# Mastering Cloud Computing

**Rajkumar Buyya**
*The University of Melbourne and Manjrasoft Pvt Ltd,*
*Melbourne, Australia*

**Christian Vecchiola**
*The University of Melbourne and IBM Research,*
*Melbourne, Australia*

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


**McGraw Hill Education (India) Private Limited**
NEW DELHI

*McGraw Hill Education Offices*

**New Delhi** New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto

# Contents

# 1 Introduction

Computing is being transformed to 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 they 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 in which 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 of its key technologies, as well as some insights into developments 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, like other utility services such as water, electricity, telephone, and gas 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 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 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 on demand their infrastructure and size it according to their business needs. End users leveraging Cloud computing services can access their documents and data at anytime, anywhere, and from any device connected to the Internet. Many other points of view exist[1]. One of the most diffused 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 with the way we make use of 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 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, whose sizing can be difficult to estimate and needs are limited in time. This is one of the most important advantages of Cloud computing, which made it a popular phenomenon. With the wide deployment of Cloud computing systems, the foundation technologies and systems enabling them are getting consolidated and standardized. This is a fundamental step in the realization of the long-term vision for Cloud computing, which provides an open environment where computing, storage, and other services are traded as computing utilities.

## 1.1.1 The Vision of Cloud Computing

Cloud computing allows anyone having a credit card to provision virtual hardware, runtime environments, and services. These are used for as long as needed and no upfront commitments are 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

[1] An interesting perspective on how Cloud Computing evocates 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. CEOs, 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: http://www.youtube.com/watch?v=6PNuQHUiV3Q.

application domains and business sectors (see Fig. 1.1). The demand has fast-tracked the technical development and enriched the set of services offered, which have also become more sophisticated and cheaper.

Despite its evolution, the usage of Cloud computing is often limited to a single service at time or, more commonly, a set of related services offered by the same vendor. 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 one. The capability of Web-based access to documents and their processing using sophisticated applications is one the appealing factors for end-users.

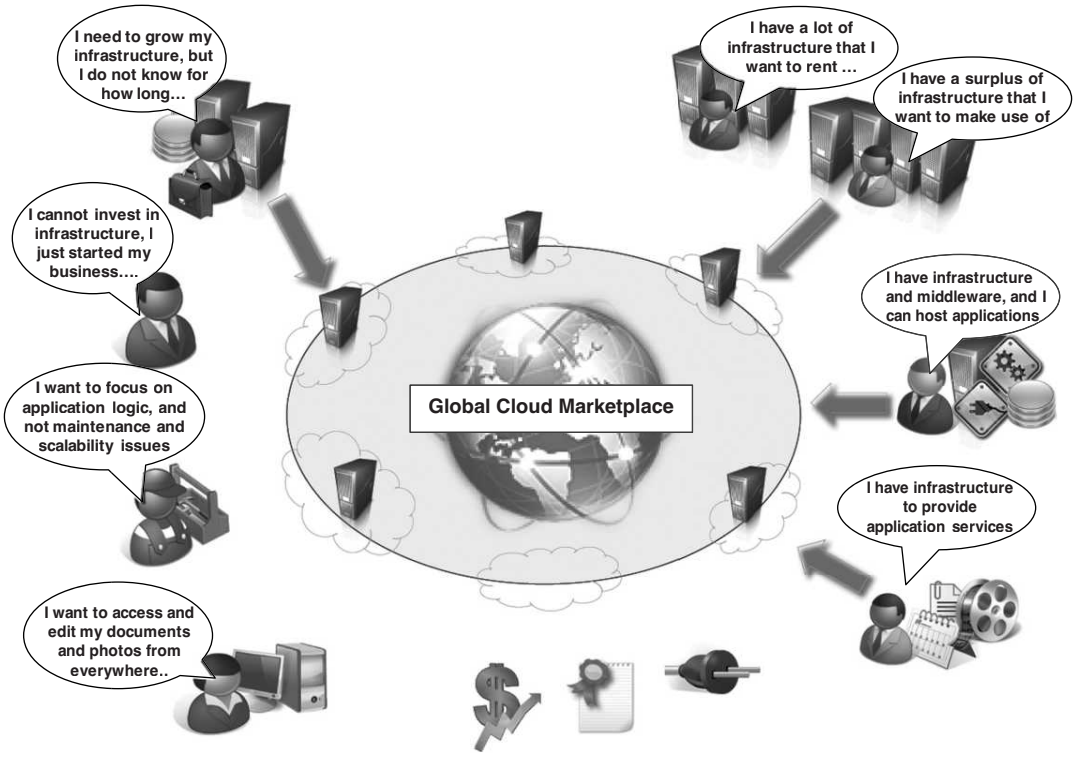


**Fig. 1.1.** Cloud-Computing Vision.

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. In a near future, we imagine that 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 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 to become more visible, and therefore to 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 promises to customers.

These are all possibilities that are introduced with the establishment of a global Cloud computing market place and by defining an effective standard for the unified representation of Cloud services as well as the interaction among different Cloud technologies. A considerable shift towards 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 capabilities to serve multiple users. This consolidation model will reduce the waste of energy and carbon emission, thus contributing to a greener IT on one end, and increase the revenue on the other end.

## 1.1.2 Defining a Cloud

Cloud computing has become a popular buzzword and 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 one portrays when defining Cloud computing.

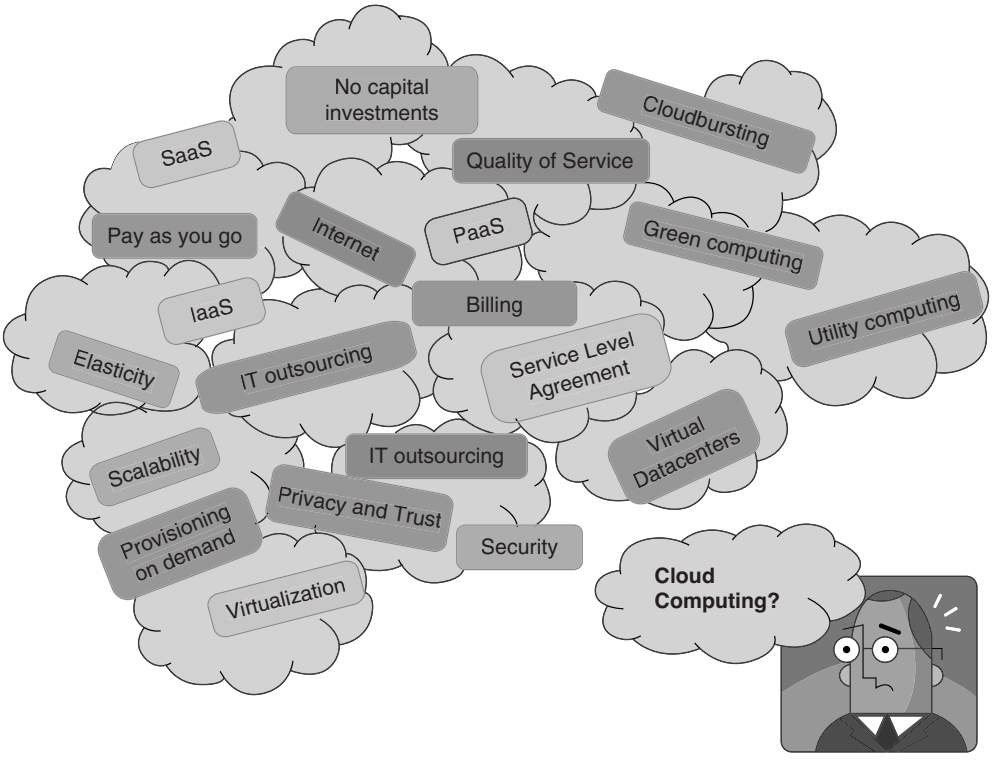


**Fig. 1.2.** Cloud Computing Technologies, Concepts, and Ideas.

The term “Cloud” has historically been used in the telecommunication industry as an abstraction of the network in system diagrams. It then became the symbol of the most popular computer network: Internet. This meaning also applies to Cloud computing, which refers to an Internet-centric way of doing computing. 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 into 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 can be delivered, measured and consequently priced, as a service: IT infrastructure, development platforms, databases, and so on. This new approach significantly influences not only the way in which we build software, but also the way in which we deploy it, make it accessible, design our IT infrastructure, and even the way in which 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 IT infrastructure in public Cloud. This notion of multiple parties using shared Cloud computing environment is highlighted in a definition proposed by American 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 of the cases a “pay-per-use” strategy. It makes possible to access online storage, to rent virtual hardware, or to use development platforms and pay only for their effective usage, with no or minimal upfront 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 to 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 (non-proprietary) or Web services 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 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.”*

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

### 1.1.3 A Closer Look

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

**(a) Large enterprises can offload some of their activities to Cloud-based systems.** Recently, the New York Times has converted its digital library about 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, it performed this task in 36 hours, and relinquished these resources without any additional costs.

**(b) Small enterprises and start-ups can afford to translate into business results their ideas more quickly without excessive upfront 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 is sized on demand according to the overall workload to be processed. Such workload can vary a lot and requires instant scalability[3]. Upfront investment is clearly not an effective solution and Cloud computing systems become an appropriate alternative.

**(c) 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 rental bicycle services. The company has managed to back the widget's computing needs on Google AppEngine and be on market in only one week.

**(d) 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 they connect to it. This makes it possible taking a picture with a smart phone, going back home and editing the same picture on your laptop, and having it shown updated on their tablet. This process is completely transparent to the users who do not have to set up cables and connect these devices with each other.

How all of this is made possible? The same concept of IT services on demand—whether they are computing power, storage, or runtime environments for applications—on a pay-as-you-go basis accommodates these four different scenarios. Cloud computing does not only contribute with the opportunity of easily accessing IT services on demand, but also introduces a new thinking about how IT services and resources should be perceived: as utilities. A bird eye view of Cloud computing environment is shown in Fig. 1.3.

The three major models for deployment and accessibility of Cloud computing environments are: public Clouds, private/enterprise Cloud, and hybrid Clouds (see Fig. 1.4). *Public Clouds* are the most common deployment models in which necessary IT infrastructure (e.g., virtualized Data Center) is established by a 3rd party service provider who makes it available to any consumer on subscription basis. Such Clouds are appealing to users as they allow them to quickly leverage compute, storage,

[3] It has been reported that Animoto, in one single week, scaled from 70 to 8500 servers because of the user demand.

and application services. In this environment, users' data and applications are deployed on Cloud Data centers on the vendor's premises.

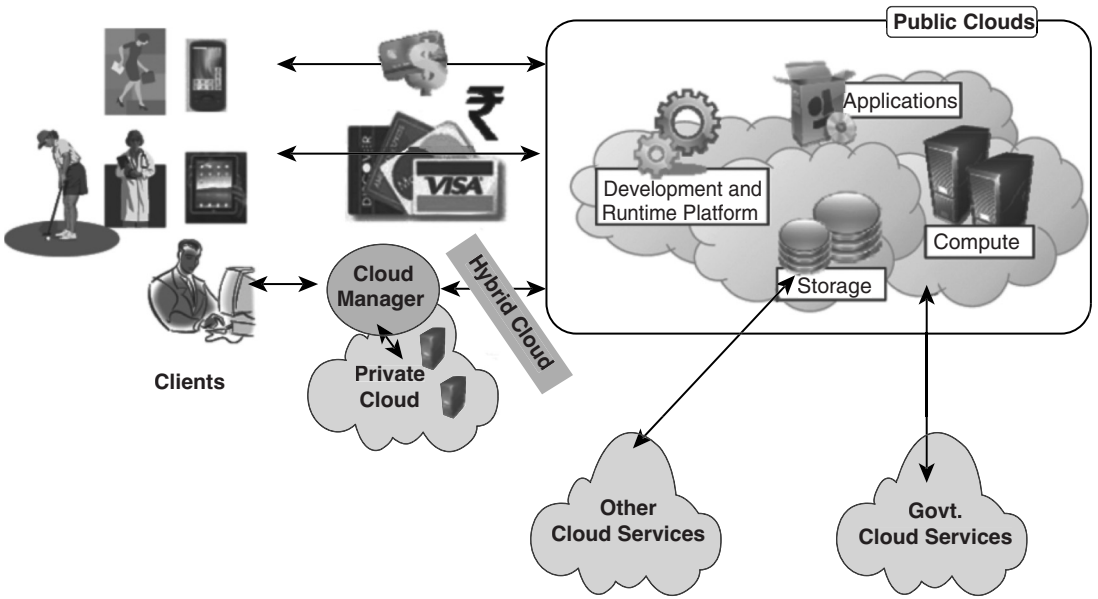


**Fig. 1.3.** A Bird's Eye View of Cloud Computing.

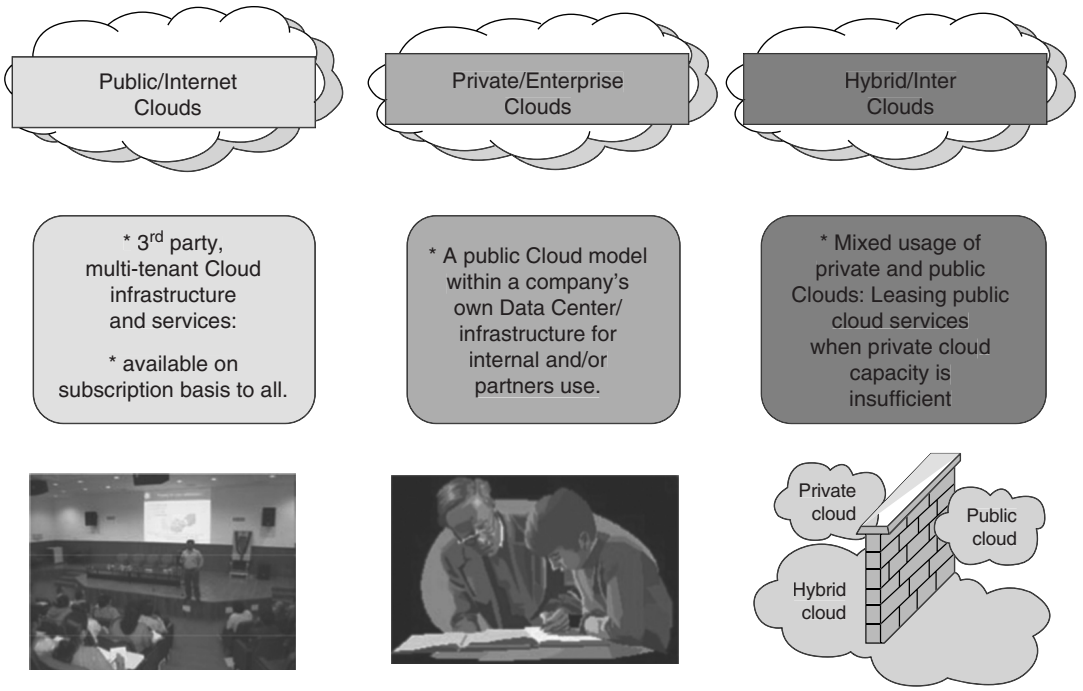


**Fig. 1.4.** Major Deployment Models for Cloud Computing.

Large organizations, owning massive computing infrastructures, can still benefit from Cloud computing by replicating the Cloud IT service delivery model in-house. This has given birth to the concept of *private Cloud*, as opposed to the term *public Cloud*. In 2010, the U.S. federal government, one of the world's largest consumers of IT spending around $76 billion on more than 10000 systems, has started a Cloud computing initiative aimed at providing the 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 of keeping confidential information within the organization's premises. Institutions such as governments and banks with 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 such the deadline, hybrid computing systems, partially composed by public Cloud resources and privately owned infrastructures, are created to serve the organization's need. These are often referred as *hybrid Clouds,* which are becoming a common way to start exploring the possibilities offered by Cloud computing by many stakeholders.

## 1.1.4 Cloud-Computing Reference Model

A fundamental characteristic of Cloud computing is the capability of delivering on demand a variety of IT services, which are quite diverse from each other. This variety creates a different perception of what Cloud computing is among users. Despite this, 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 Fig. 1.5, which provides an organic view of Cloud computing. We refer to this diagram as *"Cloud Computing Reference Model"* and we will use it throughout the 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.

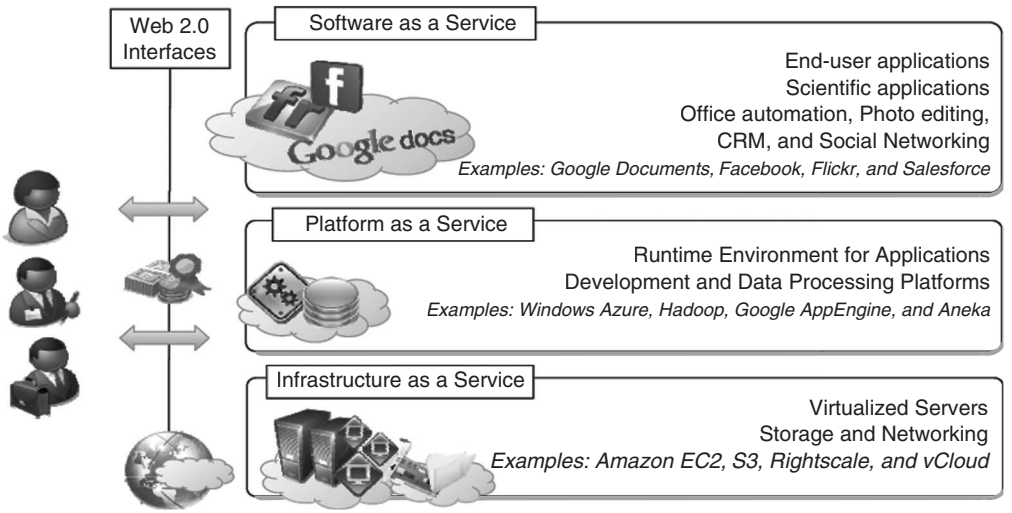


**Fig. 1.5.** Cloud-Computing Reference Model.

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 machines instances. These are created on 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 hours, 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 towards 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 that 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, 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 that want to leverage Cloud computing from building dynamically scalable computing systems requiring a specific software stack. IaaS services are therefore used to develop scalable Web sites or for background processing. PaaS solutions provide scalable programming platforms for developing applications, and are more appropriate when new systems have to be developed. IaaS 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 user's needs (i.e., 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). They are

- no upfront commitments;
- on demand access;
- nice pricing;
- simplified application acceleration and scalability;
- efficient resource allocation;
- energy efficiency; and
- seamless creation and the 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 and not upfront. Capital costs are costs associated to 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 upfront to afford a computing infrastructure enabling the business activities of an 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 to depreciable values. This is the case of hardware: a server bought today for 1000 dollars will have a market value less than its original price when it will be replaced by a new hardware. In order to make profit, organizations have also to compensate this depreciation created by

time, thus reducing the net gain obtained from revenue. Minimizing capital costs is then fundamental. Cloud computing transforms IT infrastructure and software into utilities, thus significantly contributing in increasing the net gain. Moreover, it 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 not responsible anymore for their maintenance. This task is the responsibility of the Cloud service provider, who, thanks to the economies of scale, can bear maintenance costs.

Increased agility in defining and structuring software systems is another significant benefit. 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 to react to unplanned surges in demand quite rapidly. For example, organizations can add more servers to process workload spikes, and dismiss them when there is no longer need. 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 run-time 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 (i.e., 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 with 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. Multi-tenancy allows for a 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 the 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 and new interesting problems and challenges are 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 they should 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. Also, 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 secutity standards, which give a minimum guarantee on the treatment of information on Cloud-computing system, are sought.

The problem is not as evident as it seems: even though cryptography can help in securing 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 for processing. This is the weak point of the chain: since virtualization allows capturing almost transparently the memory pages of an instance, these data could be easily 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 the privacy in different countries may potentially create disputes on what are the rights that third parties (including government agencies) have on your data. American legislation is known to give extreme powers to government agencies to aquire confidential data, when there is the suspect 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 an American organization uses Cloud services, which 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 American government to take control of the data stored in a Cloud Data Center located in Europe.

## 1.2 HISTORICAL DEVELOPMENTS

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

Figure 1.6 provides an overview of the evolution of the technologies for distributed computing 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 are: distributed systems, virtualization, Web 2.0, service-oriented computing computing and utility computing.

