56], the Message-Passing Interface, and the Actor Model [57].

5.3 Building Aneka clouds

Aneka is primarily a platform for developing distributed applications for clouds. As a software platform it requires infrastructure on which to be deployed; this infrastructure needs to be managed. Infrastructure management tools are specifically designed for this task, and building clouds is one of the primary tasks of administrators. Aneka supports various deployment models for public, private, and hybrid clouds.

5.3.1 Infrastructure organization

Figure 5.3 provides an overview of Aneka Clouds from an infrastructure point of view. The scenario is a reference model for all the different deployments Aneka supports. A central role is played by the Administrative Console, which performs all the required management operations. A fundamental element for Aneka Cloud deployment is constituted by *repositories*. A repository provides storage for all the libraries required to lay out and install the basic Aneka platform. These libraries constitute the software image for the node manager and the container programs. Repositories can make libraries available through a variety of communication channels, such as HTTP, FTP, common file sharing, and so on. The Management Console can manage multiple repositories and select the one that best suits the specific deployment. The infrastructure is deployed by harnessing a collection of nodes and installing on them the Aneka node manager, also called the *Aneka daemon*. The daemon constitutes the remote management service used to deploy and control container instances. The collection of resulting containers identifies the Aneka Cloud.

From an infrastructure point of view, the management of physical or virtual nodes is performed uniformly as long as it is possible to have an Internet connection and remote administrative access to the node. A different scenario is constituted by the dynamic provisioning of virtual instances; these are generally created by prepackaged images already containing an installation of Aneka, which only need to be configured to join a specific Aneka Cloud. It is also possible to simply install the container or install the Aneka daemon, and the selection of the proper solution mostly depends on the lifetime of virtual resources.

5.3.2 Logical organization

The logical organization of Aneka Clouds can be very diverse, since it strongly depends on the configuration selected for each of the container instances belonging to the Cloud. The most common

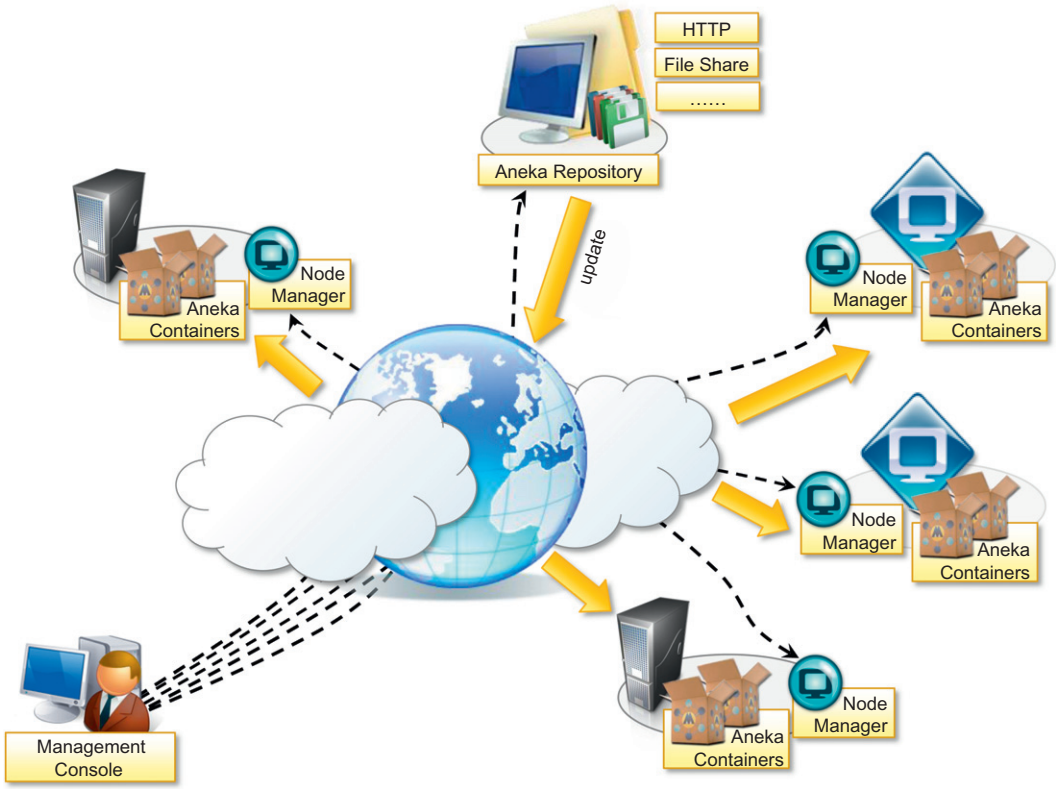

FIGURE 5.3

Aneka cloud infrastructure overview.

scenario is to use a master-worker configuration with separate nodes for storage, as shown in Figure 5.4.

The master node features all the services that are most likely to be present in one single copy and that provide the intelligence of the Aneka Cloud. What specifically characterizes a node as a master node is the presence of the *Index Service* (or Membership Catalogue) configured in master mode; all the other services, except for those that are mandatory, might be present or located in other nodes. A common configuration of the master node is as follows:

- Index Service (master copy)
- Heartbeat Service
- Logging Service
- Reservation Service
- Resource Provisioning Service
- Accounting Service
- Reporting and Monitoring Service
- Scheduling Services for the supported programming models

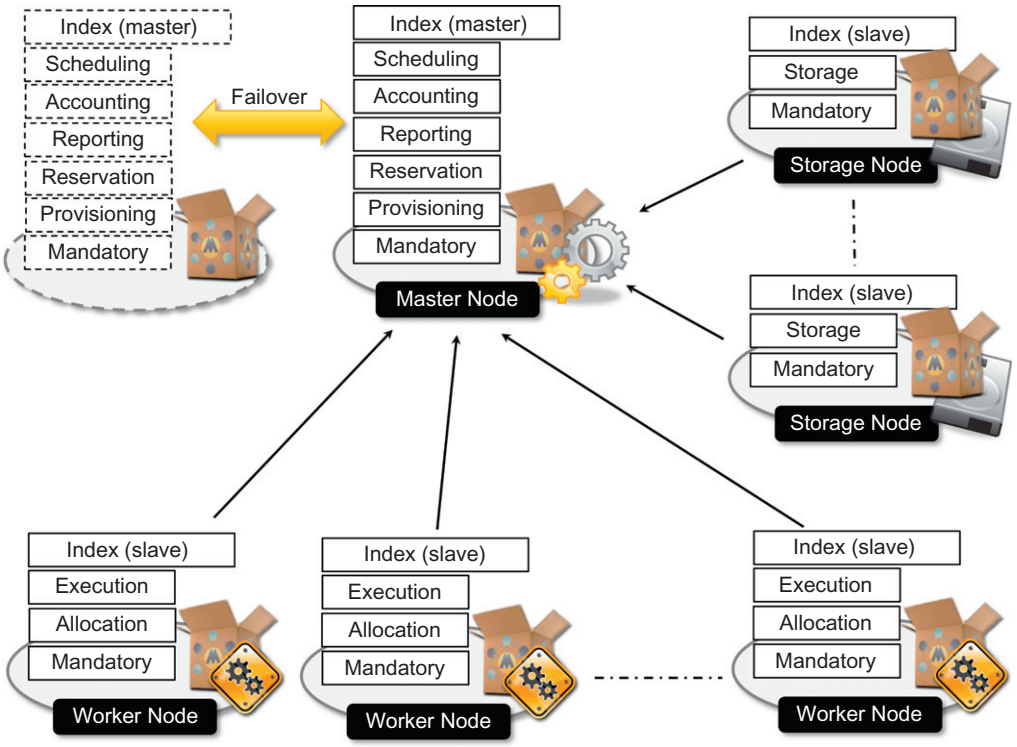

FIGURE 5.4

Logical organization of an Aneka cloud.

The master node also provides connection to an RDBMS facility where the state of several services is maintained. For the same reason, all the scheduling services are maintained in the master node. They share the application store that is normally persisted on the RDBMS in order to provide a fault-tolerant infrastructure. The master configuration can then be replicated in several nodes to provide a highly available infrastructure based on the failover mechanism.

The worker nodes constitute the workforce of the Aneka Cloud and are generally configured for the execution of applications. They feature the mandatory services and the specific execution services of each of the supported programming models in the Cloud. A very common configuration is the following:

- Index Service
- Heartbeat Service
- Logging Service
- Allocation Service
- Monitoring Service
- Execution Services for the supported programming models

A different option is to partition the pool of worker nodes with a different selection of execution services in order to balance the load between programming models and reserve some nodes for a specific class of applications.

Storage nodes are optimized to provide storage support to applications. They feature, among the mandatory and usual services, the presence of the Storage Service. The number of storage nodes strictly depends on the predicted workload and storage consumption of applications. Storage nodes mostly reside on machines that have considerable disk space to accommodate a large quantity of files. The common configuration of a storage node is the following:

- Index Service
- Heartbeat Service
- Logging Service
- Monitoring Service
- Storage Service

In specific cases, when the data transfer requirements are not demanding, there might be only one storage node. In some cases, for very small deployments, there is no need to have a separate storage node, and the Storage Service is installed and hosted on the master node.

All nodes are registered with the master node and transparently refer to any failover partner in the case of a high-availability configuration.

5.3.3 Private cloud deployment mode

A private deployment mode is mostly constituted by local physical resources and infrastructure management software providing access to a local pool of nodes, which might be virtualized. In this scenario Aneka Clouds are created by harnessing a heterogeneous pool of resources such has desktop machines, clusters, or workstations. These resources can be partitioned into different groups, and Aneka can be configured to leverage these resources according to application needs. Moreover, leveraging the Resource Provisioning Service, it is possible to integrate virtual nodes provisioned from a local resource pool managed by systems such as XenServer, Eucalyptus, and OpenStack.

Figure 5.5 shows a common deployment for a private Aneka Cloud. This deployment is acceptable for a scenario in which the workload of the system is predictable and a local virtual machine manager can easily address excess capacity demand. Most of the Aneka nodes are constituted of physical nodes with a long lifetime and a static configuration and generally do not need to be reconfigured often. The different nature of the machines harnessed in a private environment allows for specific policies on resource management and usage that can be accomplished by means of the Reservation Service. For example, desktop machines that are used during the day for office automation can be exploited outside the standard working hours to execute distributed applications. Workstation clusters might have some specific legacy software that is required for supporting the execution of applications and should be preferred for the execution of applications with special requirements.

5.3.4 Public cloud deployment mode

Public Cloud deployment mode features the installation of Aneka master and worker nodes over a completely virtualized infrastructure that is hosted on the infrastructure of one or more resource

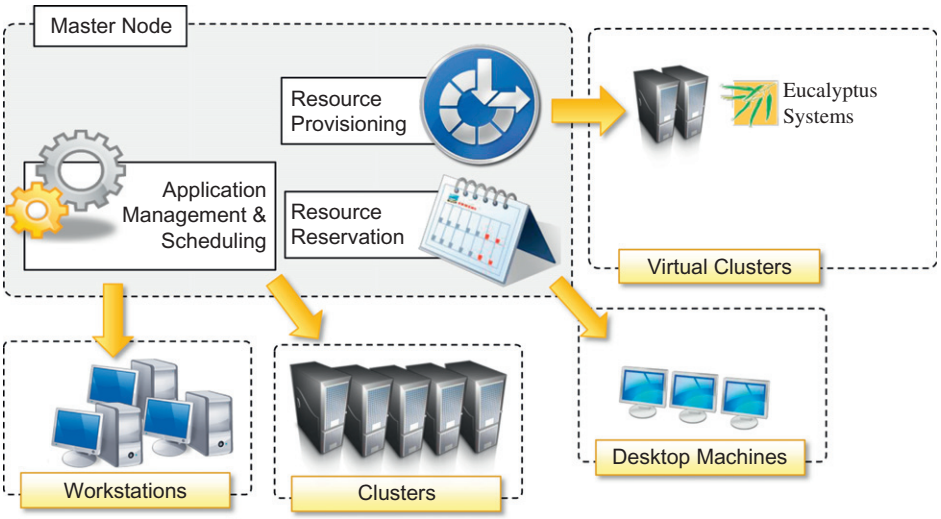

FIGURE 5.5

Private cloud deployment.

providers such as Amazon EC2 or GoGrid. In this case it is possible to have a static deployment where the nodes are provisioned beforehand and used as though they were real machines. This deployment merely replicates a classic Aneka installation on a physical infrastructure without any dynamic provisioning capability. More interesting is the use of the elastic features of IaaS providers and the creation of a Cloud that is completely dynamic. Figure 5.6 provides an overview of this scenario.

The deployment is generally contained within the infrastructure boundaries of a single IaaS provider. The reasons for this are to minimize the data transfer between different providers, which is generally priced at a higher cost, and to have better network performance. In this scenario it is possible to deploy an Aneka Cloud composed of only one node and to completely leverage dynamic provisioning to elastically scale the infrastructure on demand. A fundamental role is played by the Resource Provisioning Service, which can be configured with different images and templates to instantiate. Other important services that have to be included in the master node are the Accounting and Reporting Services. These provide details about resource utilization by users and applications and are fundamental in a multitenant Cloud where users are billed according to their consumption of Cloud capabilities.

Dynamic instances provisioned on demand will mostly be configured as worker nodes, and, in the specific case of Amazon EC2, different images featuring a different hardware setup can be made available to instantiate worker containers. Applications with specific requirements for computing capacity or memory can provide additional information to the scheduler that will trigger the appropriate provisioning request. Application execution is not the only use of dynamic instances; any service requiring elastic scaling can leverage dynamic provisioning. Another example is the Storage Service. In multitenant Clouds, multiple applications can leverage the support for storage; in this

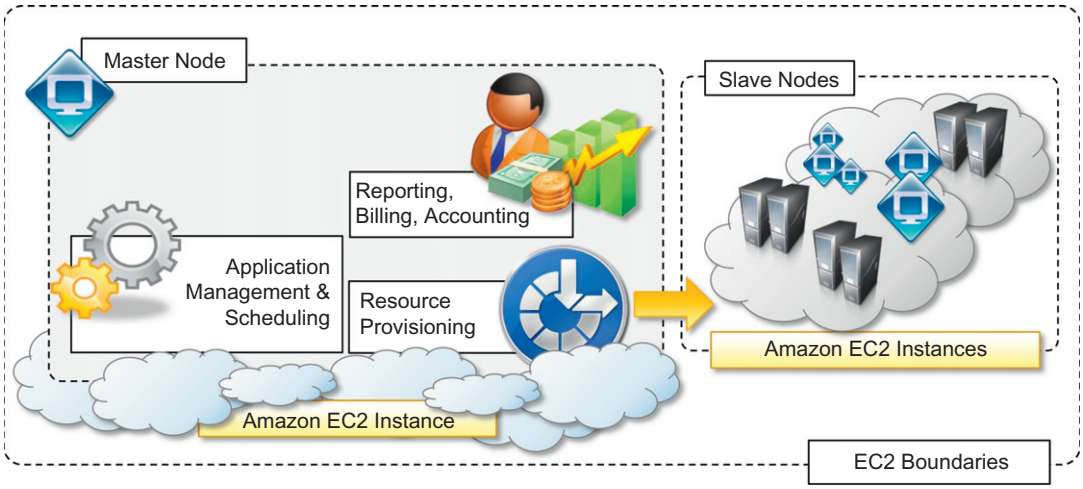

FIGURE 5.6

Public Aneka cloud deployment.

scenario it is then possible to introduce bottlenecks or simply reach the quota limits allocated for storage on the node. Dynamic provisioning can easily solve this issue as it does for increasing the computing capability of an Aneka Cloud.

Deployments using different providers are unlikely to happen because of the data transfer costs among providers, but they might be a possible scenario for federated Aneka Clouds [58]. In this scenario resources can be shared or leased among providers under specific agreements and more convenient prices. In this case the specific policies installed in the Resource Provisioning Service can discriminate among different resource providers, mapping different IaaS providers to provide the best solution to a provisioning request.

5.3.5 Hybrid cloud deployment mode

The hybrid deployment model constitutes the most common deployment of Aneka. In many cases, there is an existing computing infrastructure that can be leveraged to address the computing needs of applications. This infrastructure will constitute the static deployment of Aneka that can be elastically scaled on demand when additional resources are required. An overview of this deployment is presented in Figure 5.7.