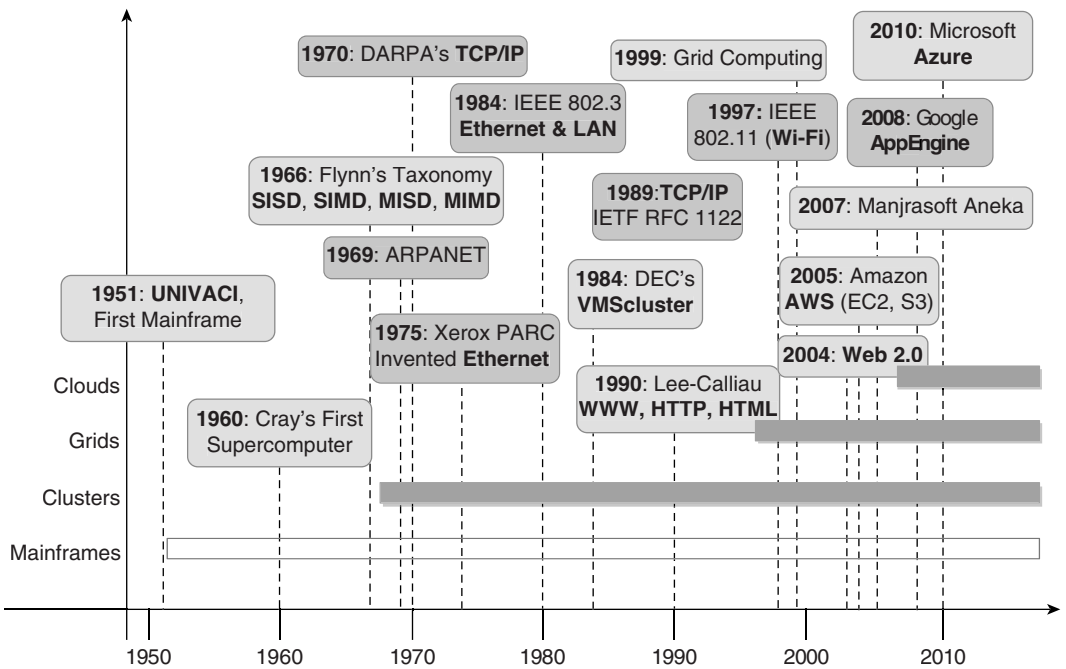


**Fig. 1.6.** Evolution of Distributed Computing Technologies.

## 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."*

This is a general definition, which includes a variety of computer systems but it evidences two very important elements characterizing a distributed system: the fact it is composed of multiple independent components and that these components are perceived as a single entity by users. This is particularly true in case of Cloud computing, where Clouds hide the complex architecture they rely on and provide a single interface to the users. The primary purpose of distributed systems is to share resources and to 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 for Cloud computing has been the availability of large computing facility of IT giants (Amazon, Google, etc.), who found that offering their computing capabilities as a service to be an opportunity for better utilization of 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.

**(a) 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 IO operations. They were mostly used by large organizations for bulk data processing such as online transactions, enterprise resource planning, and other operations involving the processing of significant amount of data. Even though mainframes cannot be considered distributed systems, they were offering 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 shut down was required to replace failed components, and the system could work without interruptions. 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 (i.e., online banking, airline ticket booking, supermarket and telcos, and government services).

**(b) 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 has 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. By starting from the 1980s, clusters became the standard technology for parallel and high-performance computing. Being built by commodity machines, they were cheaper than mainframes, and made available high-performance computing to a large number of groups, including universities and small research labs. Cluster technology considerably contributed to the evolution of tools and framework for distributed computing, some of them include: Condor [5], *Parallel Virtual Machine* (*PVM*) [6], and *Message Passing Interface* (*MPI*)[4] [7]. One of the

[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.

attractive features of clusters was that the computational power of commodity machines could be leveraged to solve problems previously manageable only on expensive supercomputers. Moreover, clusters could be easily extended if more computational power was required.

**(c) Grids.** Grid computing [8] appeared in the early 90s as an evolution of cluster computing. In analogy with 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 aggregation of geographically dispersed clusters by means of Internet connection. 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 reasons made possible the diffusion of computing grids: i) clusters were now resources quite common; ii) they were often under-utilized; iii) new problems were requiring computational power going beyond the capability of single clusters; iv) the improvements in networking and the diffusion of Internet made possible long distance high bandwidth connectivity. All these elements led to the development of grids, which now serve a multitude of users across the world.

Cloud computing is often considered as the successor of Grid computing. In reality, it embodies aspects of all of these three major technologies. Computing Clouds are deployed on 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 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 implement fully 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 named as 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 co-existence of different software stacks on top of the same hardware. These stacks are contained inside *virtual machine instances,* which operate completely isolated from each other. High-performance server can host several virtual machine instances, thus creating the opportunity of having customized software stack on demand. This is the base technology that enables Cloud computing solutions delivering virtual server on demands, 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. In the case of *process virtual machines*, which include the foundation of technologies such as Java or .NET, where applications instead of being executed by the operating system are run by a specific program called *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 the hardware virtualization since the guest is only constituted by an application rather than a complete

software stack. This approach is used in Cloud computing in order 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 an 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 deliver its services. At present time, it encompasses a set of technologies and services that facilitate interactive information sharing, collaboration, user-centered design, and application composition. This has transformed the Web into a rich platform for application development. Such evolution is known as "Web 2.0". This term captures a new way in which developers architect applications, deliver services through the Internet, and provide a new user experience for their users.

Web 2.0 brings *interactivity* and *flexibility* into Web pages, which provide 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 building applications leveraging the contribution of users, who now become providers of content. Also, the capillary diffusion of the Internet opens new opportunities and markets for the Web, whose services can now be accessed from a variety of devices: mobile phones, car dashboards, TV sets, and others. This 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 together, thus providing added value. By doing this, 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 either in terms of media accessibility or cost.

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 from Web 2.0. The level of interaction in Web sites like Facebook or Flickr would not have been possible without the support of AJAX, 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 Website leveraging the user activity for providing content, and Blogger as any other blogging Website provides an online diary that is fed by the 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 fully realize in 2004. Today, it is a mature platform for supporting the need of Cloud computing, which strongly leverages Web 2.0. Applications and frameworks for delivering *Rich Internet Applications (RIAs)* are fundamental for making Cloud services accessible to the wider public. From a social perspective, Web 2.0 applications definitely contributed to make people more accustomed

[5] Darci DiNucci in a column for Design & New Media magazine 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."*

to the use of Internet in their everyday lives, and opened the path to the acceptance of Cloud computing as a paradigm where 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 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: this can be 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 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 for Cloud computing: *Quality of Service (QoS)* and *Software as a Service (SaaS)*.

- Quality of Service identifies a set of functional and non-functional 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 between a Service Level Agreement (SLA) that identifies the minimum values (or an acceptable range) for the QoS attributes that need to be satisfied upon service call.
- The concept of Software as a Service introduces a new delivery model for applications. It has been inherited from the world of Application Service Providers (ASPs). These deliver software services-based solutions across the wide area network from a central data center and make them available on subscription or rental basis. The ASP is responsible for maintaining the infrastructure and making available the application, and the client is freed from maintenance cost and difficult upgrades. This software delivery model is possible because economies of scale are reached by means of multi-tenancy. The SaaS approach reaches its full development with Service Oriented Computing, where loosely coupled software component 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 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 exposing functionalities accessible by using a method invocation pattern that goes over the HTTP protocol. 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 defining how to invoke a Web service method and collect the result. By using SOAP and WSDL over HTTP, Web services become platform independent and accessible as the World Wide Web. The standards and specifications concerning Web services are controlled by the W3C, while 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 integrated with the mainstream World Wide Web (WWW) environment easier.

## 1.2.5 Utility-Oriented Computing

Utility computing is a vision of computing, defining 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 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 mainframes 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 an improvement of 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 the 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 at a world-wide scale and through simple interfaces. As already discussed before, 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]. Being 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 nineties, a significant interest in buying online any kind of good spread in the wide public: food, clothes, multimedia products, and also online services such as storage space and Web hosting. After the *dot-com bubble*[6], this interest reduced in size but the phenomenon made the wide public keener to buy online services. As a result, infrastructures for on-line payment through credit card 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, but 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 on usage. Service-oriented computing 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, Quality of Service became an important topic to investigate on.

[6] The dot-com bubble is a phenomenon that started in the second half of the nineties and reached its acumen in the year 2000. During such period, a large number of companies basing 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 the revenues were not enough to cover the expenses made and partly because they did never reach the required number of customers to sustain their enlarged business.

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

Another class of applications that can potentially gain considerable advantage by leveraging Cloud computing is represented by *resource-intensive applications*. These can be either data-intensive or compute-intensive applications. In both cases, a considerable amount of resources is required to complete execution in a reasonable time frame. It is worth noticing that the large amount of resources is 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 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 mimics the behavior of desktop applications but that are completely hosted and managed on the provider side. All these capabilities leverage service orientation, which allow a simple and seamless integration into existing systems. Developers access such services via simple Web interfaces, often implemented through REST Web services. These have become well-known abstractions, making the development and the 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 in 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 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 system on this scalable infrastructure to make use of such opportunity 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. Beside 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 of service orientation. Despite the absence of a unique standard for accessing the resources serviced by different Cloud providers, the commonality of technology smoothens 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, virtualization is a concept more than 40 years old, but Cloud computing introduces new challenges, especially in the management of virtual environments whether they are abstraction of virtual hardware or of 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 in which 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.

# 1.4 COMPUTING PLATFORMS AND TECHNOLOGIES

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

## 1.4.1 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 of saving a specific running instance as 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 container 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.

## 1.4.2 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 machine by using the AppEngine 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 containing the cost generated, and make it available to the world. The languages currently supported are Python, Java, and Go.

## 1.4.3 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 where 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.

## 1.4.4 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, while the latter aggregates the output obtained by the map operations. Hadoop provides the runtime environment, and developers need only to provide the input data, and specify the map and reduce functions that need to be executed. Yahoo! is the sponsor of the Apache Hadoop project, and has put considerable effort in transforming the project to 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.

## 1.4.5 Force.com and Salesforce.com

*Force.com* is a Cloud computing platform for developing social enterprise applications. The platform is the basis of *SalesForce.com*—a Software-as-a-Service solution for customer relationship management. Force.com allows creating 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.

## 1.4.6 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. These mostly fall into the three major market segments indentified 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 this 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 directly map the broad classification about 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 their revenue. This and other advantages, have also downsides that are of diverse nature. Security and legislation are two of the challenging aspects of Cloud computing that are beyond the technical sphere.

From the perspective of the 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. It 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. In order to provide a practical grasp on 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 that Cloud computing relies on?

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 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 does Cloud development differentiate from traditional software development?

# 2 Principles of Parallel and Distributed Computing

Cloud computing is a new technological trend and it supports better utilization of IT infrastructures, services, and applications. It adopts a service-delivery model based on pay-per-use approach, where users do not own infrastructure, platform, and 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 1940s; parallel (and distributed) computing era followed it within a decade (see Fig. 2.1). The four key elements of computing developed during these eras were: architectures, compilers, applications, and problem-solving environments.

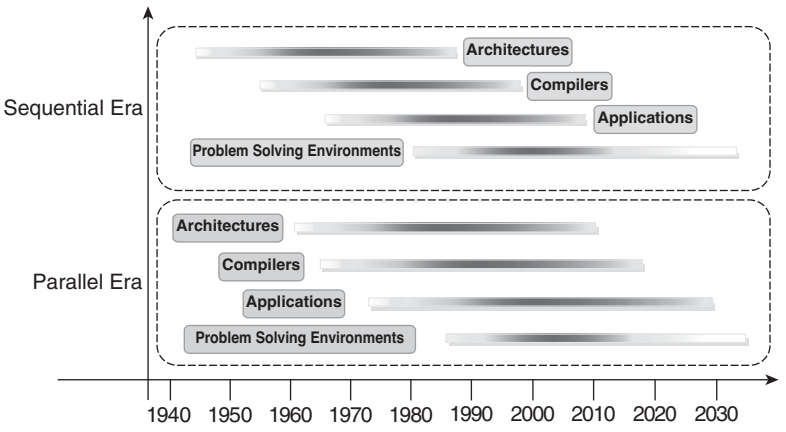


**Fig. 2.1.** Eras of Computing.

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 supported the management of such systems and the development of applications. The development of applications and systems are the major element of interest and it comes to consolidation when problem-solving environments are designed and introduced to facilitate and empower engineers. This is when the paradigm characterizing the computing achieves maturity and becomes a 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"* and "*distributed"* computing are often used interchangeably even though they mean slightly different things. The term "parallel" implies a tightly coupled system while "distributed" refers to a wider class of system including those who are tightly coupled.

More precisely, the term "parallel computing" refers to a model where 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 of 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 executions that can be allocated to different processors, and can communicate with each other by means of the shared memory. Originally, only those architectures were considered as parallel systems which featured multiple processors sharing the same physical memory, and they were considered a single computer. Over time, these restrictions have been relaxed and parallel systems now include all those 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 whose nodes are connected through *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, it includes a wider range of systems and applications with respect to parallel computing and it is often considered a more general term. Even though it is not a rule, the term "distributed" often implies that the location of the computing elements is 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 the 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 number of transistors package density in 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 towards 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 it is often used in place of that. 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 divide-and-conquer technique, and each one of them is processed on different CPUs. Programming on multi–processor system using *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 now be shared between different processors. This results in higher computing power and performance than a single processor system.

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

- computational requirements are ever increasing, both in the area of 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, etc.
- sequential architectures are reaching physical limitation as they are constrained by the speed of light and thermodynamics laws. Speed with 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, etc., are non-scalable and require sophisticated compiler technology. Developing such compiler technology is a difficult task.
- vector processing works well for certain kind of problems. It is suitable mostly for scientific problems (involving lots of matrix operations) and graphical processing. It is not useful to other areas such as database.
- the technology of parallel processing is mature and can be exploited commercially; there is already significant research and development (R & D) work on development tools and environment.
- significant development in networking technology is paving a way for heterogeneous computing.

## 2.3.2 Hardware Architectures for Parallel Processing

The core elements of parallel processing are CPUs. Based on a 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)*
- *Single Instruction Multiple Data (SIMD)*
- *Multiple Instruction Single Data (MISD)*
- *Multiple Instruction Multiple Data (MIMD)*

### *1. Single Instruction Single Data (SISD)*

A SISD computing system is a uniprocessor machine capable of executing a single instruction, which operates on a single data stream (see Fig. 2.2). In SISD, machine instructions are processed sequentially, and hence computers adopting this model are popularly called sequential computers.

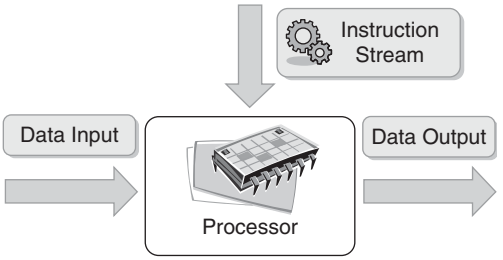


**Fig. 2.2.** Single Instruction Single Data (SISD) Architecture.

Most of the conventional computers are built using SISD model. All the instructions and data to be processed have to be stored in the primary memory. The speed of processing element in SISD model is limited by the rate at which the computer can transfer information internally. Dominant representative SISD systems are IBM-PC, Macintosh, Workstations, etc.

## 2. Single Instruction Multiple Data (SIMD)

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

$$Ci = Ai * Bi$$

can be passed to all the PEs (processing elements); organized data elements of vector 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, Thinking Machines's cm*, etc.

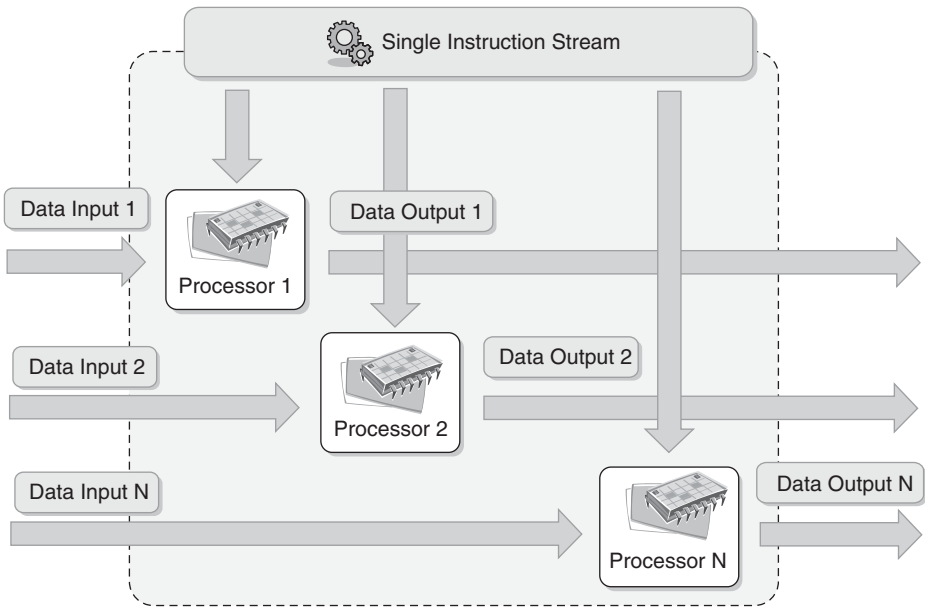


**Fig. 2.3.** Single Instruction Multiple Data (SIMD) Architecture.

## 3. Multiple Instruction Single Data (MISD)

A 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 Fig. 2.4). For instance, statements such as

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

perform different operations on the same data set. Machines built using MISD model, are not useful in fmost 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.

## 4. Multiple Instruction Multiple Data (MIMD)

A MIMD computing system is a multiprocessor machine capable of executing multiple instructions on multiple data sets (see Fig. 2.5). Each PE in the MIMD model has separate instructions and data

streams, and hence machines built using this model are well suited for 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 how PEs are coupled to the main memory.

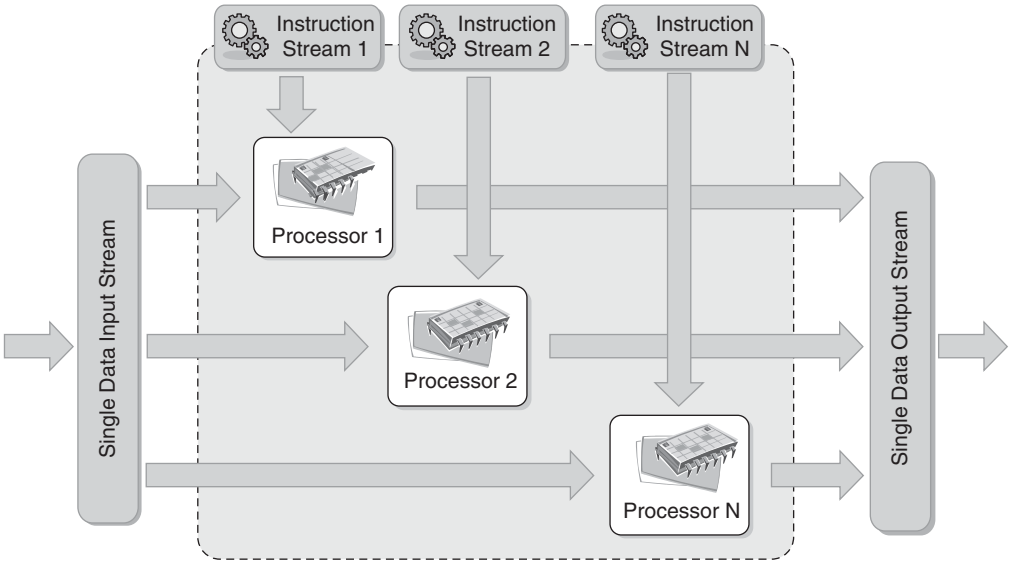


**Fig. 2.4.** Multiple Instruction Single Data (MISD) Architecture.

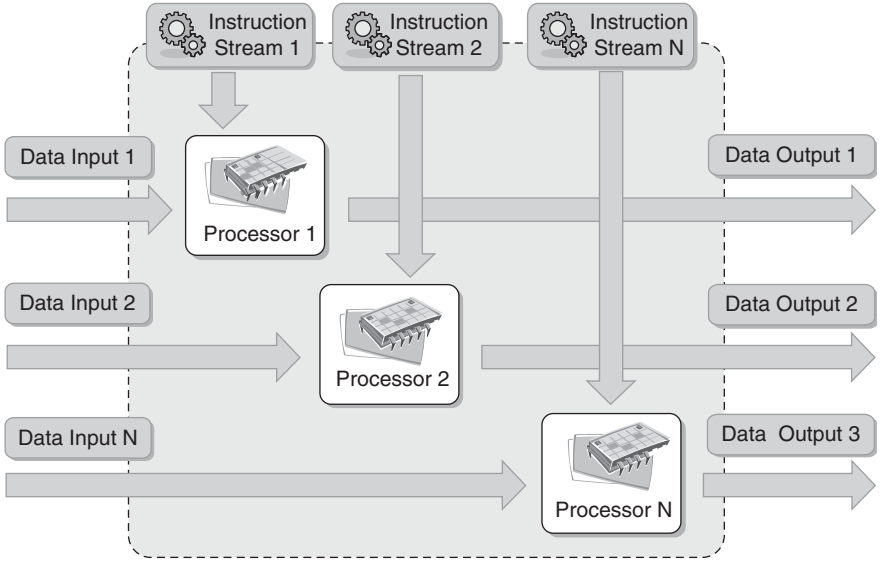


**Fig. 2.5.** Multiple Instructions Multiple Data (MIMD) Architecture.

**(a) Shared Memory MIMD Machine.** In the shared memory MIMD model, all the PEs are connected to a single global memory and they all have access to it (see Fig. 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 SMPs (Symmetric Multi-Processing).

**(b) Distributed Memory MIMD Machine.** In 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 (IPC-inter-process communication channel). The network connecting PEs can be configured to tree, mesh, cube, etc. Each PE operates asynchronously and if communication/synchronization among tasks is necessary, they can do so by exchanging messages between them.