This scenario constitutes the most complete deployment for Aneka that is able to leverage all the capabilities of the framework:

- Dynamic Resource Provisioning
- Resource Reservation
- Workload Partitioning
- Accounting, Monitoring, and Reporting

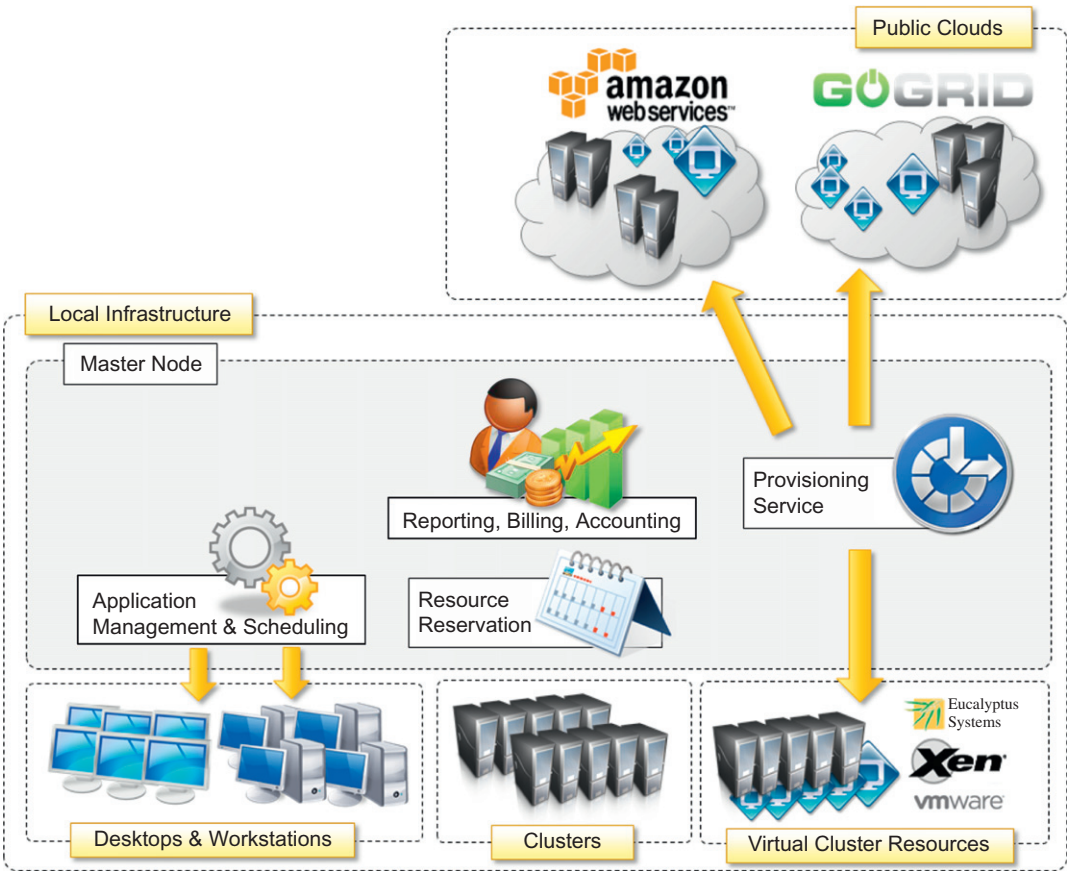

FIGURE 5.7

Hybrid cloud deployment.

Moreover, if the local premises offer some virtual machine management capabilities, it is possible to provide a very efficient use of resources, thus minimizing the expenditure for application execution.

In a hybrid scenario, heterogeneous resources can be used for different purposes. As we discussed in the case of a private cloud deployment, desktop machines can be reserved for low priority workload outside the common working hours. The majority of the applications will be executed on workstations and clusters, which are the nodes that are constantly connected to the Aneka Cloud. Any additional computing capability demand can be primarily addressed by the local virtualization facilities, and if more computing power is required, it is possible to leverage external IaaS providers.

Different from the Aneka Public Cloud deployment is the case in which it makes more sense to leverage a variety of resource providers to provision virtual resources. Since part of the infrastructure is local, a cost in data transfer to the external IaaS infrastructure cannot be avoided. It is then important to select the most suitable option to address application needs. The Resource Provisioning

Service implemented in Aneka exposes the capability of leveraging several resource pools at the same time and configuring specific policies to select the most appropriate pool for satisfying a provisioning request. These features simplify the development of custom policies that can better serve the needs of a specific hybrid deployment.

5.4 Cloud programming and management

Aneka's primary purpose is to provide a scalable middleware product in which to execute distributed applications. Application development and management constitute the two major features that are exposed to developers and system administrators. To simplify these activities, Aneka provides developers with a comprehensive and extensible set of APIs and administrators with powerful and intuitive management tools. The APIs for development are mostly concentrated in the Aneka SDK; management tools are exposed through the Management Console.

5.4.1 Aneka SDK

Aneka provides APIs for developing applications on top of existing programming models, implementing new programming models, and developing new services to integrate into the Aneka Cloud. The development of applications mostly focuses on the use of existing features and leveraging the services of the middleware, while the implementation of new programming models or new services enriches the features of Aneka. The SDK provides support for both programming models and services by means of the *Application Model* and the *Service Model*. The former covers the development of applications and new programming models; the latter defines the general infrastructure for service development.

5.4.1.1 Application model

Aneka provides support for distributed execution in the Cloud with the abstraction of programming models. A programming model identifies both the abstraction used by the developers and the runtime support for the execution of programs on top of Aneka. The *Application Model* represents the minimum set of APIs that is common to all the programming models for representing and programming distributed applications on top of Aneka. This model is further specialized according to the needs and the particular features of each of the programming models.

An overview of the components that define the Aneka Application Model is shown in Figure 5.8. Each distributed application running on top of Aneka is an instance of the *ApplicationBase* $< M >$ class, where *M* identifies the specific type of application manager used to control the application. Application classes constitute the developers' view of a distributed application on Aneka Clouds, whereas application managers are internal components that interact with Aneka Clouds in order to monitor and control the execution of the application. Application managers are also the first element of specialization of the model and vary according to the specific programming model used.

Whichever the specific model used, a distributed application can be conceived as a set of tasks for which the collective execution defines the execution of the application on the Cloud. Aneka further specializes applications into two main categories: (1) applications whose tasks are generated

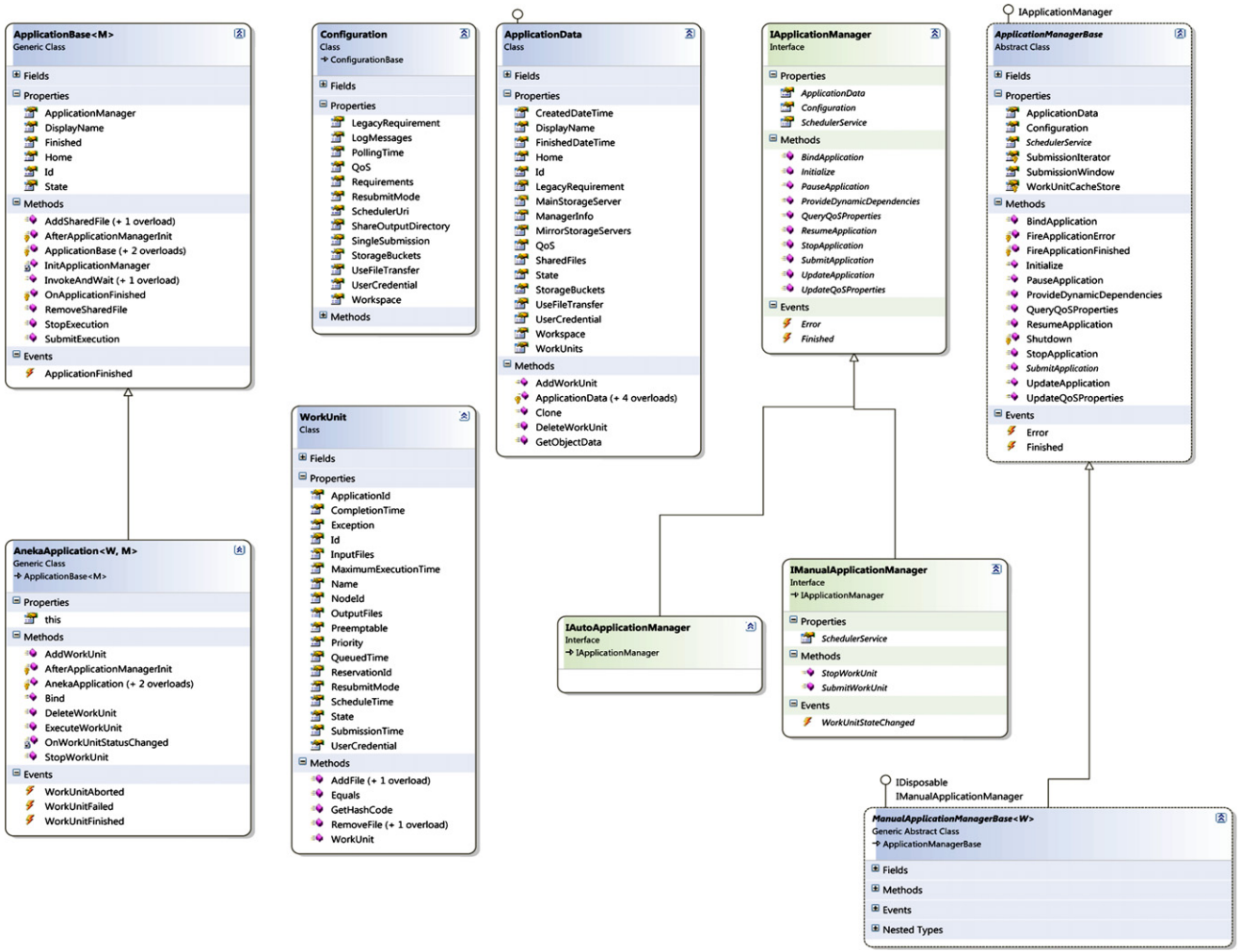

FIGURE 5.8

The Aneka application model.

by the user and (2) applications whose tasks are generated by the runtime infrastructure. These two categories generally correspond to different application base classes and different implementations of the application manager.

The first category is the most common and it is used as a reference for several programming models supported by Aneka: the *Task Model*, the *Thread Model*, and the *Parameter Sweep Model*. Applications that fall into this category are composed of a collection of units of work submitted by the user and represented by the *WorkUnit* class. Each unit of work can have input and output files, the transfer of which is transparently managed by the runtime. The specific type of *WorkUnit* class used to represent the unit of work depends on the programming model used (*AnekaTask* for the *Task Model* and *AnekaThread* for the *Thread Model*). All the applications that fall into this category inherit or are instances *of AnekaApplication* $< W,M >$, where W is the specific type of *WorkUnit* class used, and M is the type of application manager used to implement the *IManualApplicationManager* interface.

The second category covers the case of *MapReduce* and all those other scenarios in which the units of work are generated by the runtime infrastructure rather than the user. In this case there is no common unit-of-work class used, and the specific classes used by application developers strictly depend on the requirements of the programming model used. For example, in the case of the *MapReduce* programming model, developers express their distributed applications in terms of two functions, *map* and *reduce*; hence, the *MapReduceApplication* class provides an interface for specifying the *Mapper* $< K,V >$ and *Reducer* $< K,V >$ types and the input files required by the application. Other programming models might have different requirements and expose different interfaces. For this reason there are no common base types for this category except for *ApplicationBase* $< M >$, where M implements *IAutoApplicationManager*.

A set of additional classes completes the object model. Among these classes, the most notable are the *Configuration* class, which is used to specify the settings required to initialize the application and customize its behavior, and the *ApplicationData* class, which contains the runtime information of the application.

Table 5.1 summarizes the features that are available in the Aneka Application Model and the way they reflect into the supported programming model. The model has been designed to be extensible, and these classes can be used as a starting point to implement a new programming model. This can be done by augmenting the features (or specializing) an existing implementation of a

Table 5.1 Aneka's Application Model Features

Category	Description	Base Application Type	Work Units?	Programming Models
Manual	Units of work are generated by the user and submitted through the application.	*AnekaApplication* $< W,M >$ *IManualApplicationManager* $< W >$ *ManualApplicationManager* $< W >$	Yes	Task Model Thread Model Parameter Sweep Model
Auto	Units of work are generated by the runtime infrastructure and managed internally.	*ApplicationBase* $< M >$ *IAutoApplicationManager*	No	*MapReduce* Model

programming model or by using the base classes to define new models and abstractions. For example, the Parameter Sweep Model is a specialization of the Task Model, and it has been implemented in the context of management of applications on Aneka. It is achieved by providing a different interface to end users who just need to define a template task and the parameters that customize it.

5.4.1.2 Service model

The Aneka *Service Model* defines the basic requirements to implement a service that can be hosted in an Aneka Cloud. The container defines the runtime environment in which services are hosted. Each service that is hosted in the container must be compliant with the *IService* interface, which exposes the following methods and properties:

- Name and status
- Control operations such as *Start*, *Stop*, *Pause*, and *Continue* methods
- Message handling by means of the *HandleMessage* method

Specific services can also provide clients if they are meant to directly interact with end users. Examples of such services might be Resource Provisioning and Resource Reservation Services, which ship their own clients for allowing resource provisioning and reservation. Apart from control operations, which are used by the container to set up and shut down the service during the container life cycle, the core logic of a service resides in its message-processing functionalities that are contained in the *HandleMessage* method. Each operation that is requested to a service is triggered by a specific message, and results are communicated back to the caller by means of messages.

Figure 5.9 describes the reference life cycle of each service instance in the Aneka container. The shaded balloons indicate transient states; the white balloons indicate steady states. A service instance can initially be in the *Unknown* or *Initialized* state, a condition that refers to the creation of the service instance by invoking its constructor during the configuration of the container. Once the container is started, it will iteratively call the *Start* method on each service method. As a result the service instance is expected to be in a *Starting* state until the startup process is completed, after which it will exhibit the *Running* state. This is the condition in which the service will last as long as the container is active and running. This is the only state in which the service is able to process messages. If an exception occurs while starting the service, it is expected that the service will fall back to the *Unknown* state, thus signaling an error.

When a service is running it is possible to pause its activity by calling the *Pause* method and resume it by calling *Continue*. As described in the figure, the service moves first into the *Pausing* state, thus reaching the *Paused* state. From this state, it moves into the *Resuming* state while restoring its activity to return to the *Running* state. Not all the services need to support the pause/continue operations, and the current implementation of the framework does not feature any service with these capabilities.

When the container shuts down, the *Stop* method is iteratively called on each service running, and services move first into the transient *Stopping* state to reach the final *Stopped* state, where all resources that were initially allocated have been released.

Aneka provides a default base class for simplifying service implementation and a set of guidelines that service developers should follow to design and implement services that are compliant with Aneka. In particular, the guidelines define a *ServiceBase* class that can be further extended to

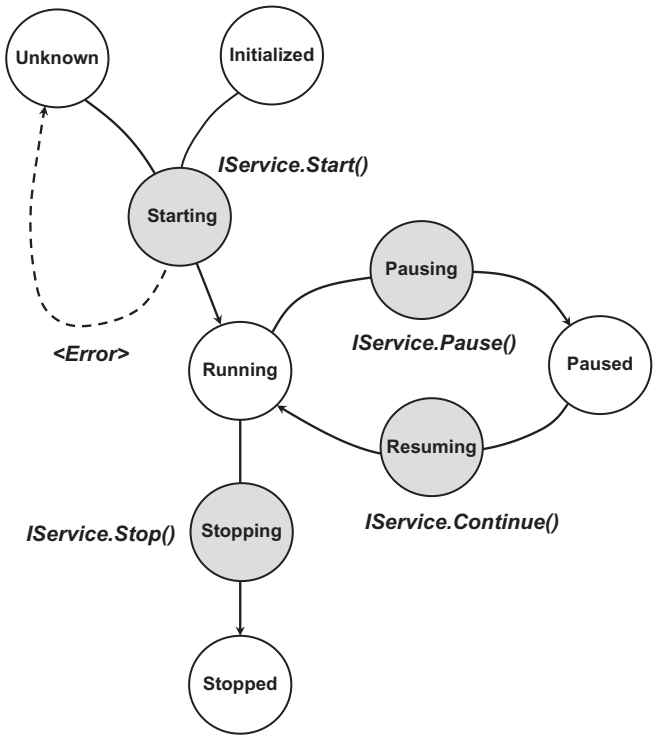

FIGURE 5.9

Service life cycle.

provide a proper implementation. This class is the base class of several services in the framework and provides some built-in features:

- Implementation of the basic properties exposed by *IService*
- Implementation of the control operations with logging capabilities and state control
- Built-in infrastructure for delivering a service specific client
- Support for service monitoring

Developers are provided with template methods for specializing the behavior of control operations, implementing their own message-processing logic, and providing a service-specific client.