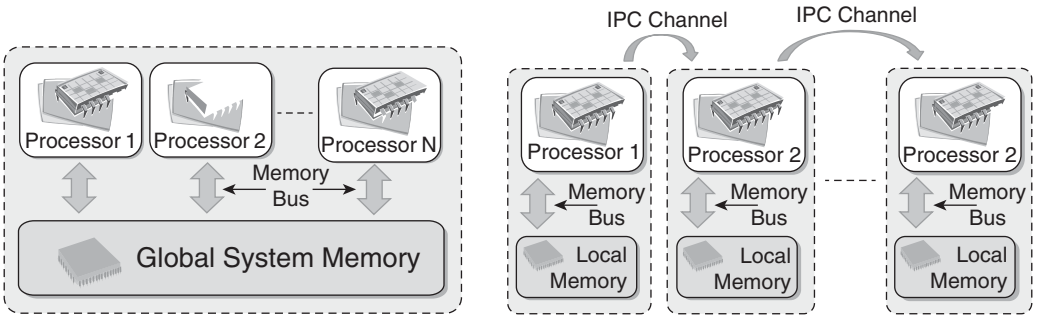


**Fig. 2.6.** Shared (left) and Distributed (right) Memory MIMD Architecture.

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 where each of the PEs can be easily isolated. Moreover, shared-memory MIMD architectures are less lightly to scale because the addition of more PEs leads to memory contention. This is a situation that does not happen in case of distributed memory where each of the PEs has its own memory. As a result, distributed memory MIMD architectures are most popular nowadays.

## 2.3.3 Approaches to Parallel Programming

A sequential program is one which 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*

All these three models are suitable for task-level parallelism. In case of data parallelism, divide-and-conquer technique is used to split data into multiple sets, and each data set is processed on different PEs by using the same instruction. This approach is highly suitable for processing on machines based on the SIMD model. In the case of process parallelism, a given operation has multiple (but distinct) activities, which can be processed on multiple processors. In case of 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 a job to slave PEs, and they on completion inform the master which in turn collects the results. The above 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.

**Table 2.1.** Levels of Parallelism

| *Grain Size* | *Code Item* | *Parallelized by* |
|---|---|---|
| *Large* | Separate and heavy-weight process | Programmer |
| *Medium* | Function or procedure | Programmer |
| *Fine* | Loop or instruction block | Paral lelizing Compiler |
| *Very Fine* | Instruction | Processor |

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

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

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.

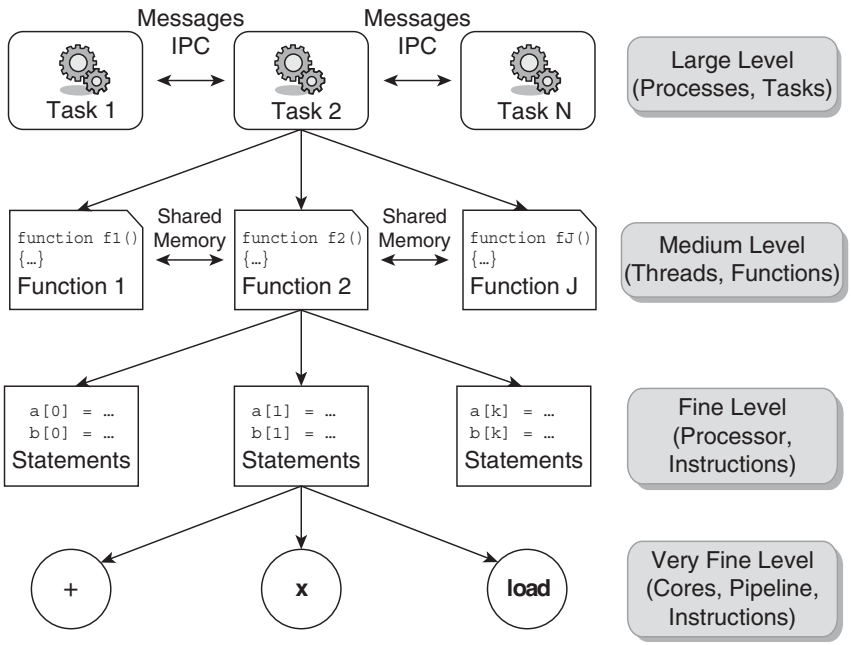


**Fig. 2.7.** Levels of Parallelism in an Application.

### 2.3.5 Laws of Caution

After having introduced some general aspects of parallel computing in terms of architectures and models, we can make some considerations that have been drawn from the experience in designing and implementing such systems. These considerations are guidelines that can help in understanding how much benefit an application or a software system can have 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. It is an ideal situation, which 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, faster the system, more the expense to increase its speed (Fig. 2.8).
- Speed-up by a parallel computer increases as the logarithm of the number of processors; (i.e., $y = k*log(N)$). It is shown in Fig. 2.9.

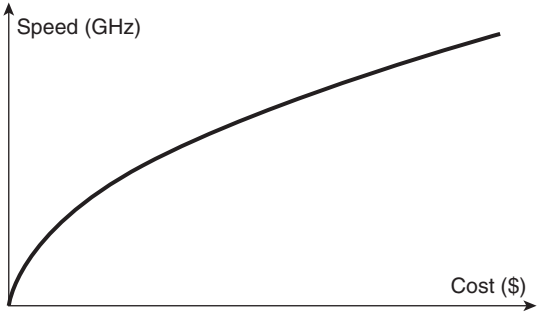


**Fig. 2.8.** Cost versus Speed.

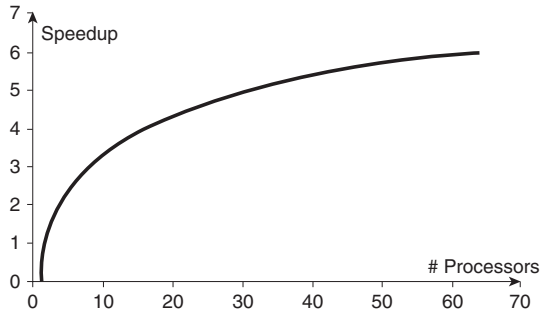


**Fig. 2.9.** Number of Processors versus Speed-up.

The very fast development in parallel processing and related areas has blurred the conceptual boundaries, causing lot of terminological confusion. Even well-defined distinctions like 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, all—computer architects, OS designers, language designers and computer network designers—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 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, the architectures, and the algorithms used for building and managing distributed systems. As a general definition of 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 by 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 the above 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 to the users the illusion of a single coherent system. Fig. 2.10 provides an overview of the different layers that are involved in providing the services 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 that provides the basic services for: inter-process communication, 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.

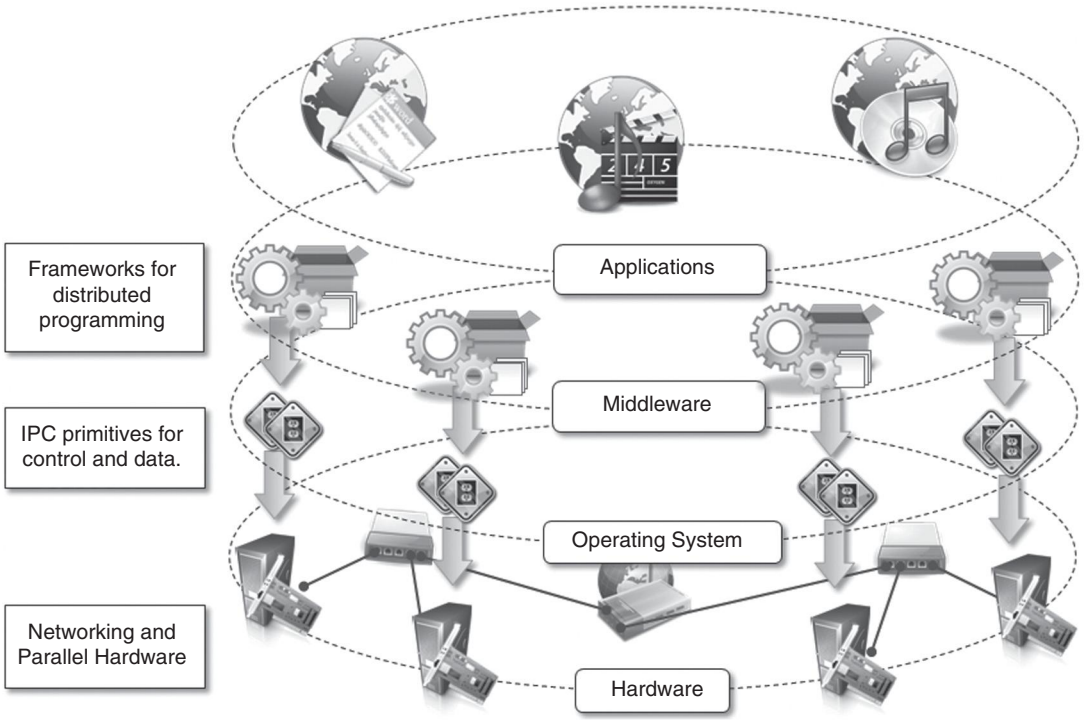


**Fig. 2.10.** Layered View of a Distributed System.

The use of well-known standards at the operating system, 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, inter-process communication services are implemented on top of standardized communication protocols such as TCP/IP and UDP.

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. 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 applications developers that is completely independent from the underlying operating system and hides all the heterogeneities of the bottom layers.

The top of the distributed system stack is represented by the applications and services designed and developed to use the middleware. These can serve several purposes and often expose their features in the form of graphical user interfaces accessible locally or through the Internet via a Web browser. For example, in the case of 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 provides facilities for creating virtual machines, organizing them together into a cluster, and deploying applications and systems on top of it. Fig. 2.11 shows an example on how the general reference architecture of a distributed system is contextualized in the case of Cloud-computing system.

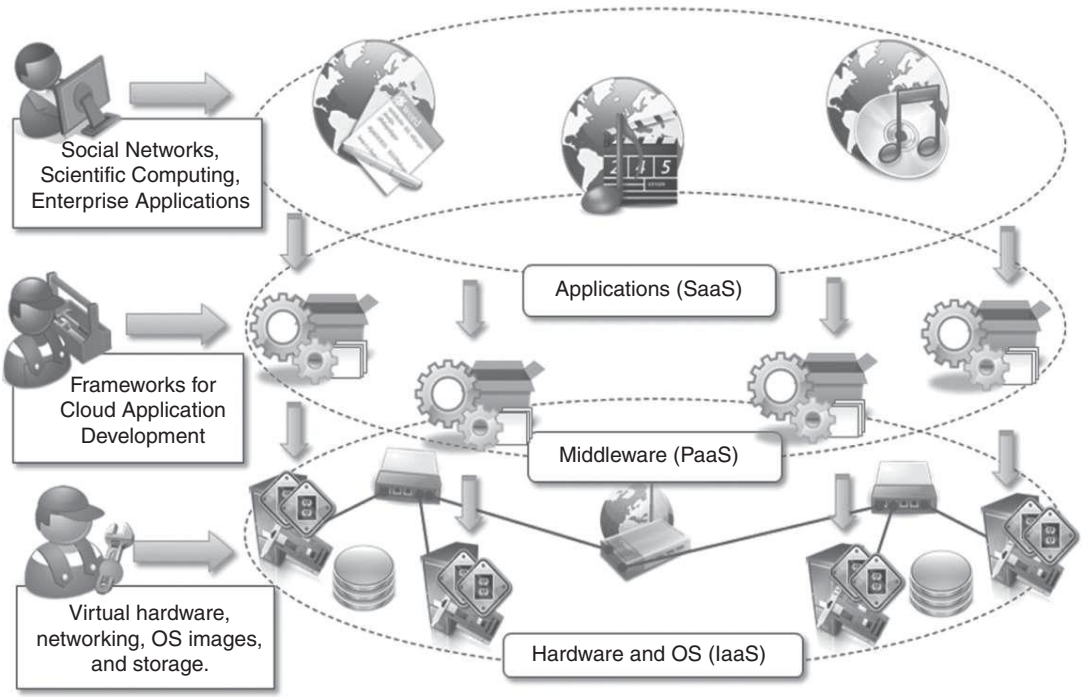


**Fig. 2.11.** Cloud-Computing Distributed System.

It can be noticed that hardware and operating system layer 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 that is deployed on the physical infrastructure in order to maximize its utilization and to 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 are offered through Web 2.0 compliant interfaces and 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 of interaction of several layers, the middleware layer is the one that enables distributed computing, as it provides a coherent and uniform runtime environment for applications. There are many different ways in which it is possible 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 their architecture. Architectural styles [104] help 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 on 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, while the second class includes all those styles that describe the physical organization of distributed software systems in terms of their major components.

### *1. Components and Connectors*

Before discussing the architectural styles in detail, it is important to build an appropriate vocabulary. 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 the 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. 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 what are the main abstractions that are used to shape the components of the system, and what are the expected interaction patterns between them. According to Garlan and Shaw [105], architectural styles are classified as shown in Table 2.2.

**Table 2.2.** Architectural Styles.

| *Category* | *Most Common Architectural Styles* |
|---|---|
| *Data Centered* | 1. Repository<br>2. Blackboard |
| *Data Flow* | 1. Pipe and Filter<br>2. Batch Sequential |
| *Virtual Machine* | 1. Rule-based System<br>2. Interpreter |
| *Call & Return* | 1. Main Program and Subroutine Call / Top Down Systems<br>2. Object Oriented Systems<br>3. Layered Systems |
| *Independent Components* | 1. Communicating Processes<br>2. Event Systems |

These models constitute the foundations on top of which distributed systems are designed from a logical point of view and they are discussed below:

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

The **Repository** architectural style is the most relevant reference model in this category. It is characterized by two main components: the central data structure, which represents the current state of the system, and a collection of independent components, which operate on the central data. The ways in which the independent components interact with the central data structure can be very heterogeneous. In particular, repository-based architectures differentiate and specialize further into sub categories 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 operation on the central repository trigger the selection of specific processes that operate on data. In blackboard systems, instead, 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 as 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 where the blackboard maintains the knowledge about a domain in the form of assertion 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 domain of speech recognition and signal processing.

**(b) 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, where 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: 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 sequence of programs that for example: pre-filter, analyze, and post-process data. It is very common to compose these phases by 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 *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 starts its processing. Filters, generally, do not state neither know the identity of the previous or the next filter, and they are connected within memory data structures such as 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 where 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, while the data streams are represented by the registries that are shared within the processors. Another example are the Unix shell pipes (i.e., *cat <file-name> | grep <pattenr> | 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 compilers design (e.g. the lex/yacc model is based on a pipe of the following phases: *scanning | parsing | semantic analysis |* ), image and signal processing, and voice and video streaming.

Data-flow architectures are optimal when the system to be designed embodies a multi-stage 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 data streams between these ports. The main differences between the two sub-categories are reported in Table 2.3.

**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 |
| Non-interactive | Interactive awkward but possible |

**(c) Virtual Machine Architectures.** This class of architectural styles is characterized by the presence of an abstract execution environment (generally referred as *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 environment as long as there is an implementation of the virtual machine they interface with. The general interaction flow for systems implementing this pattern is the following: the program (or the application) defines its operations and state in an abstract format which is interpreted by the virtual machine engine. The interpretation of a program constitutes its execution. It is quite common in this scenario that the engine maintains an internal representation of the program state. Very popular examples within this category are: rule-based systems, interpreters, and command language processors.

*Rule-Based Style.* This architecture is characterized by representing the abstract execution environment as an *inference engine*. Programs are expressed in the form of rules or predicates that hold true. The input data for applications is generally represented by a set of assertions or facts that the inference engine uses to activate rules or to apply predicates, thus transforming data. The output can either be the product of the rules activation or a set of assertions that holds true for the given input data. The set of rules or predicates identify 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[7], 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 machine for high-level programming (Java, C#) and scripting languages (Awk, PERL, etc.). Within this scenario, the virtual machine closes the gap between the end user abstractions and the software/hardware environment where 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 introduces some disadvantages such as slow-down 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.

**(d) Call and Return Architectures.** This category identifies all those systems that are composed by 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 sub-categories, which differentiate by how the system is structured and how methods are invoked.

[7] A PLC (Programmable Logic Controller) 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 usage in factory environments. PLCs are an example of hard real-time systems since they are expected to produce the output within a given time interval since the reception of the input.

*Top-Down Style.* This architectural style is quite representative for systems developed with imperative programming, which leads to a divide-and-conquer approach for problem resolution. Systems developed according to this style are composed of one large main program that accomplishes its tasks by invoking sub-programs or procedures. The components in this style are procedures and sub-programs, 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 RPC and all its descendants. The overall structure of the program execution at any point in time is characterized by a tree whose root is constituted by 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 case of 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. 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 this data. One of the main advantages over the previous 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 the 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 is that, as it happens for the object-oriented style, it supports a modular design of system and in addition, it allows 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 an additional layer without changing the protocols and the interfaces between layers[8]. This also makes it complex to add additional operations. Examples of layered architectures are the modern operating systems kernels and the ISO/OSI or the TCP/IP stack.

**(e) Architectural Styles based on Independent Components.** This class of architectural styles model systems in terms of independent components, having their own life cycle, which interact to each other in order to perform their activities. There are two major categories within this class, which differentiate in the way the interaction among components is managed.

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

*Communicating Processes.* In this architectural style, components are represented by independent processes that leverage inter-process communication (IPC) facilities for coordination management. This is an abstraction that is quite suitable to model distributed systems that, being distributed over a network of computing nodes, are necessarily composed by 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: peer-to-peer or client-server[9]. 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 operation for data and state manipulation, each component also publishes (or announces) a collection of events that other components can register with. 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 API level or at programming language level[10]. The main advantage of such architectural styles 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 the object-oriented style. 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 handler to events can be done without changes in the source code of applications. Secondly, the event source does not need to know the identity of the event handler in order to invoke the callback. The disadvantage of such 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 if there is any registered handler. 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 and other styles can be found in [105].

# 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 in having a common vocabulary in describing the physical layout of systems but also in quickly identifying 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*.

**(a) Client-Server.** This architecture is very popular in distributed computing and it is suitable for a wide variety of applications. As depicted in Fig. 2.12, the client-server model features two major components: a server and a client. These two components interact with each other through a

[9] The terms “client-server” and “peer-to-peer” will be further discussed in the next section.

[10] The *Observer* pattern [106] is a fundamental element of software designs, while 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.

network connection by using a given protocol. The communication is unidirectional: the client issues a request to the server, and server after processing the request returns a response. There could be multiple client components issuing requests to the 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).

This 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 server requests coming from different clients efficiently. This consideration has implications on both client design and server design. For the client design, we identify two major models:

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

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

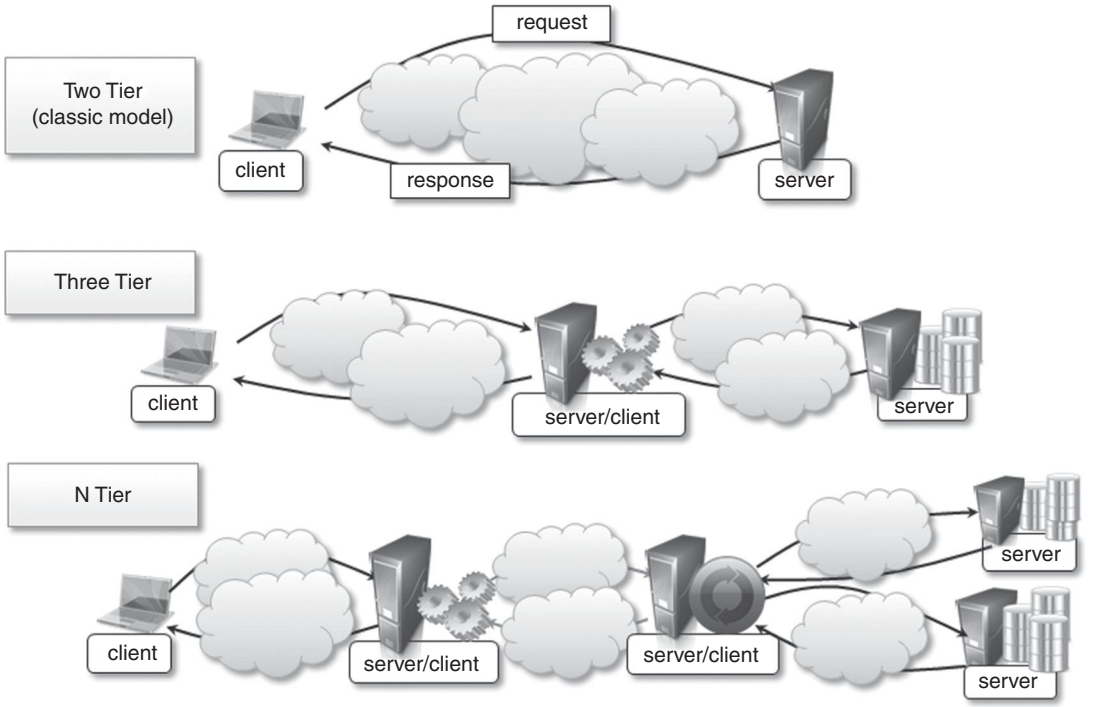


**Fig. 2.12.** Client-Server Architectural Styles.

We can identify three major components in the client-server model: presentation, application logic, and data storage. With respect to the previous classification, we can observe that in the thin-client model, the client embodies only the presentation component, while the server absorbs the other two, while 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 *multi-tiered 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, while 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 previous one because it is possible to distribute the tiers into several computing nodes, thus isolating the performance bottlenecks. At the same time, they 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. Within this scenario, the client component is represented by a Web browser that embodies the presentation tier, while 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 in order to satisfy the requests of clients 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 case of a many-to-one scenario, where the interaction is unidirectional and started by the clients. This model suffers from scalability issues, therefore it is not appropriate in very large systems.

**(b) Peer-to-Peer.** This model introduces a symmetric architecture where all the components, called peers, play the same role and incorporate both the client and server capabilities of the previous 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 which partitions the responsibilities of the inter-process communication 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 if compared to the client-server model (Fig. 2.13).

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 that each of these networks have in coordinating the nodes and sharing the information of the files and their locations, all of them provide a user client which is at the same time a server (providing files to other peers) and a client (downloading files from other peers). In order to address an incredibly large number of peers, different architectures

have been designed, which 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.

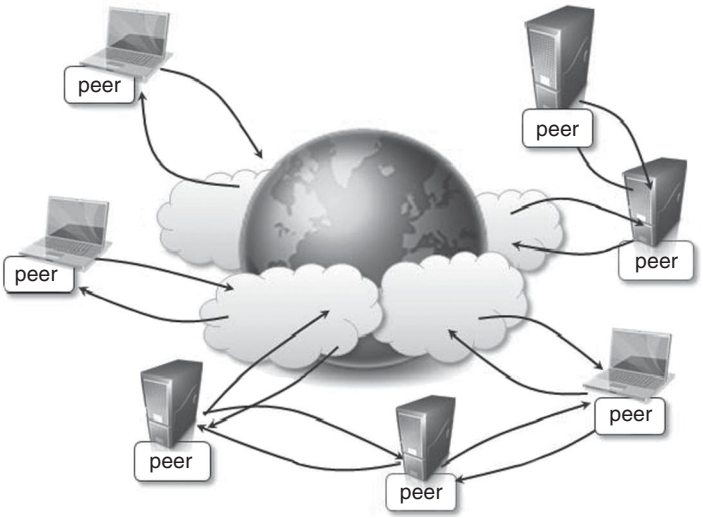


**Fig. 2.13.** Peer-to-Peer Architectural Style.

The system architectural styles presented in this section constitute a reference model that are 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 multi-tier 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 Inter-Process Communication

Distributed systems are composed of a collection of concurrent processes interacting with each other by means of a network connection. Therefore, inter-process communication (IPC) is a fundamental aspect of distributed systems design and implementation. IPC is used to either exchange data and information or to coordinate the activity of processes. It is what ties together the different components of a distributed system, thus making them acting as a single system. There are several different models in which processes can interact with each other, and these map to different abstractions for IPC; among the most relevant we can mention: 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 inter-process communication. In this section, we introduce the most important reference model for architecting the communication among processes.

# 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. [1] 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 inter-process communication 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 abstraction that are presented to developers for programming the interaction of distributed components. Here are some of the most popular and important:

**(a) Message Passing.** This paradigm introduces the concept of message as the main abstraction of the model. The entities exchanging information explicitly encode, in the form of message, the data to be exchanged. The structure and the content of a message vary according to the model. Examples of this model are *Message Passing Interface* (*MPI*) and *OpenMP*.

**(b) 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 also the return values. The use of messages within this context is also referred as marshaling of parameters and return values.

**(c) 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*.

**(d) 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.

**(e) Web Services.** Web service technology provides an implementation of the RPC concept over the HTTP transport protocol, thus allowing the interaction of components that are developed with different technologies. A Web service is exposed as a remote object hosted in a Web server, and method invocations are transformed in HTTP requests opportunely packaged by using specific protocol such as *SOAP* (*Simple Object Access Protocol*) or *REST* (*REpresentational State Transfer*).

It is important to observe that the concept of message is a fundamental abstraction of inter-process communication and it is used either explicitly or implicitly. Their principal use—in any of the cases discussed—is to define interaction protocols among distributed components for coordinating their activity and exchange data.

## 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 how 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, identifies 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.

**(a) 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 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 sender of the message. It is possible to identify two major sub-categories: 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 where the messages received are placed for later processing. The point-to-point message model is useful for implementing systems mostly based on one-to-one or many-to-one communication.

**(b) Publish-Subscribe Message Model.** This model introduces a different strategy which 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 which are attached to a specific event. This 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 the responsibility of the subscribers to check whether there are messages on the events that are registered.

The publish-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 subscriber to make the communication happen.

**(c) Request-Reply Message Model.** The request-reply message model identifies all those communication models, where for each of the 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 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-subscribe models are less likely to be based on request-reply since they rely on notifications.

The models that have been presented constitute a reference model 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 them 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 objects frameworks, and service-oriented computing.

## 2.5.1 Remote Procedure Call

Remote Procedure Call (RPC) is the fundamental abstraction enabling the execution of procedures on client's request. It allows extending the concept of 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 the 1976, and completely formalized by Nelson [111] and Birrell [112] in the early eighties. From there on, it has not changed in its major components. Even though it is a quite old technology, it is still used today as a fundamental component for inter-process communication in more complex systems.

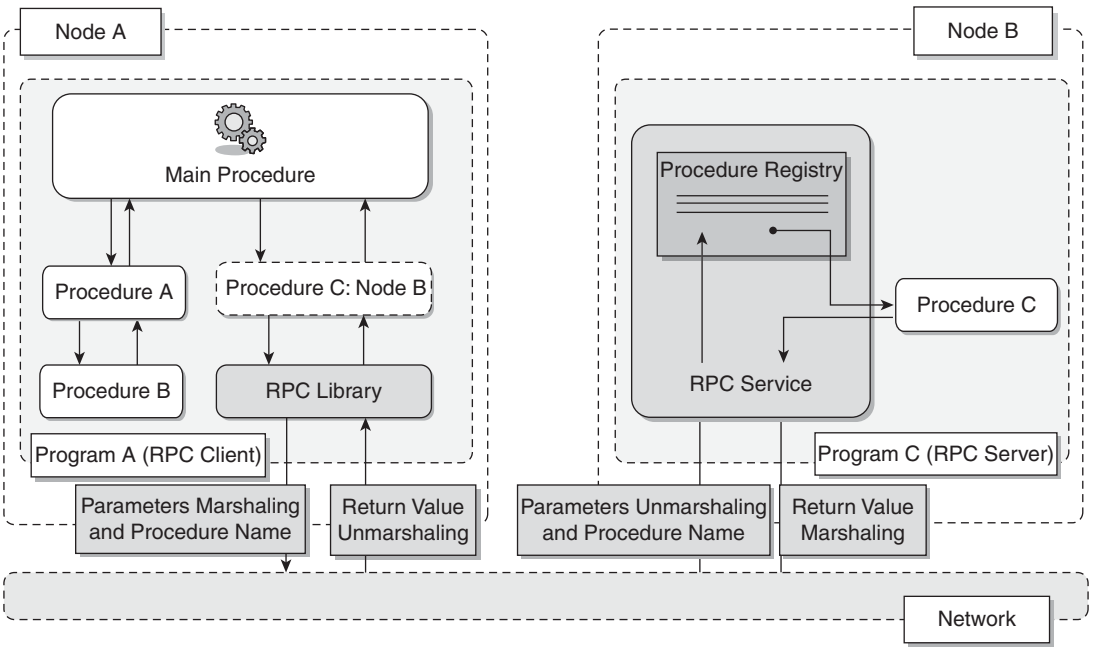


**Fig. 2.14.** RPC Reference Model.

Fig. 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 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 in-process procedure 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 parameters packing and unpacking but also for handling the request-reply interaction that happens between the client and the server process in a complete transparent manner. Therefore, developing a system leveraging RPC for inter-process communication consists of the following steps:

- Design and implementation of the server procedures that will be exposed for remote invocation.
- Registration of remote procedure 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.

Each RPC implementation generally provides client and server APIs that facilitate the use of this simple and powerful abstraction. An important observation has to be made concerning the passing of parameters and of return values. Being the server and the client process in two separate address spaces, the use of parameters passed by reference or pointers is not suitable in this scenario, since once unmarshaled these will refer a memory location that is not accessible from within the server process. Secondly, in case of 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 by simple types, for which marshaling is naturally provided.

RPC has been a dominant technology for inter-process communication 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, are considered RPC implementations even frameworks that evolved this concept towards more powerful abstractions such as frameworks for distributed object programming (CORBA, DCOM, Java RMI, and .NET Remoting) and Web Services. We discuss the peculiarity of these approaches in the following sections.

## 2.5.2 Distributed Object Frameworks

Distributed object frameworks extend the object-oriented programming systems by allowing objects to be distributed across a heterogeneous network and provide facilities so that they can coherently act as if 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:

- 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.
- 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.
- 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 case of RPC.

Distributed objects frameworks give the illusion of interaction with a local instance while invoking remote methods. This is done by a mechanism called *proxy-skeleton*. Fig. 2.15 gives an overall 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 remotely invoked, while the clients maintain the proxy component allowing its hosting environment to remotely invoke methods through the proxy interface. The transparency of remote method invocation is achieved by using one of the fundamental properties of object-oriented programming: inheritance and sub-classing. 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 run-time object sub-classing the type published by the server is generated. This object translates the local method invocation into an RPC call for the corresponding method on the remote active object. On the server side, whenever a RPC request is received, this 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, sent back to the client, and the local method call on the proxy returns.

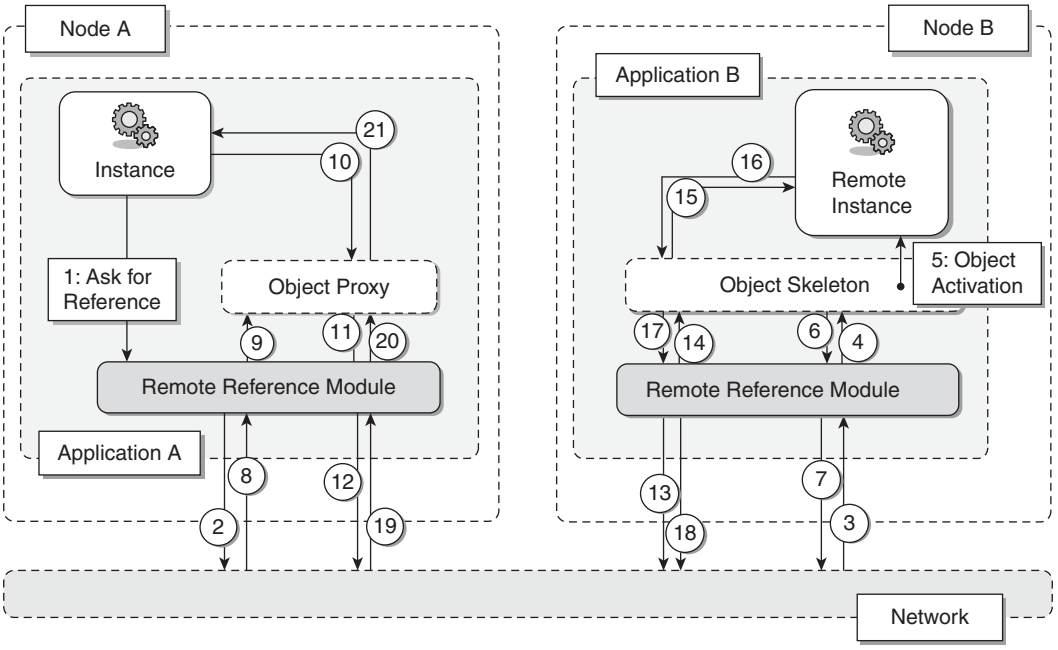


**Fig. 2.15.** Distributed Object Programming Model.

Distributed objects frameworks introduce objects as first-class entities for inter-process communication. They are the principal gateways 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 when designing distributed objects 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 tech-

nique 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.

## 1. Objects Activation and Lifetime

The management of distributed objects poses additional challenges with respect to the simple invocation of 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. There are different strategies that can be used to manage object activation and 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 and 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 case of client-based activation, the creation of the remote object is implicit, and therefore its lifetime is controlled by some policy by 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 it 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 only having a single instance at a time, and the lifetime of the object is then controlled by the number and the frequency of method calls. Different frameworks provide different level of controls on 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 for understanding the behavior of a distributed system. In particular, these two aspects are becoming fundamental while 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. Examples of Distributed Object Frameworks

The support for distributed object programming has evolved over time, and nowadays, 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 has been designed as a separate component that could be used in several programming languages. In this section, we briefly review the most relevant approaches and technologies for distributed objects programming.

**(a) 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 has been originally designed to provide an interoperation standard that could be effectively used at 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. In order to simplify cross-platform interoperability, interfaces are defined in *IDL (Interface Definition Language)*, 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 environments by simply using the stub and the skeleton that match the development language used. Being a specification meant to be used at an industry level, CORBA provides interoperability among different implementations of its runtime. In particular, at the lowest levels, ORB implementations communicate with each other by using the *Internet Inter-ORB Protocol (IIOP)*, which standardize the interactions by 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 where the skeletons are hosted and managed. Again, the interface among these two layers is clearly defined, thus giving more freedom and allowing different implementations to work together seamlessly.

**(b) 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, and it has been designed to be easily plugged into another application to leverage the features exposed through its interface. In order 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 interprocess communication support. The architecture of DCOM is quite similar to CORBA but simpler, since it does not aim to foster the same level of interoperability, being its implementation monopolized by Microsoft, which provides a single runtime environment. A DCOM server object can expose several interfaces, each of them representing a different behavior of the object. In order to invoke the methods exposed by the interface, clients obtain a pointer to that interface and use it as if it was 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 exist third-party implementations that allow the use of DCOM even in Unix-based environments. Currently, even if still used in industry, this technology is not popular anymore since it has been replaced by other approaches such as .NET Remoting and Web Services.

**(c) 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 a different Java Virtual Machine (JVM) residing either on the local node or on a remote one. As it happens for CORBA, RMI is based on the *stub-skeleton* concept. Developers define an interface extending *java.rmi.Remote* that defines the contract for inter-process communication. 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 by 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 if it was a local instance, and RMI performs all the required operations to enable the inter-process communication. 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.

**(d) .NET Remoting.** Remoting is the technology allowing for inter-process communication 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 objects technologies, remoting is a fully customizable architecture that allows the developer 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, it 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 at least have one.

Remoting allows objects located in different application domains to interact in a completely transparent manner, despite the two domains being in the same process, in the same machine, or in different nodes. The reference architecture is based on classic client-server model where 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*— this is 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 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 client application domain. In order to make a component accessible through remoting, it needs to be registered with the remoting runtime and mapped to a specific URI in the form— *scheme://host:port/ServiceName*— where the 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*. In order 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 enabling distributed objects programming. CORBA is an industrial standard technology for developing distributed systems spanning across different platform and vendors. The technology has been designed to be interoperable among different implementation and languages. Java RMI and .NET Remoting are built-in infrastructures for inter-process communication, 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, 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 developer 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 aggregation 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.

### *1. What is Service?*

A service encapsulates a software component providing 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 by using different technologies and protocols. Don Box [107] identifies four major characteristics that identify a service:

**(a) Boundaries are explicit.** A service-oriented application is generally composed by services that are spread across different domains, trust authorities, and execution environments. Generally, crossing such boundaries is costly, and therefore service invocation is explicit by design, and often leverages message passing. With respect to distributed objects programming where 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.

**(b) Services are autonomous.** Services are components that exist to offer functionality and are aggregated and coordinated to build more complex systems. 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 consider 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 how services handle failures. Services operate in an unknown environment and interact with third-party applications. Therefore, minimal assumptions can be made concerning such an environment: 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 manages trust relationship among different domains.

**(c) Services share schema and contract, not the class or interface definition.** 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 a wider and heterogeneous environment. At the same time, services orientation requires that contract and schema remain stable over time, since it would be possible to propagate changes to all its possible clients. To address this issue, contract and schema definition 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 feature, rather than class definition or an interface declaration.

**(d) Services compatibility is determined based on policy.** Service orientation separates structural compatibility from semantic compatibility. Structural compatibility is based on contract 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 the requirement for a service. Policies are organized in terms of expressions that must hold true in order to enable the normal operation of a service.

Services constitute today 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 for integration. Also, being designed as autonomous components, they can be more easily reused and aggregated. These features are not carved out from a smart system design and implementation—as happens in the case of distributed objects programming—but are part of the service characterization.

## 2. *Service Oriented Architecture (SOA)*

SOA [20] is an architectural style supporting service orientation[11]. 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. A 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. In order to advertise services, the provider can publish them into 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 charging fees. The service consumer can locate the service metadata in the registry, and develops 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 both play the role of service providers and service consumers. 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 single point of control.

SOA provides a reference model for architecting several software systems especially enterprise business applications and systems. Within 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:

**(a) Standardized Service Contract.** Services adhere to a given communication agreement, which is specified through one or more service description documents.

**(b) Loose Coupling.** Services are designed as self-contained components, maintain relationship that minimizes dependencies on other services, and only require being aware of each other.

[11] This definition is given by the Open Group (***http://www.opengroup.org***), which is a vendor and technology neutral consortium including over three hundred 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, being 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: ***http://www.opengroup.org/soa/soa/def.htm***.

Service contracts will enforce the required interaction among services. This simplifies the flexible aggregation of services and enables a more agile design strategy supporting the evolution of the enterprise business.

**(c) Abstraction.** A service is completely defined by service contracts and description documents. They hide their logic, which is encapsulated within their implementation. The use of service description documents and contracts removes the need to consider the technical implementation details, and provides a more intuitive framework to define software systems within a business context.

**(d) Reusability.** Being designed as components, services can be reused more effectively, thus reducing the development time and the associated costs. It 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 than developing in house the same capability.

**(e) 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.

**(f) 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 where a single service is used by multiple consumers belonging to different administrative and business domains.

**(g) 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.

**(h) Composability.** By 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 the business goals.

Together with these principles, other resources guide the use of SOA for *Enterprise Application Integration (EAI)*. The SOA manifesto[12] integrates the principles previously described with general considerations about the overall goals of a service-oriented approach to enterprise application software design and what is valued in SOA. Also, 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.

## 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 of all, they allow for interoperability across different platforms and programming languages. Secondly, they are based on well-known and vendor-independent standards such as HTTP, SOAP [23], XML, and WSDL [22]. Thirdly, they provide

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

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

The concept behind a Web service is very simple. By 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 by interacting with Web Services. Fig. 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 a compatible Web or application server. The service description document, expressed by means of 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 by using UDDI or, most likely, directly retrieve the service metadata by interrogating the Web Service first. The Web service description document allows the service consumer to automatically generate clients for the given service and embed them in their existing application. Web services are now extremely popular, thus bindings exist for any mainstream programming language in the form of libraries or development support tools. This makes the use of Web service seamless and straightforward, with respect to technologies such as CORBA that require much more integration effort. Moreover, being interoperable, they constitute a better solution for SOA with respect to several distributed objects frameworks, such as .NET Remoting, Java RMI, and DCOM/COM+, which limit their applicability to a single platform or environment.

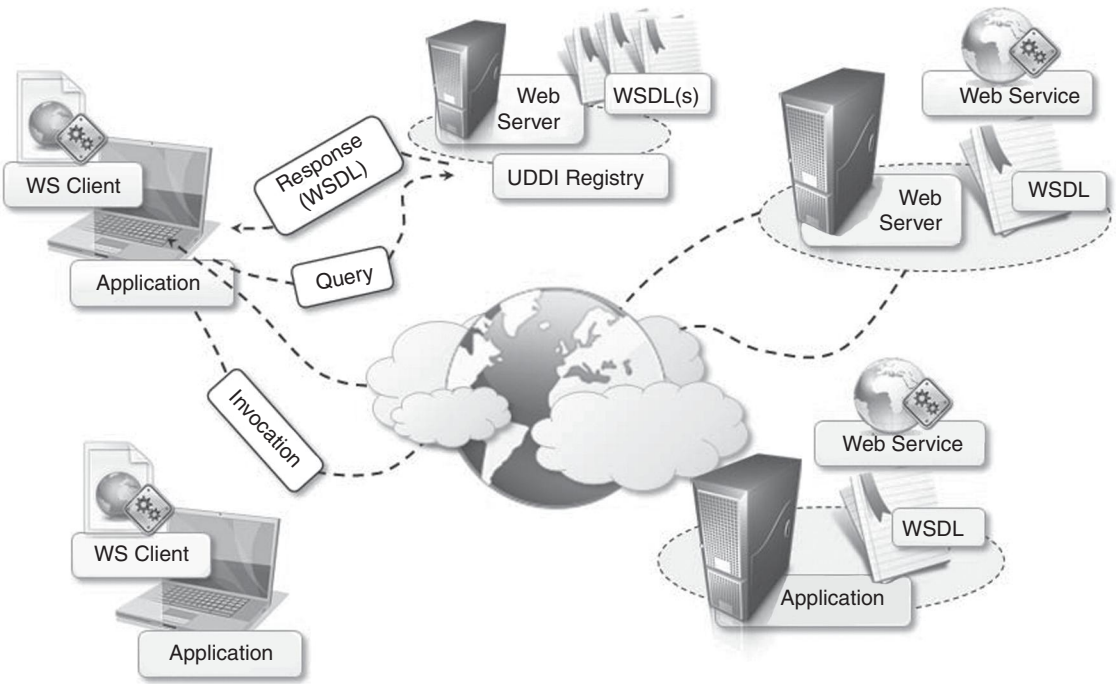


**Fig. 2.16.** Web Services Interaction Reference Scenario.

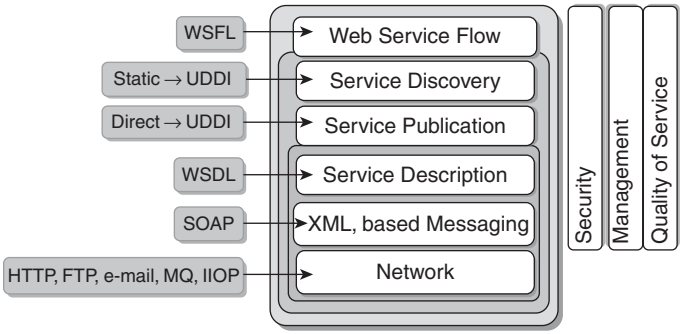


**Fig. 2.17.** Web Services Technologies Stack.

Besides the main function of enabling remote method invocation by using Web based and interoperable standards, Web Services encompass several technologies that when put together can facilitate the integration of heterogeneous applications and enable service-oriented computing. Fig. 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] is an XML-based language for exchanging structured information in a platform independent manner and 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 inter-process communication. 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 also 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 use of SOAP message is method invocation and result retrieval. Fig. 2.18 shows an example of a SOAP message used to invoke a Web-service method that retrieves the price of a 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 mark-up 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)*. REST provides a model for designing network based software systems utilizing the client-server model and leverages the facilities provided by HTTP for inter-process communication without additional burden.