Aneka uses a strongly typed message-passing communication model, whereby each service defines its own messages, which are in turn the only ones that the service is able to process. As a result, developers who implement new services in Aneka need also to define the type of messages that the services will use to communicate with services and clients. Each message type inherits from the base class *Message* defining common properties such as:

- Source node and target node
- Source service and target service
- Security credentials

Additional properties are added to carry the specific information for each type. Messages are generally used inside the Aneka infrastructure. In case the service exposes features directly used by applications, they may expose a service client that provides an object-oriented interface to the operations exposed by the service. Aneka features a ready-to-use infrastructure for dynamically injecting service clients into applications by querying the middleware. Services inheriting from the *ServiceBase* class already support such a feature and only need to define an interface and a specific implementation for the service client. Service clients are useful to integrate Aneka services into existing applications that do not necessarily need support for the execution of distributed applications or require access to additional services.

Aneka also provides advanced capabilities for service configuration. Developers can define editors and configuration classes that allow Aneka's management tools to integrate the configuration of services within the common workflow required by the container configuration.

5.4.2 Management tools

Aneka is a pure PaaS implementation and requires virtual or physical hardware to be deployed. Hence, infrastructure management, together with facilities for installing logical clouds on such infrastructure, is a fundamental feature of Aneka's management layer. This layer also includes capabilities for managing services and applications running in the Aneka Cloud.

5.4.2.1 Infrastructure management

Aneka leverages virtual and physical hardware in order to deploy Aneka Clouds. Virtual hardware is generally managed by means of the Resource Provisioning Service, which acquires resources on demand according to the need of applications, while physical hardware is directly managed by the Administrative Console by leveraging the Aneka management API of the PAL. The management features are mostly concerned with the provisioning of physical hardware and the remote installation of Aneka on the hardware.

5.4.2.2 Platform management

Infrastructure management provides the basic layer on top of which Aneka Clouds are deployed. The creation of Clouds is orchestrated by deploying a collection of services on the physical infrastructure that allows the installation and the management of containers. A collection of connected containers defines the platform on top of which applications are executed. The features available for platform management are mostly concerned with the logical organization and structure of Aneka Clouds. It is possible to partition the available hardware into several Clouds variably configured for different purposes. Services implement the core features of Aneka Clouds and the management layer exposes operations for some of them, such as Cloud monitoring, resource provisioning and reservation, user management, and application profiling.

5.4.2.3 Application management

Applications identify the user contribution to the Cloud. The management APIs provide administrators with monitoring and profiling features that help them track the usage of resources and relate them to users and applications. This is an important feature in a cloud computing scenario in which

users are billed for their resource usage. Aneka exposes capabilities for giving summary and detailed information about application execution and resource utilization.

All these features are made accessible through the Aneka Cloud Management Studio, which constitutes the main Administrative Console for the Cloud.

SUMMARY

In this chapter we introduced Aneka, a platform for application programming in the cloud. Aneka is a pure PaaS implementation of the Cloud Computing Reference Model and constitutes a middleware product that enables the creation of computing clouds on top of heterogeneous hardware: desktop machines, clusters, and public virtual resources.

One of the key aspects of Aneka's framework is its configurable runtime environment, which allows for the creation of a service-based middleware where applications are executed. A fundamental element of the infrastructure is the container, which represents the deployment unit of Aneka Clouds. The container hosts a collection of services that define the capabilities of the middleware. Fundamental services in the Aneka middleware are:

- Fabric Services for monitoring, resource provisioning, hardware profiling, and membership
- Foundation Services for storage, resource reservation, billing, accounting, and reporting
- Application Services for scheduling and execution

From an application programming point of view, Aneka provides the capability of supporting different programming models, thus allowing developers to express distributed applications with different abstractions. The framework currently supports three different models: independent "bag of tasks" applications, multithreaded applications, and *MapReduce*.

The infrastructure is extensible, and Aneka provides both an application model and a service model that can be easily extended to integrate new services and programming models.

Review questions

1. Describe in a few words the main characteristics of Aneka.
2. What is the Aneka container and what is its use?
3. Which types of services are hosted inside the Aneka container?
4. Describe Aneka's resource-provisioning capabilities.
5. Describe the storage architecture implemented in Aneka.
6. What is a programming model?
7. List the programming models supported by Aneka.
8. Which are the components that compose the Aneka infrastructure?
9. Discuss the logical organization of an Aneka Cloud.
10. Which services are hosted in a worker node?
11. Discuss the private deployment of Aneka Clouds.
12. Discuss the public deployment of Aneka Clouds.

13. Discuss the role of dynamic provisioning in hybrid deployments.
14. Which facilities does Aneka provide for development?
15. Discuss the major features of the Aneka Application Model.
16. Discuss the major features of the Aneka Service Model.
17. Describe the features of the Aneka management tools in terms of infrastructure, platform, and applications.

This page intentionally left blank

CHAPTER

Concurrent Computing
Thread Programming

6

Throughput computing focuses on delivering high volumes of computation in the form of transactions. Initially related to the field of transaction processing [60], throughput computing has since been extended beyond that domain. Advances in hardware technologies led to the creation of multicore systems, which have made possible the delivery of high-throughput computations, even in a single computer system. In this case, throughput computing is realized by means of multiprocessing and multithreading. *Multiprocessing* is the execution of multiple programs in a single machine, whereas *multithreading* relates to the possibility of multiple instruction streams within the same program.

This chapter presents the concept of multithreading and describes how it supports the development of high-throughput computing applications. It discusses how multithreaded programming, originally conceived to be contained within the boundaries of a single machine, can be extended to a distributed context and which limitations apply. The Aneka Thread Programming Model will be taken as a reference model to review a practical implementation of a multithreaded model for computing clouds.

6.1 Introducing parallelism for single-machine computation

Parallelism has been a technique for improving the performance of computers since the early 1960′s, when Burroughs Corporation designed the D825, the first MIMD multiprocessor ever produced. From there on, a variety of parallel strategies have been developed. In particular, *multiprocessing*, which is the use of multiple processing units within a single machine, has gained a good deal of interest and gave birth to several parallel architectures.

One of the most important distinctions is made in terms of the symmetry of processing units. *Asymmetric multiprocessing* involves the concurrent use of different processing units that are specialized to perform different functions. *Symmetric multiprocessing* features the use of similar or identical processing units to share the computation load. Other examples are *nonuniform memory access (NUMA)* and *clustered multiprocessing*, which, respectively, define a specific architecture for accessing a shared memory between processors and the use of multiple computers joined together as a single virtual computer.

Symmetric and asymmetric multiprocessing are the techniques used to increase the performance of commodity computer hardware. The introduction of *graphical processing units (GPUs)*, which

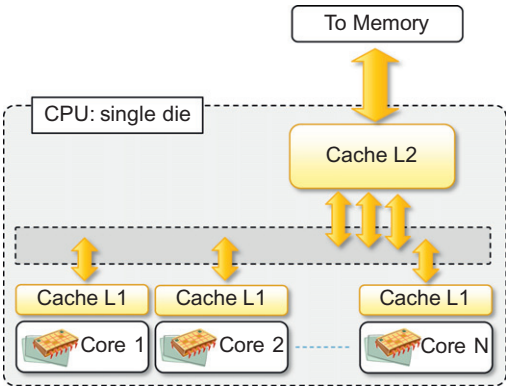

FIGURE 6.1

Multicore processor.

are *de facto* processors, is an application of asymmetric processing, whereas multicore technology is the latest evolution of symmetric multiprocessing. Multiprocessor and especially multicore technologies are now of fundamental importance because of the physical constraint imposed on frequency scaling,[1] which has been the common practice for performance gain in recent years. It became no longer possible to increase the frequency of the processor clock without paying in terms of power consumption and cooling, and this condition became unsustainable in May 2004, when Intel officially cancelled the development of two new microprocessors in favor of multicore development.[2] This date is generally considered the end of the frequency-scaling era and the beginning of multicore technology. Other issues also determined the end of frequency scaling, such as the continuously increasing gap between processor and memory speeds and the difficulty of increasing the instruction-level parallelism[3] in order to keep a single high-performance core busy.

Multicore systems are composed of a single processor that features multiple processing cores that share the memory. Each core has generally its own L1 cache, and the L2 cache is common to all the cores, which connect to it by means of a shared bus, as depicted in Figure 6.1. Dual- and quad-core configurations are quite popular nowadays and constitute the standard hardware configuration for commodity computers. Architectures with multiple cores are also available but are not designed for the commodity market. Multicore technology has been used not only as a support for processor design but also in other devices, such as GPUs and network devices, thus becoming a standard practice for improving performance.