In a *RESTful* system, a client sends a request over HTTP by 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 is 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 by using XML as part of the HTTP content, but the additional mark-up 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 within 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* leverage REST.

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

<?xml version="1.0">
                                                                    Envelope
<soap:Envelope xmlns:soap="http//www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http//www.w3.org/2001/12/soap-encoding">
                                                    Header:Metadata & Assertions
 <soap:Header></soap:Header>

 <soap:Body xmlns:m=http://www.stocks.org/stock>
  <m:GetStockPrice>
   <m:StockName> IBM <m:StockName>
  </m:GetStockPrice>
 </soap:Body>
                                                         Body:Method Call
  </soap:Envelope>
```

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

<?xml version="1.0">
                                                                    Envelope
<soap:Envelope xmlns:soap="http//www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http//www.w3.org/2001/12/soap-encoding">
                                                    Header:Metadata & Assertions
 <soap: Header></soap: Header>

 <soap: Body xmlns:m=http://www.stocks.org/stock>
  <m:GetStockPriceResponse>
   <m:Price>34.5<m:Price>
  </m:GetStockPriceResponse>
 </soap:Body>
                                                       Body:Execution Result
  </soap:Envelope>
```

**Fig. 2.18.** SOAP Messages for Web Service Method Invocation.

*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. If we have a look at Fig. 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 the 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. Being an XML-based language, WSDL allows for the automatic generation of a Web service client that can be easily embedded into an existing applications. Moreover, XML is a platform and language-independent specification, thus client for Web services can be generated for any language that is capable of interpret-

ing XML data. This is a fundamental feature that enables Web service interoperability, and one of the reasons that makes such technology a solution of choice for SOA.

Besides those directly supporting Web services, other technologies, which characterize the Web 2.0 [27], provide and contribute to enrich and empower Web applications and then SOA-based systems. These fall under the name 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 fully 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 collection of objects in a platform-independent manner. Often it is preferred for transmitting data in AJAX context because with respect to XML it is a lighter notation, and therefore allows transmitting the same amount of information in a more concise form.

## 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 *"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 since it encompasses design principles to structure, compose, and deploy software systems in terms of services. Therefore, service orientation constitutes a natural approach for shaping Cloud computing systems since it provides 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 of Cloud computing. Parallel and distributed computing emerged as a solution for solving complex/grand challenging 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 they introduce new challenges in terms of hardware architectures, technologies for inter-process 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. Parallel applications often require a specific development environment and compiler to take out the most of the underlying architectures.

Unification of parallel and distributed computing allows one to harness a set of networked and heterogeneous computers, and presents them as a unified resource. Distributed systems constitute a large umbrella under which several different software systems are classified. Architectural styles helps in categorizing and providing reference models for distributed systems. More precisely, software architectural styles define logical organization of components and their roles, while 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: client-server model and peer-to-peer model. 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.

Inter-process communication (IPC) is a fundamental element in distributed systems, and it is the element that ties together separate process and allows them to be seen as a whole. Message-based communication is the most relevant abstraction for inter-process communication which 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 for designing and utilizing distributed systems by making available entire systems or components available on demand.

## Review Questions

1. What is the difference between parallel and distributed computing?
2. Identify the reasons why parallel processing constitutes an interesting option for computing.
3. What is a 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 characterizing 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 inter-process communication in a distributed system?
11. Discuss the most important model for message-based communication.
12. Discuss RPC and how it enables inter-process 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.

# 3 Virtualization

Virtualization technology is one of the fundamental components of Cloud computing, especially in case of infrastructure-based services. It allows creation of secure, customizable, and isolated execution environment for running applications, even if they are untrusted, without affecting other user's applications. At the basis of this technology, there is the ability of a computer program—or more in general a combination of software and hardware—to emulate an executing environment separate from the one that hosts such program. For example, running Windows OS on top of virtual machine, which itself is running on Linux OS. Virtualization provides a great opportunity to build elastically scalable systems, which are capable of provisioning additional capability with minimum costs. Therefore, it is widely used to deliver customizable computing environment on demand.

This chapter discusses the fundamental concepts of virtualization, its evolution, and different 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. This term is often synonymous with *hardware virtualization*, which plays a fundamental role in efficiently delivering *Infrastructure-as-a-Service* solutions for Cloud computing. In fact, virtualization technologies have a long trail in the history of computer science and have come into many flavors by providing virtual environments at operating system level, programming language level, and application level. Moreover, virtualization technologies not only provide a virtual environment for 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 in leveraging this technology. Virtualization technologies have gained a renewed interested recently due to the confluence of different phenomena:

**(a) Increased Performance and Computing Capacity.** Nowadays, the average end-user desktop PC is powerful enough to fulfill almost all the needs of everyday computing, and there is an extra capacity that is rarely used. Almost all of these PCs have resources enough to host a virtual machine manager and execute a virtual machine with a by far acceptable performance. The same consideration applies to the high-end side of the PC market, where supercomputers can provide an immense compute power that can accommodate the execution of hundreds or thousands of virtual machines.

**(b) Underutilized Hardware and Software Resources.** Hardware and software underutilization is occurring due to (1) the increased performance and computing capacity, and (2) 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, there are a lot of computers that are partially utilized, while they could have been used without interruption on a 24/7/365 basis. As an example, desktop PCs mostly required by administrative staff for office automation tasks are only used during work hours, while overnight they remain completely unused. Using these resources for other purposes after work hours could improve the efficiency of the IT infrastructure. In order to transparently provide such a service, it would be necessary to deploy a completely separate environment, which can be achieved through virtualization.

**(C) Lack of Space.** The continuous need for additional capacity, whether this is storage or compute power, makes data centers grow quickly. Companies like Google and Microsoft expand their infrastructure 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 building another data center to accommodate additional resource capacity. This condition along with hardware underutilization led to the diffusion of a technique called server consolidation[13], for which virtualization technologies are fundamental.

**(d) 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 and contribute consistently to the impact that a company has on the environment. Maintaining a data center operational does not only involve keeping servers on, but a lot of energy is also consumed for 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.

**(e) Rise of Administrative Costs.** Power consumption and cooling costs have now become higher than the cost of the 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 the 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 in reducing 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 and, together with them, the other kinds of virtualization. The first step towards a consistent adoption of virtualization technologies has been 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 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 of Java, ability to support multiple programming languages and featuring a complete integration with other Microsoft technologies, the .NET framework soon became the principal development platform for the Microsoft world and

[13] Server consolidation is a technique of aggregating multiple services and applications, originally deployed on different servers on one physical server. Server consolidation allows reducing the power consumption of a data center and resolving hardware underutilization.

quickly became popular among developers. In 2006, two of the three "official languages" used for development at Google were based on the virtual machine model: Java and Python. This trend of shifting towards 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 and it refers to the creation of a virtual version of something, whether this is hardware, software environment, storage, or 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 it 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.

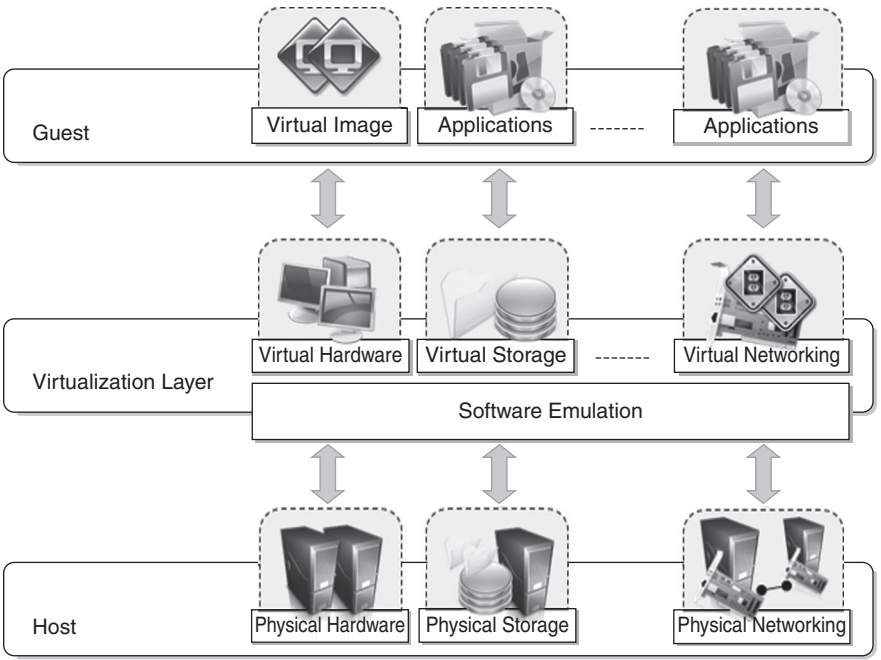


**Fig. 3.1.** Virtualization Reference Model.

Such a general abstraction finds different applications and then implementations of the virtualization technology. The most intuitive and popular is represented by *hardware virtualization*, which also constitutes the original realization of the virtualization concept[14]. In 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 *virtual machine manager*. The host is instead represented by the physical hardware, and in some cases

[14] Virtualization is a technology 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 were available to run multiple operating systems at the same time and provided a backward compatible environment that allowed customers to run previous versions of their applications.

the operating system, that defines the environment where the virtual machine manager is running. In 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—interact 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 be otherwise not available.

The main common characteristic of all these different implementations is the fact that the virtual environment is created by means of a *software program*. The ability of emulate by software 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 a 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.

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

## 2. Managed Execution

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

**(a) Sharing.** Virtualization allows the creation of a separate computing environment within the same host. In this way, it is possible to fully exploit the capabilities of a powerful guest, which would be otherwise 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.

**(b) Aggregation.** It is not only possible to share the physical resource among several guests, but virtualization also allows the 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, and a classical example is represented by cluster management software, which harnesses the physical resources of a homogeneous group of machines and represents them as a single resource.

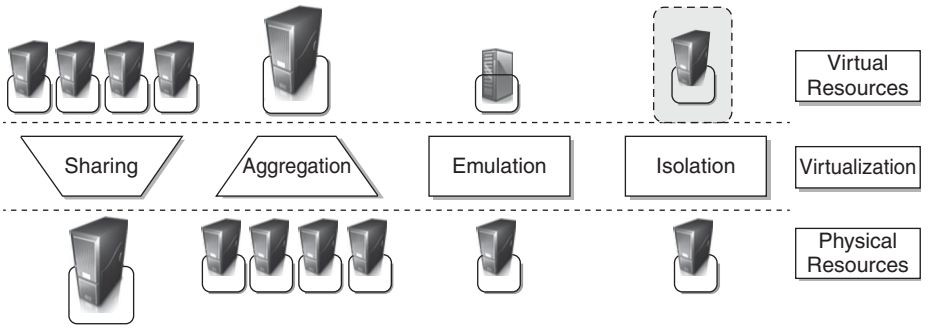


**Fig. 3.2.** Functions Enabled by Managed Execution.

**(c) Emulation.** Guests 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 complete different environment with respect to the host can be emulated, thus allowing the execution of guests 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 the 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 IO, without the hosting machine having such hardware installed. Old and legacy software, which does not meet the requirements of current systems, can be run on emulated hardware without any need of changing their code. This is possible by either 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 represented by arcade game emulators allowing playing arcade games on a normal personal computer.

**(d) Isolation.** Virtualization allows providing guests—whether they are operating systems, applications, or other entities—with a complete separate environment, in which they are executed. The guest 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 each of them interfering with the other. Secondly, 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 time, 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 provides means to effectively implement a Quality of Service infrastructure that more easily fulfills the service level agreement 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 to 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, 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, to move its virtual image into another machine, and to 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 applications demand.

### 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 of the 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 of the 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, given that the required virtual machine manager is available. This requirement is in general less stringent than having all the applications and services you need available 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 to better understand their characteristics and use.

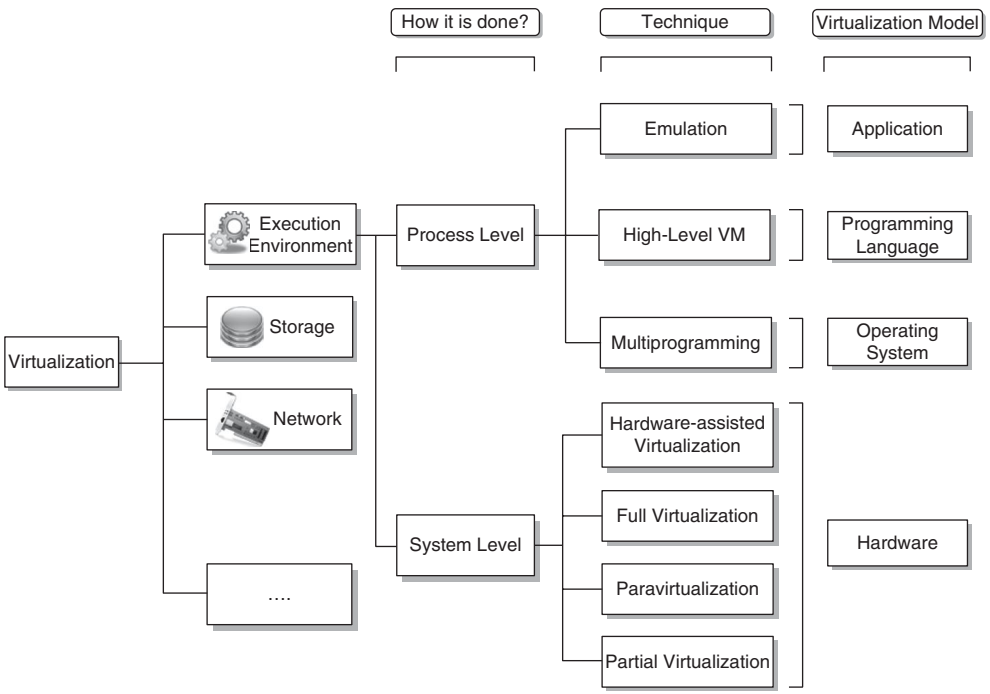


**Fig. 3.3.** Taxonomy of Virtualization Techniques.

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 a 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 support from—an existing operating system. Within these two categories we can list different techniques, which offer to 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 those techniques whose aim is 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 against an application image.

### *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 towards 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.

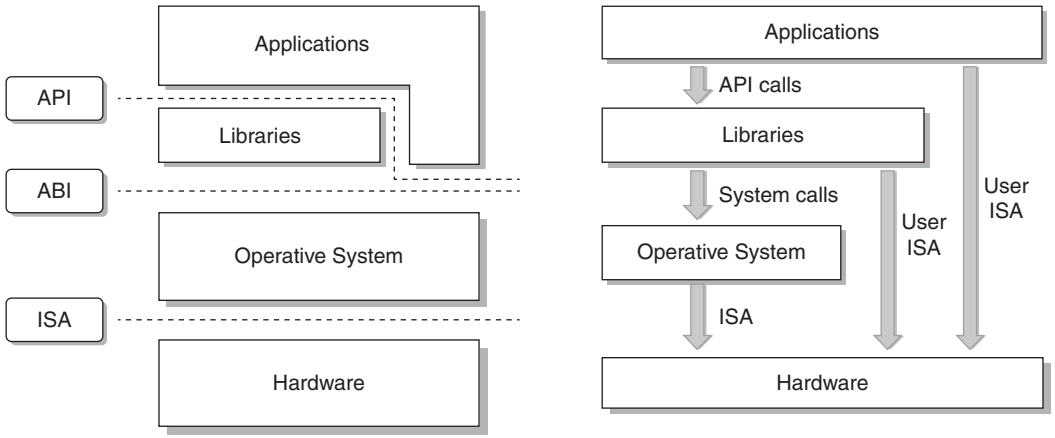


**Fig. 3.4.** Machine Reference Model.

Modern computing systems can be expressed in terms of the reference model described in Fig. 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 interrupts management. ISA is the interface between hardware and software, and it is important for the OS developer (*System ISA*), and developers of applications that directly manage the underlying hardware (*User ISA*). The *Application Binary Interface (ABI)* separates the operating system layer from the applications and libraries, which are managed by the OS. ABI covers details such as low-level data types, alignment, and call conventions and defines a format for executable programs. System calls are defined at this level. This interface allows portability of applications and libraries across operating systems that implement the same ABI. The highest level of abstraction is represented by the *Application Programming Interface (API)*, which interfaces applications to libraries and/or the underlying operating system.

For any operation to be performed in the application level API, ABI and ISA are responsible to make 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 in the hardware level of CPU. This layered approach simplifies the development and implementation of computing systems, the implementation of multi-tasking, and the co-existence of multiple executing environments. In fact, such a model not only requires limited knowledge of the entire computing stack, but also provides ways for implementing 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, which define who can operate with them. The first distinction can be made between *privileged* and *non-privileged* instructions. Non-privileged 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, while 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 on 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 the 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 the supervisor mode and Ring 3 for user mode.

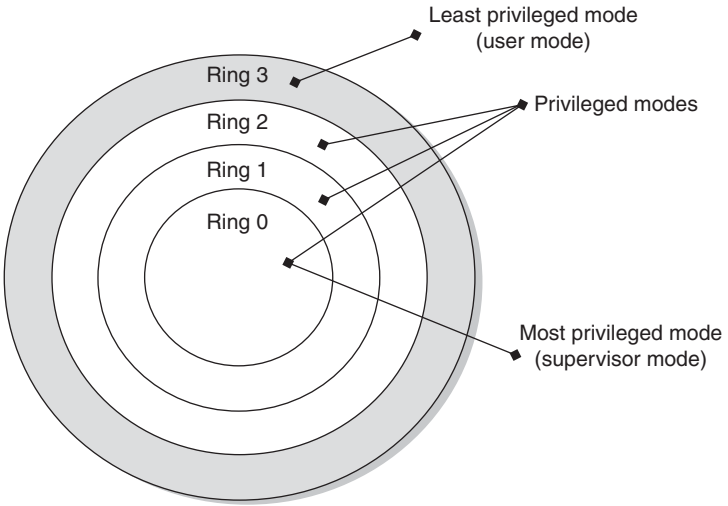


**Fig. 3.5.** Security Rings and Privileged Modes.

All the current systems support at least two different execution modes: *supervisor mode* and *user mode*. The first mode denotes an execution mode where all the instructions (privileged and non-privileged) can be executed without any restriction. This mode is also called *master mode*, or *kernel mode* and it 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 condition and non-privileged 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 so. 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 non-privileged instructions has posed challenges in designing virtual machine managers. It is

expected that all the sensitive instructions are executed in privileged mode, which requires a supervisor mode in order to avoid traps. This is because, 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[15]. 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 different techniques utilized to virtualize execution environment and their relations to the other components of the system.

## 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 virtual machine manager by the *hypervisor*. The hypervisor is generally a program, or a combination of software and hardware, that allows the abstraction of the underlying physical hardware.

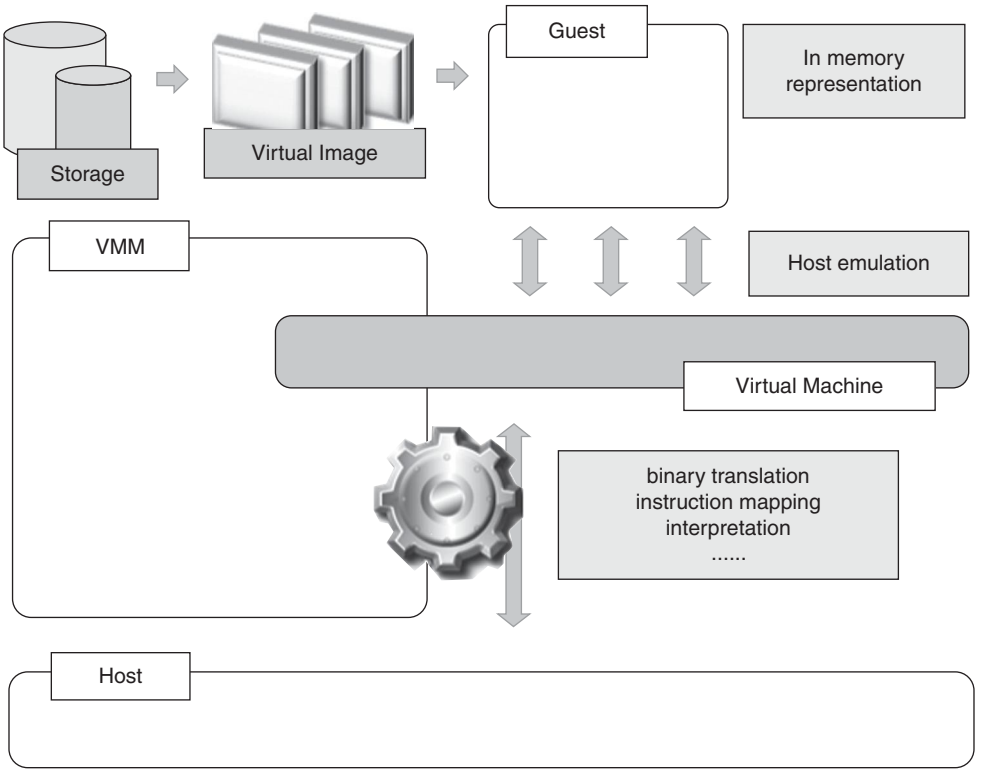


**Fig. 3.6.** Hardware Virtualization Reference Model.

[15] It is expected that in hypervisor-managed environment, the entire guest operating system code is 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 (i.e., implemented as non-privileged instructions), it is not possible anymore to completely isolate the guest OS.