[1] *Frequency scaling* refers to the practice of increasing the clock frequency of a processor to improve its performance. The increase of clock frequency leads to higher power consumption and a higher temperature on the die, which becomes unsustainable over certain values of the frequency clock. Also known as *frequency ramping*, this was the dominant technique for achieving performance gain from the mid-1980s to the end of 2004.

[2] www.nytimes.com/2004/05/08/business/08chip.html?ex = 1399348800&en = 98cc44ca97b1a562&ei = 5007.

[3] *Instruction-level parallelism* (ILP) is a measure of how many operations a computer program can perform at one time. There are several techniques that can be applied to increase the ILP at the microarchitectural level. One of these is *instruction pipelining*, which involves the division of instructions into stages so that a single processing unit can execute multiple instructions at the same time by carrying out different stages for each of them.

Multiprocessing is just one technique that can be used to achieve parallelism, and it does that by leveraging parallel hardware architectures. Parallel architectures are better exploited when programs are designed to take advantage of their features. In particular, an important role is played by the operating system, which defines the runtime structure of applications by means of the abstraction of *process* and *thread*. A process is the runtime image of an application, or better, a program that is running, while a thread identifies a single flow of the execution within a process. A system that allows the execution of multiple processes at the same time supports *multitasking*. It supports *multithreading* when it provides structures for explicitly defining multiple threads within a process.

Note that both multitasking and multithreading can be implemented on top of computer hardware that is constituted of a single processor and a single core, as was the common practice before the introduction of multicore technology. In this case, the operating system gives the illusion of concurrent execution by interleaving the execution of instructions of different processes and of different threads within the same process. This is also the case in multiprocessor/multicore systems, since the number of threads or processes is higher than the number of processors or cores. Nowadays, almost all the commonly used operating systems support multitasking and multithreading. Moreover, all the mainstream programming languages incorporate the abstractions of process and thread within their APIs, whereas direct support of multiple processors and cores for developers is very limited and often reduced and confined to specific libraries, which are available for a subset of the programming languages such as C/C++.

In this chapter, we concentrate our attention on multithreaded programming, which now has full support and constitutes the simplest way to achieve parallelism within a single process, despite the underlying hardware architecture.

6.2 Programming applications with threads

Modern applications perform multiple operations at the same time. Developers organize programs in terms of threads in order to express intraprocess concurrency. The use of threads might be implicit or explicit. *Implicit threading* happens when the underlying APIs use internal threads to perform specific tasks supporting the execution of applications such as graphical user interface (GUI) rendering, or garbage collection in the case of virtual machine-based languages. *Explicit threading* is characterized by the use of threads within a program by application developers, who use this abstraction to introduce parallelism. Common cases in which threads are explicitly used are I/O from devices and network connections, long computations, or the execution of background operations for which the outcome does not have specific time bounds. The use of threads was initially directed to allowing asynchronous operations—in particular, providing facilities for asynchronous I/O or long computations so that the user interface of applications did not block or became unresponsive. With the advent of parallel architectures the use of multithreading has become a useful technique to increase the throughput of the system and a viable option for throughput computing. To this purpose, the use of threads strongly impacts the design of algorithms that need to be refactored in order to leverage threads. In this section, we discuss the use of threading as a support for the design of parallel and distributed algorithms.

6.2.1 What is a thread?

A *thread* identifies a single control flow, which is a *logical sequence of instructions*, within a process. By logical sequence of instructions, we mean a sequence of instructions that have been designed to be executed one after the other one. More commonly, a thread identifies a kind of yarn that is used for sewing, and the feeling of continuity that is expressed by the interlocked fibers of that yarn is used to recall the concept that the instructions of thread express a logically continuous sequence of operations.

Operating systems that support multithreading identify threads as the minimal building blocks for expressing running code. This means that, despite their explicit use by developers, any sequence of instruction that is executed by the operating system is within the context of a thread. As a consequence, each process contains at least one thread but, in several cases, is composed of many threads having variable lifetimes. Threads within the same process share the memory space and the execution context; besides this, there is no substantial difference between threads belonging to different processes.

In a multitasking environment the operating system assigns different time slices to each process and interleaves their execution. The process of temporarily stopping the execution of one process, saving all the information in the registers (and in general the state of the CPU in order to restore it later), and replacing it with the information related to another process is known as a *context switch*. This operation is generally considered demanding, and the use of multithreading minimizes the latency imposed by context switches, thus allowing the execution of multiple tasks in a lighter fashion. The state representing the execution of a thread is minimal compared to the one describing a process. Therefore, switching between threads is a preferred practice over switching between processes. Obviously the use of multiple threads in place of multiple processes is justified if and only if the tasks implemented are logically related to each other and require sharing memory or other resources. If this is not the case, a better design is provided by separating them into different processes.

Figure 6.2 provides an overview of the relation between threads and processes and a simplified representation of the runtime execution of a multithreaded application. A running program is identified by a process, which contains at least one thread, also called the *main thread*. Such a thread is implicitly created by the compiler or the runtime environment executing the program. This thread is likely to last for the entire lifetime of the process and be the origin of other threads, which in general exhibit a shorter duration. As main threads, these threads can spawn other threads. There is no difference between the main thread and other threads created during the process lifetime. Each of them has its own local storage and a sequence of instructions to execute, and they all share the memory space allocated for the entire process. The execution of the process is considered terminated when all the threads are completed.

6.2.2 Thread APIs

Even though the support for multithreading varies according to the operating system and the specific programming languages that are used to develop applications, it is possible to identify a minimum set of features that are commonly available across all the implementations.

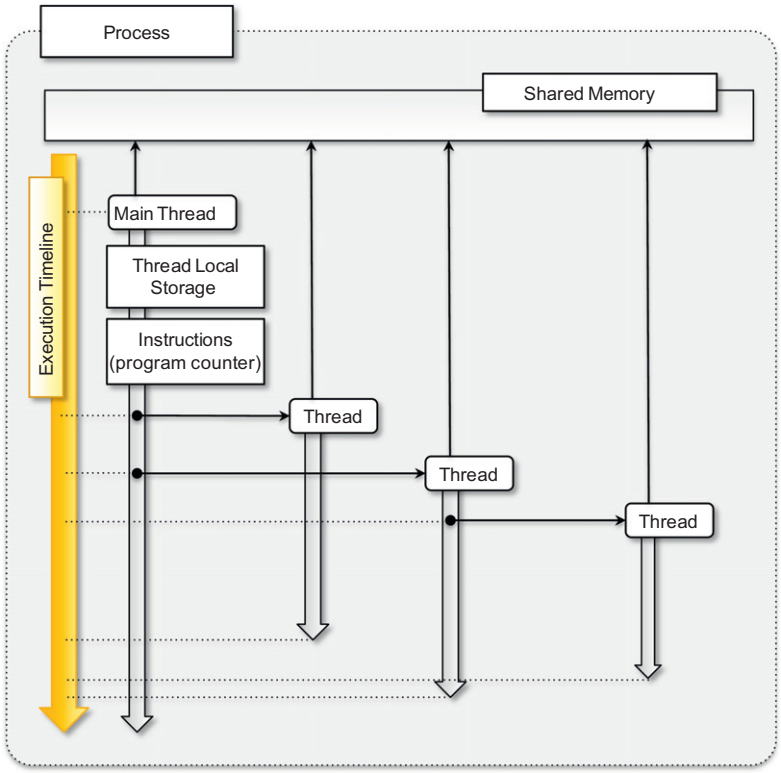

FIGURE 6.2

The relationship between processes and threads.

6.2.2.1 POSIX Threads

Portable Operating System Interface for Unix (POSIX) is a set of standards related to the application programming interfaces for a portable development of applications over the Unix operating system flavors. Standard POSIX 1.c (IEEE Std 1003.1c-1995) addresses the implementation of threads and the functionalities that should be available for application programmers to develop portable multithreaded applications. The standards address the Unix-based operating systems, but an implementation of the same specification has been provided for Windows-based systems.

The POSIX standard defines the following operations: creation of threads with attributes, termination of a thread, and waiting for thread completion (join operation). In addition to the logical structure of a thread, other abstractions, such as semaphores, conditions, reader-writer locks, and others, are introduced in order to support proper synchronization among threads.

The model proposed by POSIX has been taken as a reference for other implementations that might provide developers with a different interface but a similar behavior. What is important to remember from a programming point of view is the following:

- A thread identifies a logical sequence of instructions.
- A thread is mapped to a function that contains the sequence of instructions to execute.

- A thread can be created, terminated, or joined.
- A thread has a state that determines its current condition, whether it is executing, stopped, terminated, waiting for I/O, etc.
- The sequence of states that the thread undergoes is partly determined by the operating system scheduler and partly by the application developers.
- Threads share the memory of the process, and since they are executed concurrently, they need synchronization structures.
- Different synchronization abstractions are provided to solve different synchronization problems.

A default implementation of the POSIX 1.c specification has been provided for the C language. All the available functions and data structures are exposed in the *pthread.h* header file, which is part of the standard C implementations.

6.2.2.2 Threading support in java and .NET

Languages such as Java and C# provide a rich set of functionalities for multithreaded programming by using an object-oriented approach. Since both Java and .NET execute code on top of a virtual machine, the APIs exposed by the libraries refer to managed or logical threads. These are mapped to physical threads (i.e., those made available as abstractions by the underlying operating system) by the runtime environment in which programs developed with these languages execute. Despite such a mapping process, managed threads are considered, from a programming point of view, as physical threads and expose the same functionalities.

Both Java and .NET express the thread abstraction with the class *Thread* exposing the common operations performed on threads: *start*, *stop*, *suspend*, *resume*, *abort*, *sleep*, *join*, and *interrupt*. *Start* and *stop/abort* are used to control the lifetime of the thread instance, while *suspend* and *resume* are used to programmatically pause and then continue the execution of a thread. These two operations are generally deprecated in both of the two implementations that favor the use of appropriate techniques involving proper locks of the use of the *sleep* operation. This operation allows pausing the execution of a thread for a predefined period of time. This one is different from the *join* operation that makes one thread wait until another thread is completed. These waiting states can be interrupted by using the *interrupt* operation, which resumes the execution of the thread and generates an exception within the code of the thread to notify the abnormal resumption.

The two frameworks provide different support for implementing synchronization among threads. In general the basic features for implementing mutexes, critical regions, and reader-writer locks are completely covered by means of the basic class libraries or additional libraries. More advanced constructs than the thread abstraction are available in both languages. In the case of Java, most of them are contained in the *java.util.concurrent*[4] package, whereas the rich set of APIs for concurrent programming in .NET is further extended by the *.NET Parallel Extension* framework.[5]

[4] http://download.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html.
[5] http://msdn.microsoft.com/en-us/concurrency/default.aspx.

6.2.3 Techniques for parallel computation with threads

Developing parallel applications requires an understanding of the problem and its logical structure. Understanding the dependencies and the correlation of tasks within an application is fundamental to designing the right program structure and to introducing parallelism where appropriate. *Decomposition* is a useful technique that aids in understanding whether a problem is divided into components (or tasks) that can be executed concurrently. If such decomposition is possible, it also provides a starting point for a parallel implementation, since it allows the breaking down into independent units of work that can be executed concurrently with the support provided by threads. The two main decomposition/partitioning techniques are *domain* and *functional* decompositions.

6.2.3.1 Domain decomposition

Domain decomposition is the process of identifying patterns of *functionally repetitive, but independent, computation on data*. This is the most common type of decomposition in the case of throughput computing, and it relates to the identification of repetitive calculations required for solving a problem.

When these calculations are identical, only differ from the data they operate on, and can be executed in any order, the problem is said to be *embarrassingly parallel* [59]. Embarrassingly parallel problems constitute the easiest case for parallelization because there is no need to synchronize different threads that do not share any data. Moreover, coordination and communication between threads are minimal; this strongly simplifies the code logic and allows a high computing throughput.

In many cases it is possible to devise a general structure for solving such problems and, in general, problems that can be parallelized through domain decomposition. The master-slave model is a quite common organization for these scenarios:

- The system is divided into two major code segments.
- One code segment contains the decomposition and coordination logic.
- Another code segment contains the repetitive computation to perform.
- A master thread executes the first code segment.
- As a result of the master thread execution, as many slave threads as needed are created to execute the repetitive computation.
- The collection of the results from each of the slave threads and an eventual composition of the final result are performed by the master thread.

Although the complexity of the repetitive computation strictly depends on the nature of the problem, the coordination and decomposition logic is often quite simple and involves identifying the appropriate number of units of work to create. In general, a *while* or a *for* loop is used to express the decomposition logic, and each iteration generates a new unit of work to be assigned to a slave thread. An optimization, of this process involves the use of thread pooling to limit the number of threads used to execute repetitive computations.

Several practical problems fall into this category; in the case of embarrassingly parallel problems, we can mention:

- Geometrical transformation of two (or higher) dimensional data sets
- Independent and repetitive computations over a domain such as Mandelbrot set and Monte Carlo computations

Even though embarrassingly parallel problems are quite common, they are based on the strong assumption that at each of the iterations of the decomposition method, it is possible to isolate an independent unit of work. This is what makes it possible to obtain a high computing throughput. Such a condition is not met if the values of all the iterations are dependent on some of the values obtained in the previous iterations. In this case, the problem is said to be *inherently sequential*, and it is not possible to directly apply the methodology described previously. Despite this, it can still be possible to break down the whole computation into a set of independent units of work, which might have a different granularity—for example, by grouping into single computation-dependent iterations. Figure 6.3 provides a schematic representation of the decomposition of embarrassingly parallel and inherently sequential problems.

To show how domain decomposition can be applied, it is possible to create a simple program that performs matrix multiplication using multiple threads.

Matrix multiplication is a binary operation that takes two matrices and produces another matrix as a result. This is obtained as a result of the composition of the linear transformation of the original matrices. There are several techniques for performing matrix multiplication; among them, the *matrix product* is the most popular. Figure 6.4 provides an overview of how a matrix product can be performed.

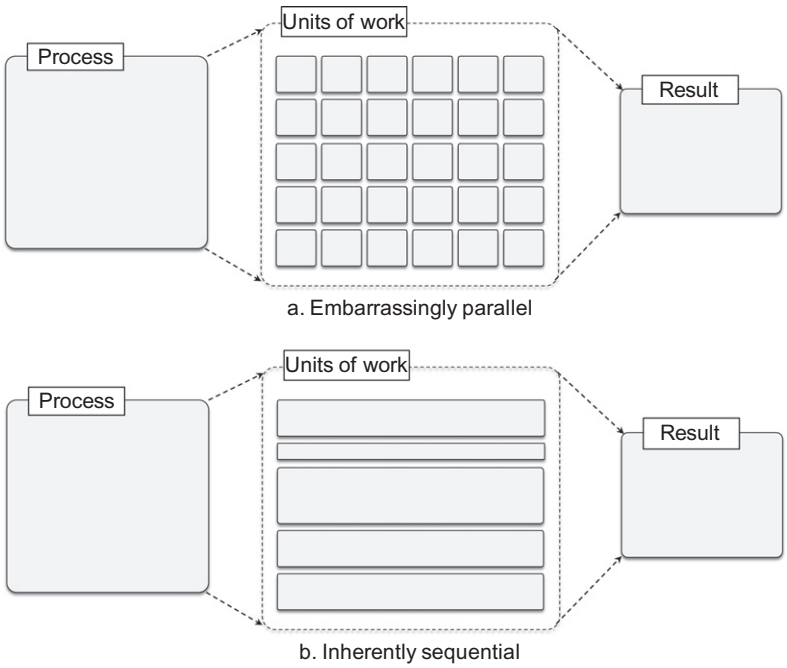

FIGURE 6.3

Domain decomposition techniques.

The matrix product computes each element of the resulting matrix as a linear combination of the corresponding row and column of the first and second input matrices, respectively. The formula that applies for each of the resulting matrix elements is the following:

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}$$

Therefore, two conditions hold in order to perform a matrix product:

- Input matrices must contain values of a comparable nature for which the scalar product is defined.
- The number of columns in the first matrix must match the number of rows of the second matrix.

Given these conditions, the resulting matrix will have the number of rows of the first matrix and the number of columns of the second matrix, and each element will be computed as described by the preceding equation.

It is evident that the repetitive operation is the computation of each of the elements of the resulting matrix. These are subject to the same formula, and the computation does not depend on values that have been obtained by the computation of other elements of the resulting matrix. Hence, the problem is embarrassingly parallel, and we can logically organize the multithreaded program in the following steps:

- Define a function that performs the computation of the single element of the resulting matrix by implementing the previous equation.
- Create a double for loop (the first index iterates over the rows of the first matrix and the second over the columns of the second matrix) that spawns a thread to compute the elements of the resulting matrix.
- Join all the threads for completion, and compose the resulting matrix.