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 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, where guest operating systems are installed. There are two major types of hypervisors: *Type I* and *Type II*.

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

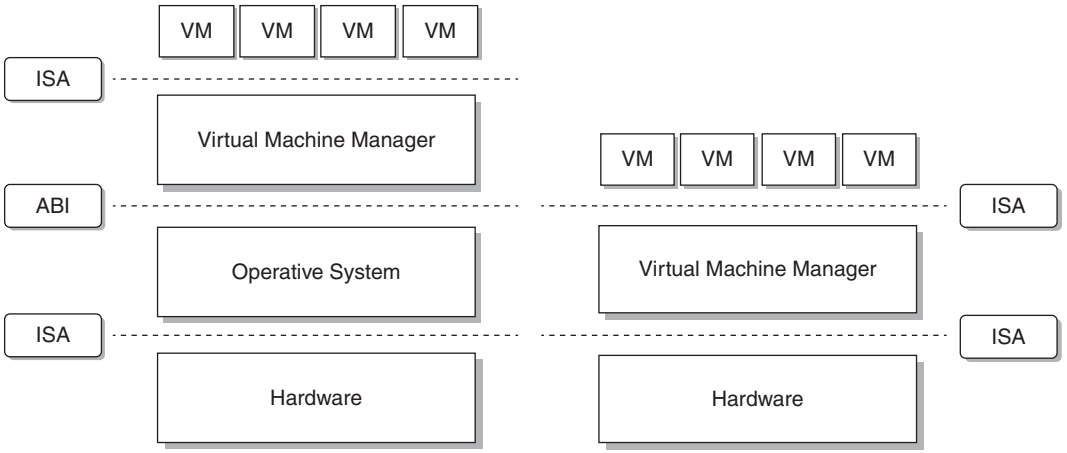


**Fig. 3.7.** Hosted (left) and Native (right) Virtual Machine.

Conceptually, a virtual machine manager is internally organized as described in Fig. 3.8. Three main modules coordinate their activity in order to emulate the underlying hardware: *dispatcher*, *allocator*, and *interpreter*. 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 if it was 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 executed directly on the physical host.

*Resource Control.* The virtual machine manager should be in complete control of virtualized resources.

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

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.

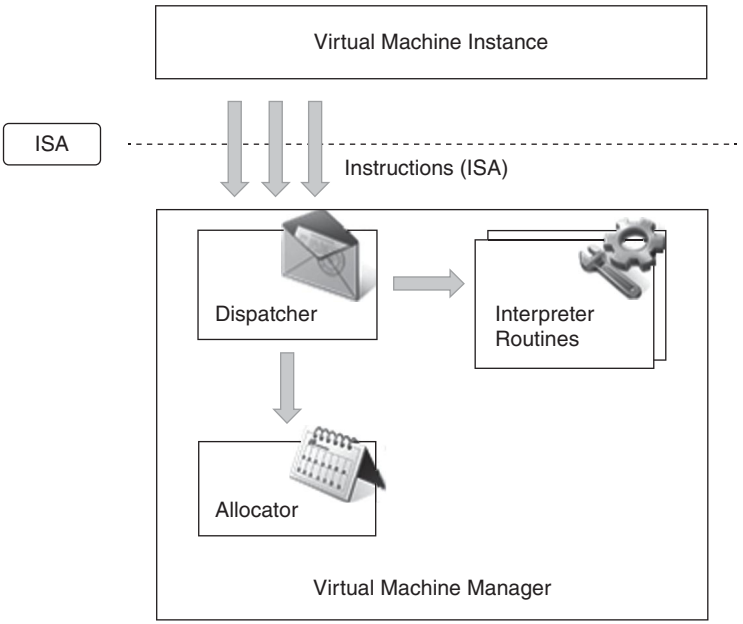


**Fig. 3.8.** Hypervisor Reference Architecture.

**Theorem 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 trap from the 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 non-privileged instructions must be executed without the intervention of 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.

**Theorem 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 of running 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.

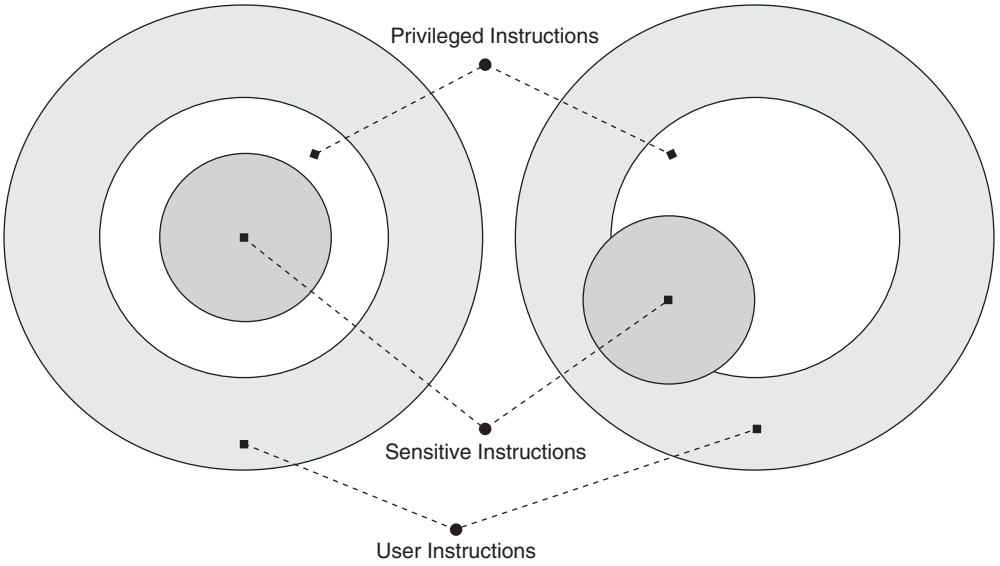


**Fig. 3.9.** Virtualizable Computer (left) and Non Virtualizable Computer (right).

**Theorem 3:** *A hybrid VMM may be constructed for any conventional third generation machine, in which the set of user sensitive instructions are a subset of the set of privileged instructions.*

There is another term called *Hybrid Virtual Machine (HVM)*, which is less efficient than the virtual machine system. In case of 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, ie., 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 on—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.

## 3. *Hardware Virtualization Techniques*

**(a) 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 in order to trap some sensitive instruction and provide an emulated version. Products such as VMware Virtual Platform, introduced in 1999 by VMware who 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.

**(b) Full Virtualization.** Full virtualization refers to the ability of running a program, most likely an operating system, on top of a virtual machine directly and without any modification, as if it were run on the raw hardware. In order 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, it poses important concerns on 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 to the performance. A successful and efficient implementation of full virtualization is obtained with a combination of hardware and software allowing not potentially harmful instructions to be executed directly on the host. This is what is accomplished through hardware-assisted virtualization.

**(c) 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 operation 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. In order to take advantage of such 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 why 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* [24] project 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. This solution is provided by Xen for running Windows-based operating systems on x86 architectures. Other solutions using paravirtualization include: *VMWare*, *Parallels*, and some solutions for embedded and real-time environment such as *TRANGO*, *Wind River*, and *XtratuM*.

**(d) 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, proces sor, 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.

# *4. Operating System Level Virtualization*

Operating system level virtualization offers the opportunity to create different and separated execution environments for applications that are managed concurrently. It is different 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, 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 name space 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. By 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 a 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 in order to run them, neither 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. By aggregating different servers 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 each of 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.

## 5. Programming-Language-Level Virtualization

Programming-language-level virtualization is mostly used for achieving 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 used this technology 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 run time, the byte code can be either interpreted or compiled on the fly—*jitted*[16]—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 became popular again with the introduction of the *Java* platform, in 1996, by Sun. Originally created as a platform for developing Internet applications, it 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 of supporting multiple programming languages has been one of the key elements of the *Common Language Infrastructure (CLI)*, which is the specification behind the .NET framework. Currently,

[16] 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 JIT compilation is that the machine code that has been compiled can be re-used for executing future calls to the same methods. Virtual machine 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.

the Java platform and the .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 operand on the stack, perform some operations with them, and put the result on the stack. Additionally, specific instructions for invoking methods, and managing object and classes are included. Stack based virtual machines possess the property of being easily interpreted and executed simply by lexical analysis, and hence to be 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 register-based virtual machine is *Parrot*—a programming-level virtual machine, 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 of providing a uniform execution environment across different a platforms. Programs compiled into byte code can be executed on any operating system, and a platform, for which a virtual machine able to execute that code, has been provided. From a development life cycle 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 point 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 prize: performance. Virtual machine programming languages generally expose an inferior performance, if 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.

## 6. Application-Level Virtualization

Application-level virtualization is a technique allowing applications to be run on runtime environments which do not natively support all the features required by such applications. In this scenario, applications are not installed in the expected runtime environment, but run as if 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:

**(a) Interpretation.** In this technique, every source instruction is interpreted by 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.

**(b) 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 a better performance, since previously translated instruction blocks are directly executed.

Emulation, as described above, is different from hardware-level virtualization. The former simply allows the execution of a program compiled against a different hardware, while 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 run-time environment if 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 solution 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 inspiration from a similar product from Sun: *WABI (Windows Application Binary Interface)*, 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* is another product in this area, which 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, there exist other types of virtualization which provide an abstract environment to interact with. These mainly cover storage, networking, and client-server interaction.

### *1. Storage Virtualization*

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

### *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 if 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 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. Desktop Virtualization*

Desktop virtualization abstracts the desktop environment available on a personal computer in order to provide access to it by using a client-server approach. Desktop virtualization provides the same outcome of hardware virtualization but serves a different purpose. Similarly to hardware virtualization, it makes accessible a different system, as if it was 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. While 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 which provides a high availability infrastructure, and ensures the accessibility and the 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 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 are: *Sun Virtual Desktop Infrastructure (VDI)*, *Parallels Virtual Desktop Infrastructure (VDI)*, *Citrix XenDesktop* and others.

### 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 of the cases, is a complimentary feature, which is naturally needed when building 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 market segment, while programming language virtualization is a technology leveraged in Platform-as-a-Service 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 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 of designing more efficient computing systems by means of consolidation, which is performed transparently to Cloud computing service users. Since virtualization allows creating 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*. As 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[17].

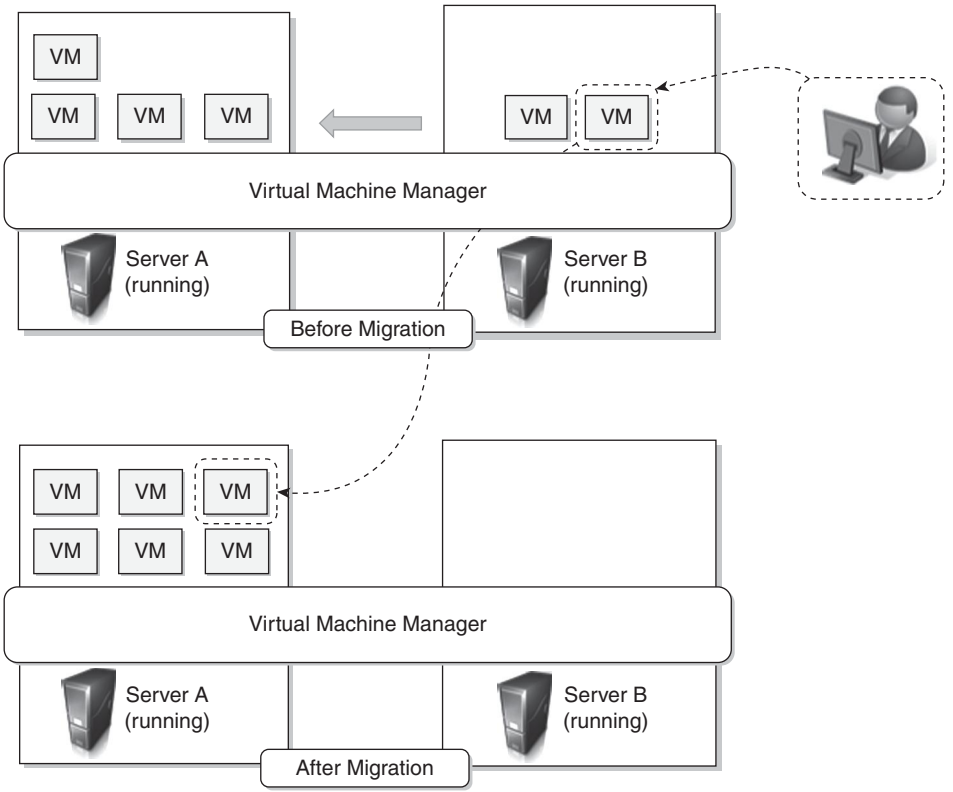


**Fig. 3.10.** Live Migration and Server Consolidation.

Server consolidation and virtual machine migration are principally used in case of hardware virtualization even though it is technically possible also in case of programming language virtualization.

Storage virtualization constitutes an interesting opportunity given by virtualization technologies, often complementary to the execution virtualization. Even in this case, vendors backed by large computing infrastructure 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 well as methods for an easy accountability of such service.

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.

[17] It is important to notice that Cloud computing is strongly leveraged for the development of applications that need to scale on demand. In most of the 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 since it does not create any service interruption during consolidation.

# 3.5 PROS AND CONS OF VIRTUALIZATION

Virtualization has now become extremely popular and is largely used, especially in Cloud computing. The primary reason of its wide success is the elimination of technology barriers that made virtualization not 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 advancement in the 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.

## *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 environment, 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 that is also a requirement for an 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 the Java virtual machine to 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 if 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 reduce the costs for maintenance, since the number of hosts is expected to be lower than the number of virtual machine instances. Being the guest executed in a virtual environment, which is often part of the virtual instance itself, there is no component that is subject to change or damage over time. Moreover, it is expected to have 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 have less impact on the environment.

## *2. The Other Side of the Coin: Disadvantages*

Virtualization has also downsides. The most evident is represented by a performance decrease of guest systems, as a result of the intermediation performed by the virtualization layer. Also, 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 on security, which are mostly due to the ability of emulating a different execution environment.

**(a) Performance Degradation.** Performance is definitely one of the major concerns when using virtualization technology. Since virtualization interposes an abstraction layer between the guest and the host, increased latencies and delays can be experienced by the guest.

For instance, in 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 by the overhead introduced by the following activities:

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

Also, 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 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, being their execution 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 the technology advancements and the ever increasing computational power available today. For example, specific techniques of hardware virtualization such as *paravirtualization* can increase the performance of guest execution by demanding most of the guest execution to the host without any change. In the case of programming level virtual machines such as the JVM or the .NET, compilation to native code is offered as an option when performance is a serious concern.

**(b) Inefficiency and Degraded User Experience.** Virtualization can sometimes 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 they become inaccessible. In the case of hardware virtualization, this could happen when for device drivers,: the virtual machine can sometimes just provide a default graphic card which 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, if compared to native applications. These issues have been resolved by providing a new framework for designing the user interface–*Swing*,–and further improvements have been done by integrating support for the *OpenGL* libraries into the Software Development Kit.

**(c) Security Holes and New Threats.** Virtualization opens the door to a new and unexpected form of phishing[18]. The capability of emulating a host in a complete transparent manner, has led the way to malicious programs which 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 towards it. The operating system is then controlled, and can be manipulated in order to extract sensitive information of interest for third parties. Examples of these kind of malware are *BluePill* and *SubVirt*. *BluePill* is a malware targeting the AMD processor family and 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.

[18] Phishing is a term that identifies a malicious practice aimed at capturing sensitive information, such as user names and passwords, by recreating an environment identical in functionalities and appearance to the one that manages this information. Phishing is most commonly used in the Web, where the user is redirected to a malicious Website that is a replica of the original one and whose purpose is to collect the information to impersonate the user against the original Website (e.g.,. a bank Website) and access his or her confidential data.

The diffusion of such kind of malware is facilitated by the fact that originally, hardware and CPU were not manufactured by keeping the virtualization in mind. In particular, the existing instruction sets cannot be simply changed or updated to suit the needs of the virtualization. Recently, both Intel and AMD have introduced hardware support for virtualization with *Intel VT* and *AMD Pacifica*.

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. In order to make this possible, the original version of the runtime environment needs to be replaced by the modified one, and this 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

There is a wide range of virtualization technologies 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.

## 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, it has now a large open source community backing it. It is also offered as a commercial solution, XenSource, by Citrix. 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, there 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 requiring special management. This is done by modifying portion 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 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 hosts an 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 system providing Infrastructure-as-a-Service (IaaS) solutions.

Many of the x86 implementations support four different security levels, called rings, where Ring 0 represents the level with the highest privileges, and Ring 3 represents the level with the lowest ones. Almost all the most popular operating systems, except for OS/2, utilize only two levels: Ring 0 for the kernel code, and Ring 3 for user application and non-privileged 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 as *Domain U*—in Ring 1, while the user applications are run in Ring 3. This allows Xen to maintain unchanged the *Application Binary Interface (ABI)* thus allowing an easy switch to Xen-virtualized solutions, from an application point of view. Be-

cause of the structure of x86 instruction set, there are some instructions which allow code executing in Ring 3 to jump into Ring 0 (kernel mode). Such an operation is performed at hardware level, and therefore within a virtualized environment, it 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. In order to avoid this situation, operating systems need to be changed in their implementation, and the sensitive system calls need to be re-implemented 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.

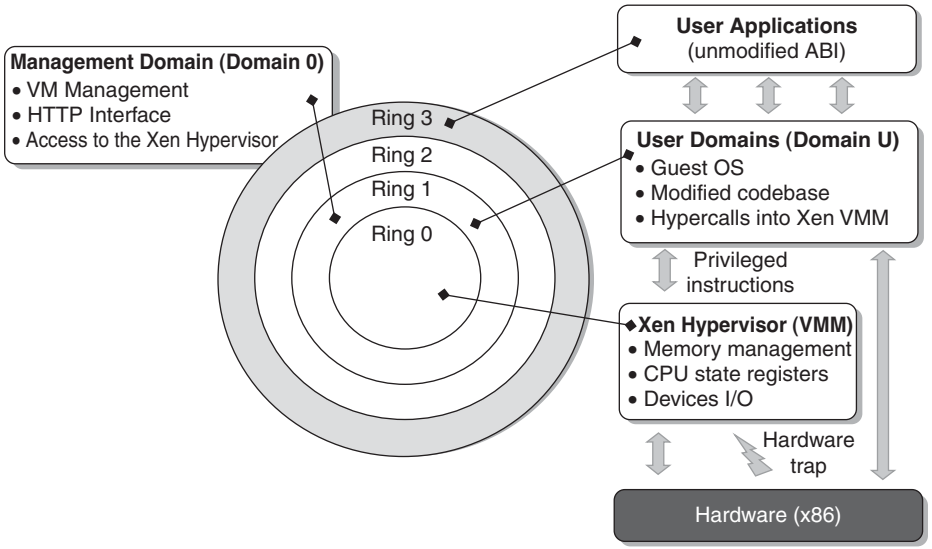


**Fig. 3.11.** Xen Architecture and Guest OS Management.

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 Ring1 and the guest operating system in Ring 0. Therefore, Xen exhibits some limitations in 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, while 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 layer 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 of the cases, full virtualization is made possible by means of *direct execution* (for non-sensitive 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.

## 1. Full Virtualization and Binary Translation

VMware is well-known for the capability of virtualizing x86 architectures, which runs unmodified on-top of their hypervisors. With the new generation of hardware architectures and the introduction of *hardware assisted virtualization* (i.e., 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's 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 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, in order to improve performance, the equivalent set of instruction is cached, so that translation is not necessary anymore for further occurrences of the same instructions. Figure 3.12 gives an idea of the process.

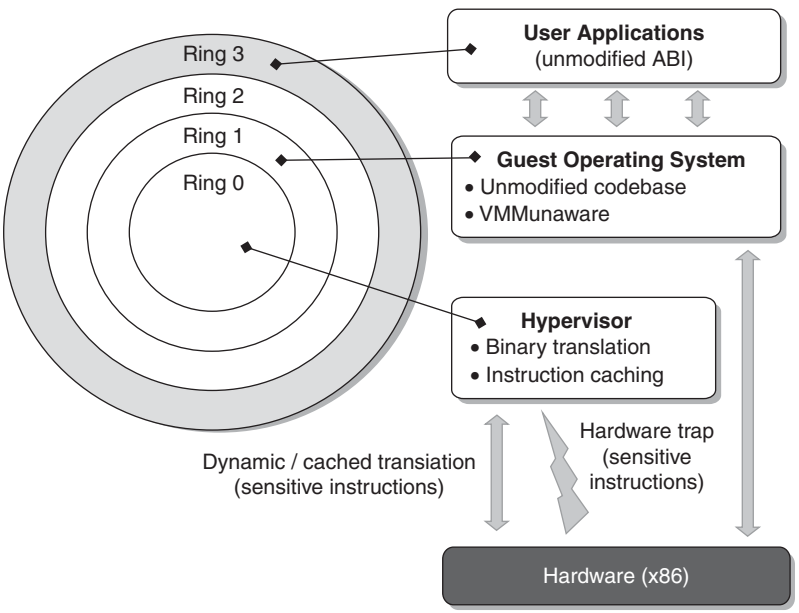


**Fig. 3.12.** Full Virtualization Reference Model.

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 whose 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 disadvantages exists, binary translation is only applied to a subset of the instruction set, while the others are managed through direct execution on the underlying hardware. This reduces somehow 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, which can deeply impact the performance without the appropriate hardware support. The main reason of this is the presence of a *Memory Management Unit (MMU),* which needs to be emulated as part of the virtual hardware. Especially, in 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. In order 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. Finally, VMware also provides a full virtualization of I/O devices such as network controllers, and other peripherals such as keyboard, mouse, disks, and USB controllers.

## 2. Virtualization Solutions

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

**(a) End-User (Desktop) Virtualization.** VMware supports virtualization of operating system environments and single applications on end-users computers. The first option is the most popular, and allows installing 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 GUI of the host operating system). Fig. 3.13 provides an overview of the architecture of these systems.

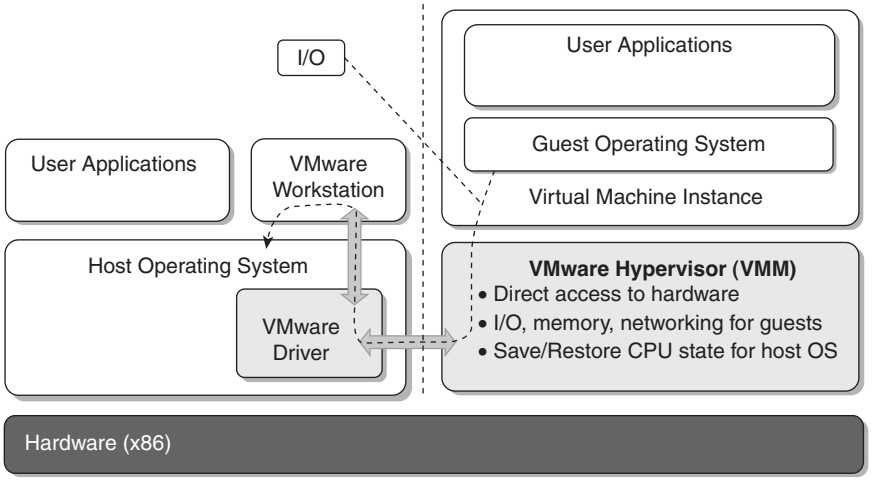


**Fig. 3.13.** VMware Workstation Architecture.

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

By 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 a 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 controls 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, pausing their execution, creating snapshots, and un-do operations by rolling back to a previous state of the virtual machine.

Other solutions related to the virtualization of end-user computing environment include *VMware Player*, *VMware ACE*, and *VMware ThinApp*. VMware Player is a reduced version of VMware Workstation which allows creating and playing virtual machines on a Windows or Linux operating environment. *VMware ACE* is a similar product to VMware Workstation which 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*.

**(b) 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 Fig. 3.14.

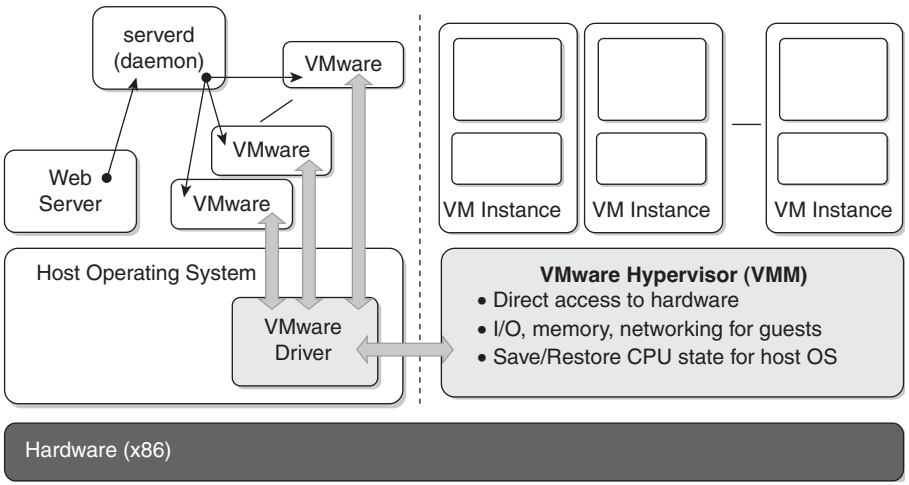


**Fig. 3.14.** VMware GSX Server Architecture.

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 request 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 hypervisor -based approach. Both of them can be installed on bare metal server 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 Fig. 3.15. The base of the infrastructure is constituted by the *VMkernel* which is a thin 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 (User world API). These 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)*[19] 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 for local users.

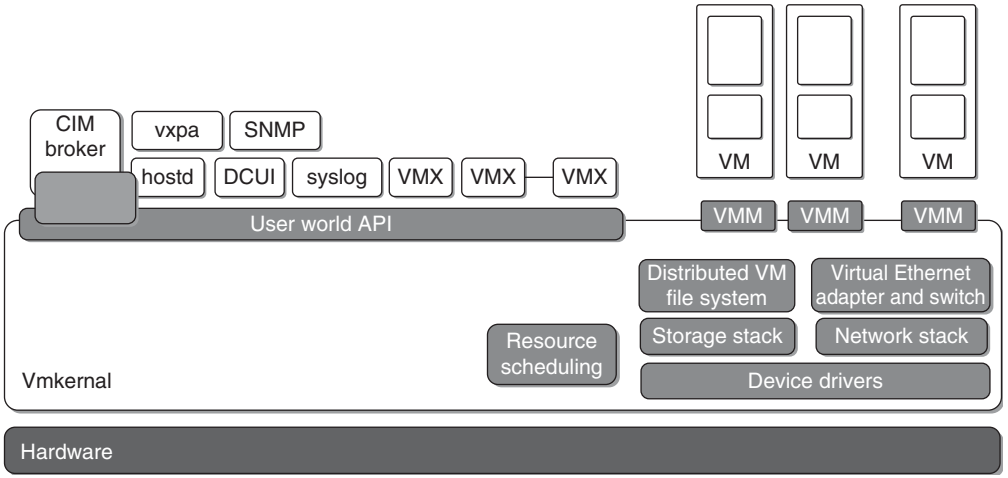


**Fig. 3.15.** VMware ESXi Server Architecture.

**(c) 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 solution 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; and application services, such as virtual machine migration, storage migration, data recovery, and security zones, complete the services offered by vSphere. The management of such 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 a virtual computing environment, 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.

[19] 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*.

VMware also provides a solution for application development in the Cloud with *VMware vFabric*, which is a set of components that facilitates 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.

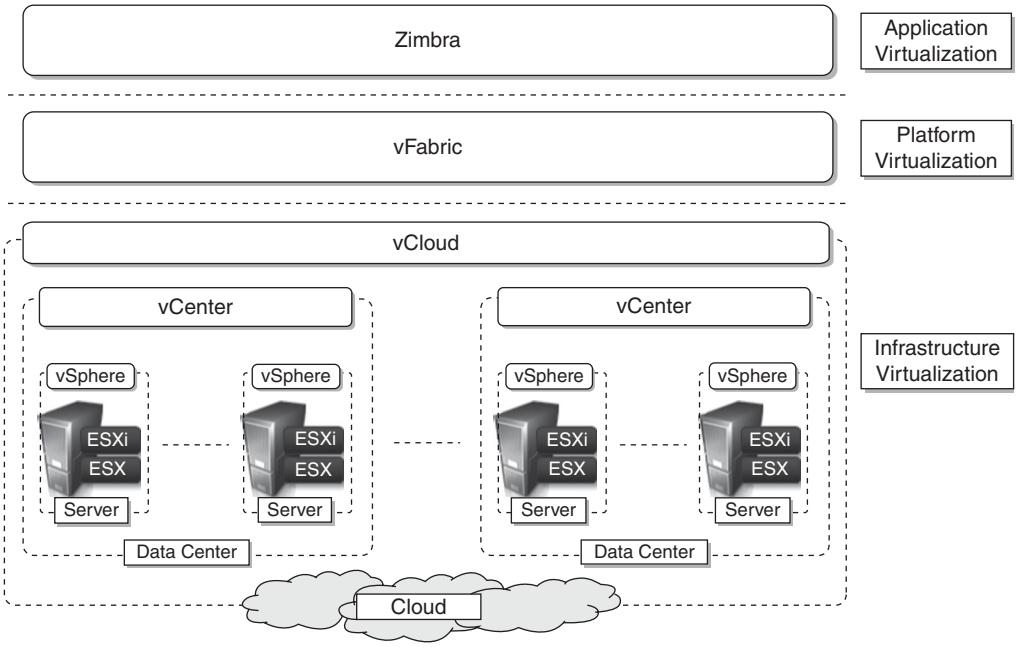


**Fig. 3.16.** VMware Cloud Solution Stack.

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

## 3. Observations

By initially starting with a solution for a 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 host operating environment. Also, VMware has strongly contributed to the development and the 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 for hardware virtualization, which leverages several techniques to support a variety of different 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.

# 1. Architecture

Hyper-V supports multiple and concurrent execution of the guest operating system 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 *parent partition.* The parent partition (also called root partition) is the only one that has direct access to the hardware, it runs the virtualization stack, host 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.

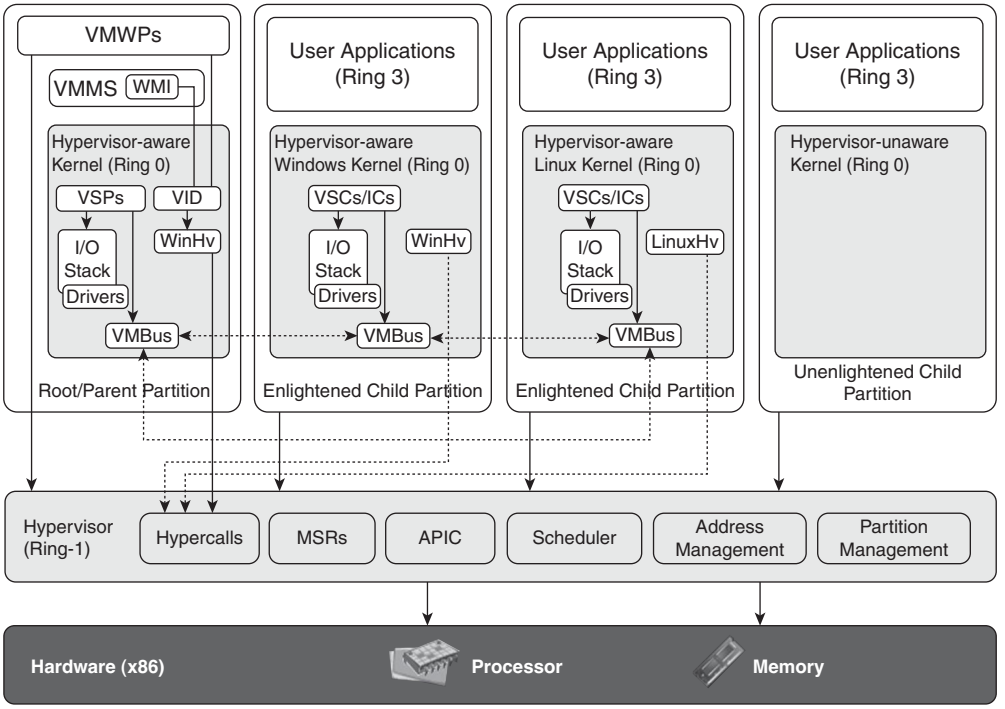


**Fig. 3.17.** Microsoft Hyper-V Architecture.

**(a) 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 sensible instructions. This is an implementation of para-virtualization approach already discussed with Xen. This interface is used by drivers in the partitioned operating system to contact the hypervisor by using the standard Windows calling convention. The parent partition also uses this interface to create children 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 the 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 it 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 children partitions.

**(b) 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 inter-partition 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 inter-partition communication channel that is used to exchange data between partitions (children and parent), and it is utilized mostly for the implementation of virtual device drivers for guest operating systems.

The architecture of Enlightened I/O is described in Fig. 3.17. There are three fundamental components: *VMBus*, *Virtual Service Providers (VSPs)*, and *Virtual Service Clients (VSCs)*. The first one 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 children partitions. Operating systems supported by Hyper-V utilize this preferred communication channel in order to perform I/O for storage, networking, graphics, and input subsystems. This also results in an enhanced performance in case of 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.

**(c) 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 children partitions. This partition is the only one that directly accesses device drivers and mediates the access to them by children partitions by hosting the Virtual Service Providers.

The parent partition is also the one that manages the creation, execution, and destruction of children partitions. It does so by means of the *Virtualization Infrastructure Driver (VID)*, which controls the access to the hypervisor, and also allows the management of virtual processors and memory. For each children partition created, a Virtual Machine Worker Process (VMWP) is instantiated in the parent partition, which manages the children partition by interacting with the hypervisor through the VID. Virtual Machine

Management services are also accessible remotely through a WMI[20] provider that allows remote hosts to access the VDI.

**(d) Children Partitions.** Children partitions are used to execute guest operating systems. These are isolated environments, which allow a secure and controlled execution of guests. There are two types of children partitions depending on whether the guest operating system is supported by Hyper-V or not. These are called *Enlightened* and *Unenlightened* partitions respectively. The first one can benefit from Enlightened I/O while the other ones are executed by leveraging hardware emulation from the hypervisor.

## 2. Cloud Computing and Infrastructure Management

Hyper-V constitutes the basic building block of Microsoft virtualization infrastructure. Other components contributed to create a ful-featured platform for server virtualization.

In order to increase the performance of a virtualized environment, a new version of Windows Server 2008, called *Windows Server Core*, has been released. This is a specific version of the operating system with a reduced set of features and 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 graphical user interface component, and other bulky components, such as the .NET framework and all the applications developed on top of it (i.e. ,*PowerShell*). This design decision has both advantages and disadvantages. On the good side, it allows for reduced maintenance (i.e., fewer software patches), reduced attack surface, reduced management, and less disk space. On the bad side, the embedded features are reduced. Still, there is the opportunity of leveraging all the "removed features" by means of remote management from a ful-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 life cycle 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 and
- host capacity management.

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

## 3. Observations

When compared with Xen and VMware, Hyper-V is a hybrid solution as 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 children partitions map Domains U. The only difference

[20] *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 which give access to a specific subsystem of the hardware to authorized clients.

is that the Xen hypervisor is installed on bare hardware and filters all the access to the underlying hardware, while Hyper-V is installed as a role in the existing operating system, and the way in which it interacts with partitions is quite similar to the strategy implemented by VMware and previously 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. Being installed as a role in Windows Server 2008, x64 can only be installed on 64-bit hardware platforms. Moreover, it requires a 64-bit processor supporting hardware-assisted virtualization and data execution prevention. For the software requirements different from vSphere and Xen that are installed on bare hardware, Hyper-V requires Microsoft Windows Server 2008.

## Summary

The term “virtualization” is a large umbrella under which different 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 this is 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 a network connection.

## Review Questions

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

# 4 Cloud-Computing Architecture

The term "*Cloud computing"* is a wide umbrella encompassing many different things. Lately, it has become a buzzword, easily misused to revamp existing technologies and ideas for the public. What makes it 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, which serves as a basis for discussion on 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 it is virtual hardware, development platform, or application software, relies on a distributed infrastructure owned by the provider or rented from a third party. As it can be noticed from the previous definition, the characterization of a Cloud is quite general: it can be implemented by using a datacenter, a collection of clusters, or a heterogeneous distributed system composed by desktop PC, workstations, and servers. Commonly, Clouds are built by relying on one or more datacenters. In most of the cases, hardware resources are virtualized to provide isolations of workloads and to exploit, at best, the infrastructure. According to the specific service delivered to the end user, different layers can be stacked up on top of the virtual infrastructure: a virtual machine manager, a development platform, or a specific application middleware.

As noted in earlier chapters, Cloud-computing paradigm emerged as a result of convergence of various existing models, technologies and concepts, which changed the way we deliver and use IT services. A broad definition of the phenomenon could then 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, which aids categorization of Cloud technologies, applications, and services.

# 4.2 CLOUD REFERENCE MODEL

Cloud computing supports any IT service that can be consumed as a utility and it is delivered through the 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 Fig. 4.1): from the hardware appliances to software systems. Cloud resources are harnessed to offer "computing/horse power" required for providing services. Often, this layer is implemented by using a datacenter in which hundreds and thousands of nodes are stacked together. Cloud infrastructure can be in heterogeneous nature as 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.

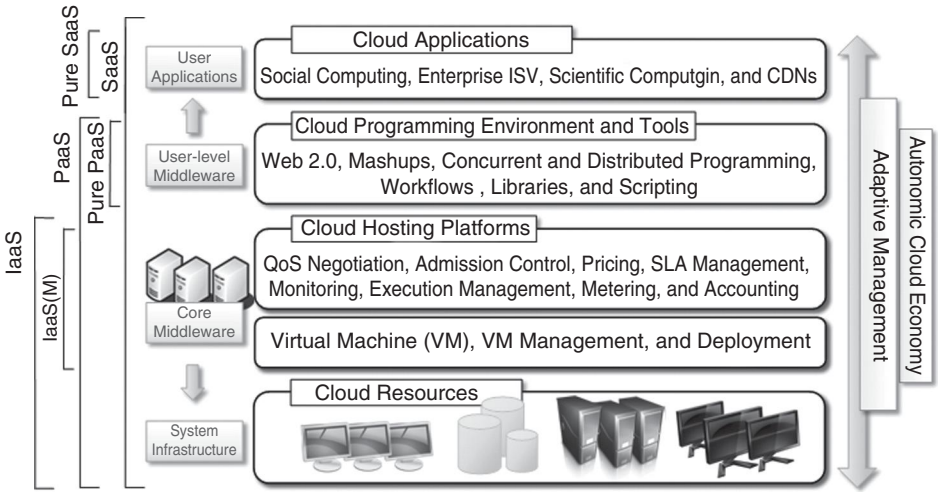


**Fig. 4.1.** Cloud-Computing Architecture.

The physical infrastructure is managed by the core middleware whose objectives are to provide an appropriate runtime environment for applications and to utilize resources at best. 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 the 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, memory, and also virtualize specific devices, thus meeting the requirement of users and applications. This solution is generally paired 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 creating a portable runtime environment where applications can be run and controlled. This scenario generally implies applications hosted in the Cloud to 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 the 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 providing physical infrastructure and adds value to them.

Infrastructure-as-a-Service 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 to 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, that is bundled as part of the service provided to users. In 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 Fig. 4.1 contains services delivered at application level. These are mostly referred as *Software-as-a-Service (SaaS)*. In most of the cases, these are Web-based applications that rely on the Cloud to provide service to end users. The horse power 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 strongly leveraging 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 Web sites.

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 the users.

**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 machines management infrastructure; Storage management; Network management. | Amazon EC2 and S3; GoGrid; Nirvanix… |

The reference model described in Fig. 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 composed together in order to provide a completely integrated solution covering all the computing stack of a system. Infrastructure-as-a-Service providers can offer the bare metal in terms of virtual machines where Platform-as-a-Service 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 by Web servers, database servers, and load balancers on top of which pre-packaged software is installed to run Web applications. This possibility has made Cloud computing an interesting option for reducing the capital investment in IT of start-ups, which can 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 them along with some references to practical implementations.

## 4.2.2 Infrastructure/Hardware as a Service

Infrastructure and Hardware as a Service 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, 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 processor, 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, it allows better exploitation of the IT infrastructure, and provides a more secure environment for 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 of the cases, virtual machines come with only the selected operating system installed and the system can be configured with all the required packages and applications. Other solutions provide prepackaged system images already containing the software stack required for the most common uses: Web servers, database servers, or LAMP[21] stacks. Besides the basic virtual machine management capabilities, additional services can be provided and they generally include the following: SLA resource based allocation, workload management, support for infrastructure design through advanced Web interfaces, and 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 interface is 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 now allow developing full-featured management consoles completely hosted in a browser or a Web page. Web services and RESTful APIs allow program to interact with the service without the human intervention, thus providing complete integration within a software system. The core features of an Infrastructure-as-a-Service solution are implemented in the infrastructure management software layer. In particular, the 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 performing different tasks:

- The *pricing*/*billing* component takes care of the cost of executing each virtual machine instance and maintains data that will be used to charge the user.

[21] *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 as server side scripting technology for developing Web applications. LAMP stacks are the most common packaged solutions for quickly deploying Web applications.

- 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.
- In case support for QoS-based execution is provided, a *QoS*/*SLA management* component will maintain a repository of all the service level agreements made with the users, and together with the monitoring 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 the users to upload their specific virtual machine image.
- A *VM pool manager* component is responsible of 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 in order to provide a virtual machine instance that is external to the local physical infrastructure directly managed by the pool.