In order to give a practical example of the implementation of such a solution, we demonstrate the use of .NET threading. The .NET framework provides the *System.Threading.Thread* class that

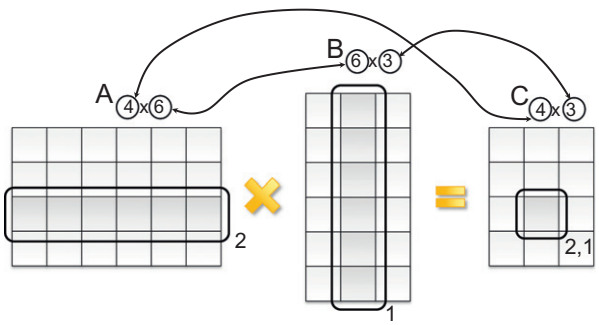

FIGURE 6.4

A matrix product.

can be configured with a function pointer, also known as a *delegate*, to execute asynchronously. Such a delegate must reference a defined method in some class. Hence, we can define a simple class that exposes as properties the row and the column to multiply and the result value. This class will also define the method for performing the actual computation. Listing 6.1 shows the class *ScalarProduct*.

The creation of the main thread of control is very simple. In this case, we skip the boilerplate code that is required to read the matrices from the standard input or from a file and concentrate our attention on the main control logic that decomposes the computation, creates threads, and waits for their completion in order to compose the resulting matrix.

To control the threads, we need to keep track of them so that we can query their status and obtain the result once they have completed the computation. We can create a simple program that reads the matrices, keeps track of all the threads in an appropriate data structure, and, once the threads have been completed, composes the final result. Listing 6.2 shows the content of the *MatrixProduct* class with some omissions.

Whereas the domain decomposition is quite simple, note that most of the complexity of the program resides in the management of threads. A few issues arise from the previous implementation:

- *Matrix layout.* Because of the way in which multidimensional arrays are stored, retrieving the column for the scalar product is not as straightforward as obtaining the row. This problem can be easily solved by memorizing the second matrix as *columns* × *rows* rather than *rows* × *columns*.
- *Result composition.* The composition of results is made on the master thread, and this requires keeping track of all the worker threads. Maintaining a reference to all the worker threads is in general a good programming practice, since it is necessary to terminate all of them before the application completes; but in this case it is possible to modify the application by using synchronization constructs that allow updating the resulting matrix from the worker threads. The new design implies storing the information about the indexes of rows and columns and a reference to the resulting matrix in the *ScalarProduct* class. As a result, there is no need to maintain a dictionary for threads, and we do not need the *ComposeResult* method in the master thread.

The example of a matrix product has been taken as a model to sketch the basic logic that is required to implement domain decomposition for an embarrassingly parallel problem and how to use threads in .NET to achieve throughput computing. This example can be taken as a reference to develop more sophisticated applications.

6.2.3.2 Functional decomposition

Functional decomposition is the process of identifying *functionally distinct but independent computations*. The focus here is on the type of computation rather than on the data manipulated by the computation. This kind of decomposition is less common and does not lead to the creation of a large number of threads, since the different computations that are performed by a single program are limited.

Functional decomposition leads to a natural decomposition of the problem in separate units of work because it does not involve partitioning the dataset, but the separation among them is clearly

```
///<summary>
/// Class ScalarProduct. Computes the scalar product between the row and the column
/// arrays.
///</summary>
public class ScalarProduct
{
   /// <summary>
   /// Scalar product.
   /// </summary>
   private double result;
   /// <summary>
   /// Gets the resulting scalar product.
   /// </summary>
   public double Result{ get { returnthis.result; } }

   /// <summary>
   /// Arrays containing the elements of the row and the column to multiply.
   /// </summary>
   private double[] row, column;

   /// <summary>
   /// Creates an instance of the ScalarProduct class and configures it with the given
   /// row and column arrays.
   /// </summary>
   /// <param name="row">Array with the elements of the row to be multiplied.</param>
   /// <param name="column">Array with the elements of the column to be multiplied.
   /// </param>
   public ScalarProduct(double[] row, double[] column)
   {
      this.row = row;
      this.colum = colum;
   }
   /// <summary>
   /// Executes the scalar product between the row and the colum.
   /// </summary>
   /// <param name="row">Array with the elements of the row to be multiplied.</param>
   /// <param name="column">Array with the elements of the column to be multiplied.
   /// </param>
   public void Multiply()
   {
      this.result = 0;
      for(int i=0; i<this.row.Length; i++)
      {
         this.result += this.row[i] * this.column[i];
      }
   }
}
```

LISTING 6.1

ScalarProduct Class.

```
using System;
using System.Threading;
using System.Collections.Generic;

///<summary>
/// Class MatrixProduct. Performs the matrix product of two matrices.
///</summary>
public class MatrixProduct
{
   ///<summary>
   /// First and second matrix of the produt.
   ///</summary>
   private static double[,]a, b;
   ///<summary>
   /// Result matrix.
   ///</summary>
   private static double[,] c;
   ///<summary>
   /// Dictionary mapping the thread instances to the corresponding ScalarProduct
   /// instances that are run inside.
   ///</summary>
   private static IDictionary<Thread, ScalarProduct>workers.

   ///<summary>
   /// Read the command line parameters and perform the scalar product.
   ///</summary>
   ///<param name="args">Array strings containing the command line parameters.</param>
   public static void Main(string[] args)
   {
       // reads the input matrices a and b.
       MatrixProduct.ReadMatrices();
       // executes the parallel matrix product.
       MatrixProduct.ExecuteProudct();
       // waits for all the threads to complete and
       // composes the final matrix.
       MatrixProduct.ComposeResult();
   }

   ///<summary>
   /// Executes the parallel matrix product by decomposing the problem in
   /// independent scalar product between rows and colums.
   ///</summary>
   private static void ExecuteThreads()
   {
       MatrixProduct.workers = newList<Thread>();
       int rows = MatrixProduct.a.Length;
       // in .NET matrices are arrays of arrays and the number of columns is
       // is represented by the length of the second array.
       int columns = MatrixProduct.b[0].Length;
       for(int i=0; i<rows; i++)
          for(int j=0; j<columns; j++)
          {
             double[] row = MatrixProduct.a[i];
             // because matrices are stored as arrays of arrays in order to
             // to get the columns we need to traverse the array and copy the
             // the data to another array.
```

LISTING 6.2

MatrixProduct Class (Main Program).

```
                double[] column = new double[common];
                for(int k=0; k<common; k++)
                {
                    column[j] = MatrixProduct.b[j][i];
                }
                // creates a ScalarProduct instance with the previous rows and
                // columns and starts a thread executing the Multiply method.
                ScalarProduct scalar = newScalarProduct(row, column);
                Thread worker = newThread(newThreadStart(scalar.Multiply));
                worker.Name =string.Format("{0}.{1}",row,column);
                worker.Start();
                // adds the thread to the dictionary so that it can be
                // further retrieved.
                MatrixProduct.workers.Add(worker, scalar);
            }
        }

        ///<summary>
        /// Waits for the completion of all the threads and composes the final
        /// result matrix.
        ///</summary>
        private static void ComposeResult()
        {
            MatrixProduct.c = new double[rows,columns];
            foreach(KeyValuePair<Thread,ScalarProduct>pair in MatrixProduct.workers)
            {
                Thread worker = pair.Key;
                // we have saved the coordinates of each  scalar product in the name
                // of the thread now we get them back by parsing the name .
                string[] indices = string.Split(worker.Name, new char[] {'.'});
                int i = int.Parse(indices[0]);
                int j = int.Parse(indices[1]);
                // we wait for the thread to complete
                worker.Join();
                // we set the result computed at the given coordinates.
                MatrixProduct.c[i,j] = pair.Value.Result;
            }
            MatrixProduct.PrintMatrix(MatrixProduct.c);
        }
        ///<summary>
        /// Reads the matrices.
        ///</summary>
        private static void ReadMatrices()
        {
            // code for reading the matrices a and b
        }
        ///<summary>
        /// Prints the given matrix.
        ///</summary>
        ///<param name="matrix">Matrix to print.</param>
        private static void PrintMatrices(double[,] matrix)
        {
            // code for printing the matrix.
        }
    }
```

LISTING 6.2

(Continued)