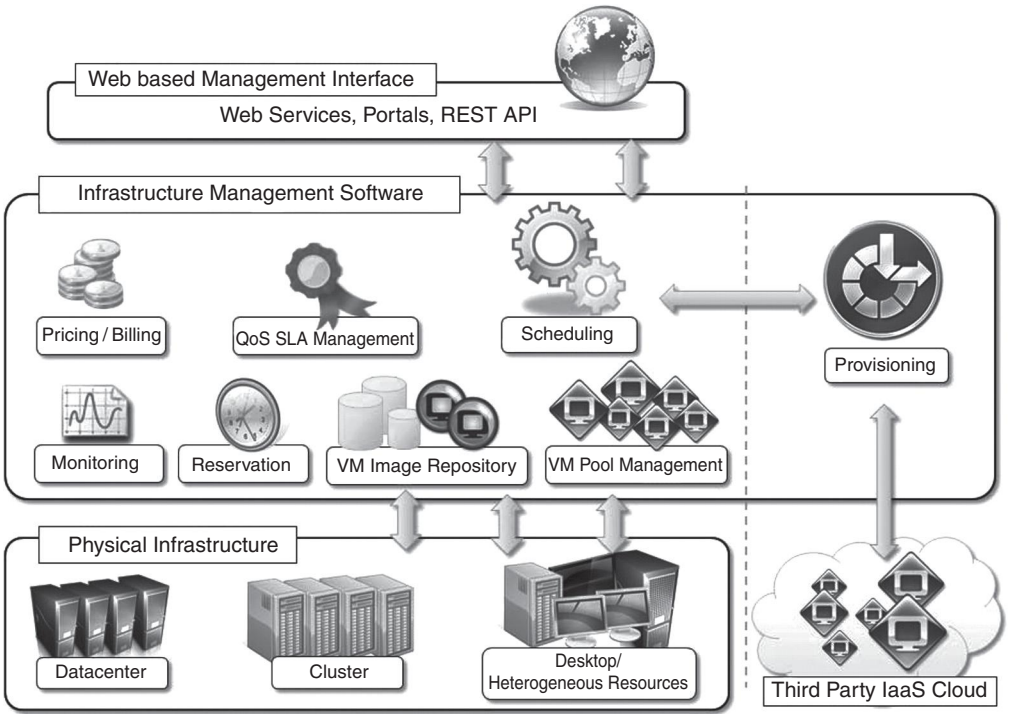


**Fig. 4.2.** Infrastructure as a Service Reference Implementation.

The bottom layer is constituted by the physical infrastructure on top of which the management layer operates. As previously discussed, the infrastructure can be of different types, and 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 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 can be aggregated: PCs, workstations, and clusters. This case mostly represents an evolution

of desktop grids where any available computing resource (such as PCs and workstations 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 case of complete IaaS solutions, all the three levels are offered as service. This is generally the case of public clouds vendors, such as Amazon, GoGrid, Joyent, Rightscale, Terremark, Rackspace, ElasticHosts, and Flexiscale, who own large datacenters and give access to their computing infrastructures by 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 where the management software is installed. This is the case of *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 a general insight about 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 do not provide support for some of the features discussed here. Finally, the reference architecture applies to IaaS implementations which provide computing resources, especially for the scheduling component. In case storage is the main service provided, it is still possible to distinguish these three layers. The role of infrastructure management software is not keeping track and managing 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 Fig. 4.3.

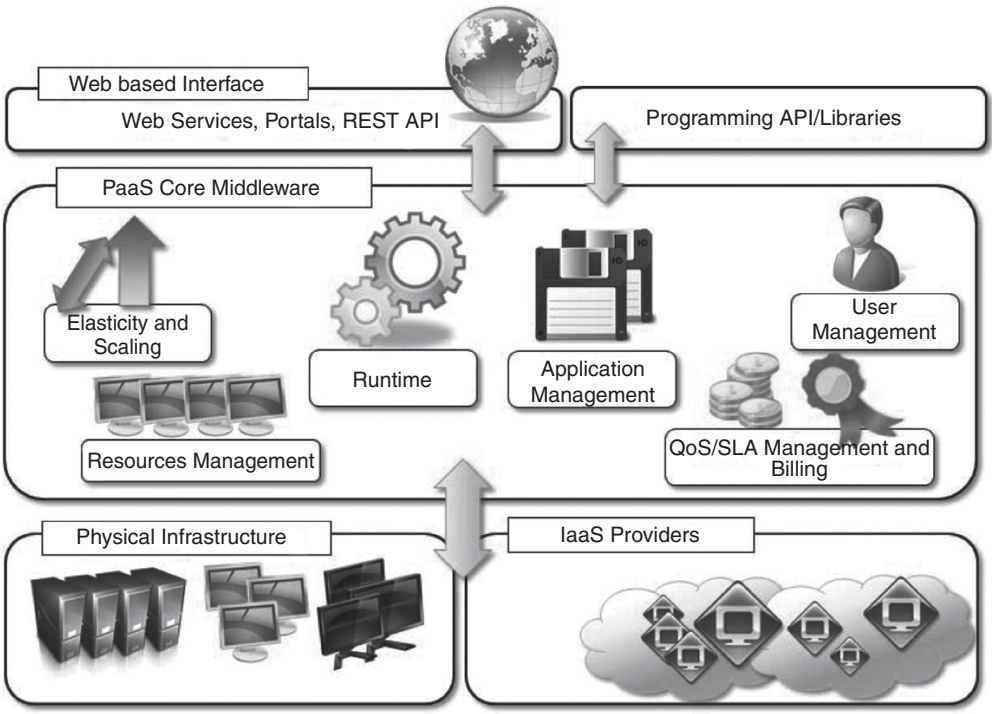


**Fig. 4.3.** Platform as a Service Reference Model.

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 applications 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 system 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 the 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 offering different 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.

Platform-as-a-Service solutions can offer a middleware for developing applications together with the infrastructure, or simply provide users with the software that is installed on the user's premises. In the first case, the PaaS provider also owns large datacenters where applications are executed, in the second case—referred 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 also only the middleware for private installations.

Table 4.2 provides a classification of the most popular PaaS implementations. It is possible to organize the different solutions into three wide categories: *PaaS-I*, *PaaS-II*, and *PaaS-III*. The first category identifies PaaS implementations which 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 of them deliver, as a platform, the combination of a middleware and infrastructure. In the second class, we can list all those solutions which 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 applications. *GoogleAppEngine* 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* is an open source implementation of Google AppEngine, and provides an 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* that 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 applications and not only Web applications. Among these, the most popular is *Microsoft Windows Azure* that 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 a middleware with different services. These are only few options available in the Platform-as-a-Service market segment.

**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<br>Middleware + Infrastructure | Force.com<br>Longjump |
| *PaaS-II* | Runtime environment for scaling Web applications. The runtime could be enhanced by additional components which provide scaling capabilities. | Middleware + Infrastructure<br>Middleware<br>Middleware + Infrastructure<br>Middleware + Infrastructure<br>Middleware + Infrastructure<br>Middleware | Google AppEngine<br>AppScale<br>Heroku<br>Engine Yard<br>Joyent Smart Platform<br>GigaSpaces XAP |
| *PaaS-III* | Middleware and programming model for developing distributed applications in the Cloud. | Middleware + Infrastructure<br>Middleware<br>Middleware<br>Middleware<br>Middleware<br>Middleware | Microsoft Azure<br>DataSynapse<br>Cloud IQ<br>Manjrasof Aneka<br>Apprenda SaaSGrid<br>GigaSpaces DataGrid |

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[22], there are some essential characteristics that identify a Platform-as-a-Service solution:

**(a) Runtime Framework.** It represents the "software stack" of the PaaS model, and is the most intuitive aspect that comes to the mind of people when referring to Platform-as-a-Service solutions. The runtime framework executes end-user code according to the policies set by the user and the provider.

**(b) 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.

**(c) Automation.** PaaS environment automates 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.

[22] 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.*

**(d) 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 in 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*. Different from Infrastructure-as-a-Service solutions, which deliver bare virtual server that can be fully customized in terms of the software stack installed, PaaS environment delivers a platform for developing applications, which exposes a well-defined set of APIs and, in most of the 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, in case the commitments made with the provider ceases. The impact of the vendor lock-in on applications obviously varies according to the different solutions. Some of them, such as *Force.com*, rely on a proprietary runtime framework, which make 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 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, while Infrastructure-as-a-Service solutions allow shifting the capital cost into operational costs through outsourcing, Platform-as-a-Service solutions can cut down the cost across development, deployment, and management of applications. It helps the management in reducing 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 that can concentrate their effort on the core value of their business. The Platform-as-a-Service approach, when bundled with a underlying IaaS solutions, helps even small startup companies to quickly offer to customers integrated solutions on a hosted platform, at a very minimal cost. These opportunities make the PaaS offering a viable option targeting different market segments.

## 4.2.4 Software as a Service

Software-as-a-Service (SaaS) is a software delivery model providing 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, who 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 upfront 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 Software-as-a-Service as a "one-to-many" software delivery model where an application is shared across multiple users. This is

the case of CRM[23] and ERP[24] applications that constitute common needs for almost all the enterprises from small, to medium 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 providing a general set of features and supporting specialization and ease of integrations of new components. Moreover, it constitutes the perfect candidate for hosted solutions, since the applications delivered to the user is the same, and the applications itself provide means to the users to shape itself according to their needs. As a result, SaaS applications are naturally multi-tenant. Multi-tenancy, 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 1.5), the concept of software as a service is precedent to Cloud computing and started to circulate at the end of 90s, 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 data center 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 applications services, started to gain interest and a new range of opportunities opened 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. Application Service Providers provided access to packaged software solutions that addressed the needs of different customers. Whereas initially this approach was affordable for service providers, 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 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.

---

[23] CRM is an acronym for Customer Relationship Management, and identify whatever concerns the interactions with customers and prospect sales. CRM solutions are software systems that simplify the process of managing customers and identifying effecting sales strategies.

[24] ERP is an acronym for Enterprise Resource Planning, and it generally refers to an integrated computer-based system used to manage internal and external resources including tangible assets, materials, 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.

How is Cloud computing related to SaaS? According to the classification by services showed in Fig. 2.2, 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) paramount
- Service level improvements
- Rapid implementation
- Stand-alone 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 platform driven by Service Level Agreements. 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 the business goals efficiently. Fundamental in this perspective is the ability of leveraging existing solutions and of integrating value-added business services. The existing SaaS infrastructures not only allow the development and the customization of applications but 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, which together provide customers with added value. This approach dramatically changes the software ecosystem of the SaaS market, which is not anymore monopolized by few vendors but is constituted by a fully interconnected network of service providers, clustered around some “big hubs” which deliver to the customer the application. In this scenario, each single component integrated in 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 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 that provides a ful-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 applications enriches the value of *SalesForce.com*. 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 a customer experience centered SaaS application that integrates together different features from chats to Web communities, to support the common activity of an enterprise.

Another important class of popular SaaS applications comprises of social networking applications such as *Facebook* and professional networking 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 the profile data of the user. The nature of these applications can be of different types: office automation components, games, and 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 aiming to address all the needs of users for documents, spreadsheets, and presentations 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 who provide an environment to integrate third-party services and share information with others. A quite successful example is *Box.net*, a 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 Web sites, 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 an infrastructure can be of different types, and provides 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 three different types of Clouds:

**(a) Public Clouds.** the Cloud is open to the wide public.

**(b) 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.

**(c) 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.

**(d) 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, we provide a brief characterization 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 where 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 constituted by 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 his/her credential and billing details, and use the services offered.

Historically, Public Clouds were the first class of Clouds 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, who are able to start their business without large upfront investments by completely relying on public infrastructure for their IT needs. What made attractive Public Cloud compared to the re-shaping of the private premises, and the purchase of hardware and software was the ability to grow or shrink according to the need 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 demand on their business. Currently, Public Clouds are used both to completely replace the IT infrastructure of enterprise and to extend it when it is required.

A fundamental characteristic of Public Clouds is multi-tenancy. A Public Cloud is meant to serve a multitude of users and not a single customer. Any customer requires its virtual computing environment that is separated, and most likely isolated, from the other users. This is a fundamental requirement to provide an effective monitoring of user activities, guarantee the desired performance, and the other Quality of Service attributes negotiated with users. QoS management is a very important aspect in Public Clouds. Hence, a significant portion of the software infrastructure is devoted to monitor the Cloud resources, to bill them according to the contract made with the user, and to keep a complete history of the Cloud usage for each customer. These features are fundamental for Public Clouds as they help providers to 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 providing infrastructure as a service, Google AppEngine is a Public Cloud providing an application development platform as a service, and Salesforce.com is a Public Cloud providing software as a service. What makes peculiar Public Clouds is the way in which 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 by geographically dispersed datacenters in order to share the load of users and better serve them according to their location. For example, Amazon Web Services has data centers installed in US and in Europe, and allow their customers to choose between three different regions: *us-west-1*, *us-east-1*, and *eu-west-1*. Such regions are priced differently and further divided into availability zones, which map to specific data centers. 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 down 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 Cloud, the provider is in control of the infrastructure and eventually of the customers' core logic and sensitive data. Even though there could be regulatory procedures in place that guarantee a fair management and the 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 like the 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 out of the law if processed with specific cryptographic techniques. For example, the USA Patriot Act[25] provides its government and other agencies with virtually limitless powers to access

[25] The US PATRIOT Act is a statute enacted by the United States Government that increases the ability of law enforcement agencies to search telephone, e-mail communications, and 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 Web site of the Library of the Congress at the following address: ***http://thomas.loc.gov/cgi-bin/bdquery/z?d107:hr03162:]*** (Accessed April 20, 2010).

information including that belonging to any company that stores information in the US 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 use the existing IT resources and optimize their revenue. All these aspects make the use of a public computing infrastructure not always possible. Yet, a 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 led to the diffusion of Private Clouds, which are similar to public Clouds, but its 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. Different from Public Clouds, instead of a pay-as-you-go model, there could be other schemes in place, which take into account the usage of the Cloud, and proportionally bill the different departments or sections of the enterprise. Private Clouds have the advantage of keeping in house the core business operations 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 since the Private Cloud can provide services to different range of users. Another interesting opportunity that comes with Private Clouds is the possibility of testing applications and systems at a comparatively lower 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:

**(a) 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 in their system. Security in-house is easier to maintain and to rely on.

**(b) Infrastructure Ensuring Service Level Agreements (SLAs).** Quality of service implies specific operations such as appropriate clustering and failover, data replication, system monitoring and maintenance, disaster recovery, and other uptime services can be commensurate to the application needs. While Public Clouds vendors provide some of these features, not all of them are available as needed.

**(c) 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 not be possible in the case of virtual public infrastructure.

All these aspects make the use of Cloud-based infrastructures within the 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, or an Enterprise Desktop Grid, or a combination of them. The physical layer is complemented with Infrastructure Management Software (i.e., IaaS (M), see Section 2.2.2) or a PaaS solution, according to the service delivered to the users of the Cloud.

Different options can be adopted to implement Private Clouds. Fig. 4.4 provides a comprehensive view of the solutions together with some reference with 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 [35], 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 a IaaS solution. While VMWare vCloud is a proprietary solution, Eucalyptus provides full compatibility with Amazon Web Services interfaces and supports different virtual machine technologies such as Xen, KVM, and VMWare. As Eucalyptus, OpenNebula is an open-source solution

for virtual infrastructure management supporting 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.

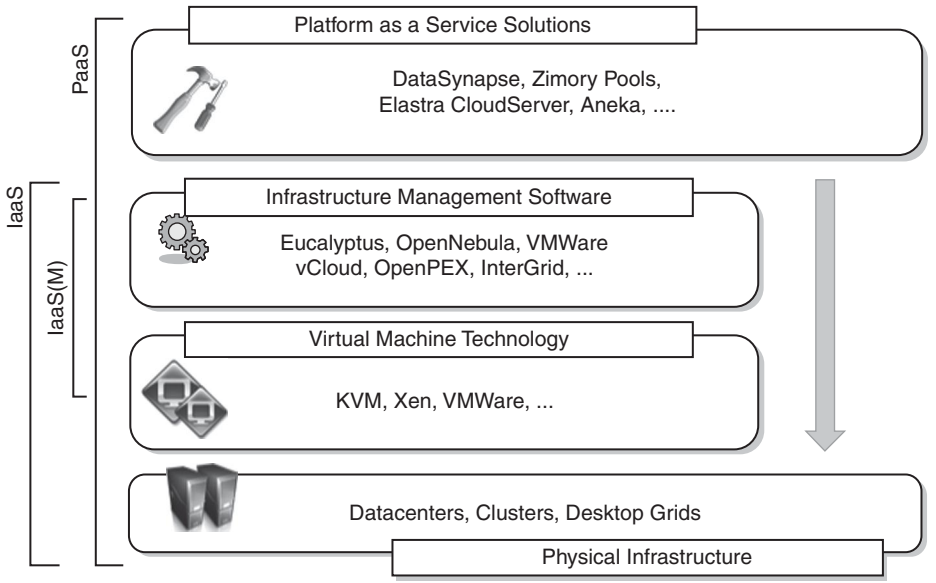


**Fig. 4.4.** Private Clouds—Hardware and Software Stack.

Solutions that rely on the previous virtual machine managers and provide added value are OpenPEX [40] and InterGrid [41]. OpenPEX is a Web-based system which 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 the OpenNebula and Amazon EC2 by allowing the reservation of virtual machine instances and managing a multi administrative domains Clouds. Platform-as-a-Service solutions can provide an additional layer and deliver a high-level service for Private Clouds. Among the different 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 provides a flexible environment for building Private Cloud 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 by 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, mainly devoted to support the execution of distributed applications with different programming models: bag of tasks, MapReduce, and others.

Although Private Clouds provide an in-house solution for Cloud computing, one of the major drawbacks of this solution is the inability to scale elastically on demand as Public Clouds do.

## 4.3.3 Hybrid Clouds

Public Clouds are large software and hardware infrastructures whose capability is huge enough to serve the needs of multiple users, but they suffer from security threats and administrative pitfalls. While the

option of completely relying on a public virtual infrastructure is appealing for companies that did not incur in IT capital costs and have just started considering their IT needs (i.e. start-ups), in most of the 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 the premises, or it is necessary to use the existing hardware and software infrastructure. One of the major lacks 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 on needed basis. Hence, a hybrid solution could be an interesting opportunity for taking advantage of both of the two worlds. This led to the development and the diffusion of Hybrid Clouds.

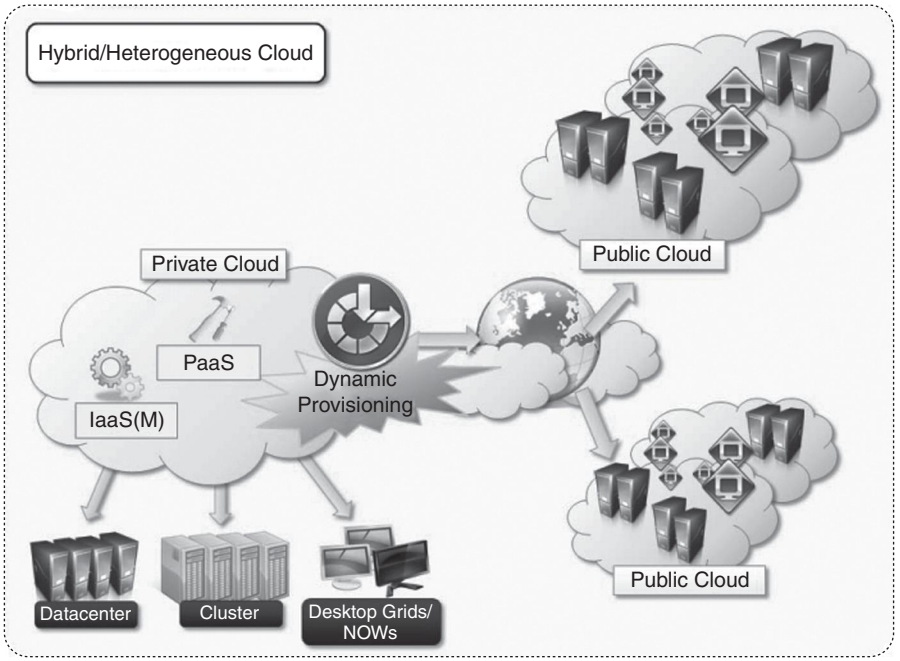


**Fig. 4.5.** Hybrid/Heterogeneous Cloud Overview.

Hybrid Clouds allow exploiting existing IT infrastructures, maintaining sensitive information within the premises, and naturally growing and shrinking by provisioning external resources and releasing them when 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 the system workload. Fig. 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 capacity demand. These resources or services are temporarily leased for the time required and then released. This practice is also known as *Cloud-bursting*[26].

[26] According to the Cloud Computing Wiki, the term "Cloudburst" has a double meaning and it also refers to the "failure of a Cloud computing environment due to the inability to handle a spike in demand". (*Reference: http://sites.google.com/site/Cloudcomputing-wiki/Home/Cloud-computing-vocabulary*). In this book, we will always refer to the dynamic provisioning of resources from Public Clouds when mentioning this term.

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. Within an IaaS scenario, dynamic provisioning refers to the ability of acquiring on demand virtual machines in order to increase the capability of the resulting distributed system and releasing 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 more complex scheduling algorithms and policies whose goal 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 manages all the other virtual machines instances maintained locally. What is missing is then an advanced scheduling engine able to differentiate these resources and providing 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 managing the allocation of virtual machines in a collection of peer networks. Such networks can be either 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. 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 supporting Hybrid Clouds. As previously discussed, one of the fundamental components of Platform-as-a-Service middleware is the mapping of distributed applications onto the Cloud infrastructure. In this scenario, the role of dynamic provisioning becomes fundamental for ensuring the execution of applications under the QoS agreed with the user. As an 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 Quality of Service for applications. In particular, each user application has a budget attached, and the scheduler uses such budget to optimize the execution of the application by renting virtual nodes if needed. Other Platform-as-a-Service 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.

## 4.3.4 Community Clouds

Community Clouds are distributed systems constituted by integrating the services of different Clouds to address the specific needs of an industry, a community, or a business sector. The 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."*

Fig. 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 if compared to Public Clouds, which serve a multitude of users with different needs. They are also different from Private Clouds where the services are generally delivered within the institution owning 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 organization, and even public virtual infrastructure providers, contribute with their resources to build the Cloud infrastructure.

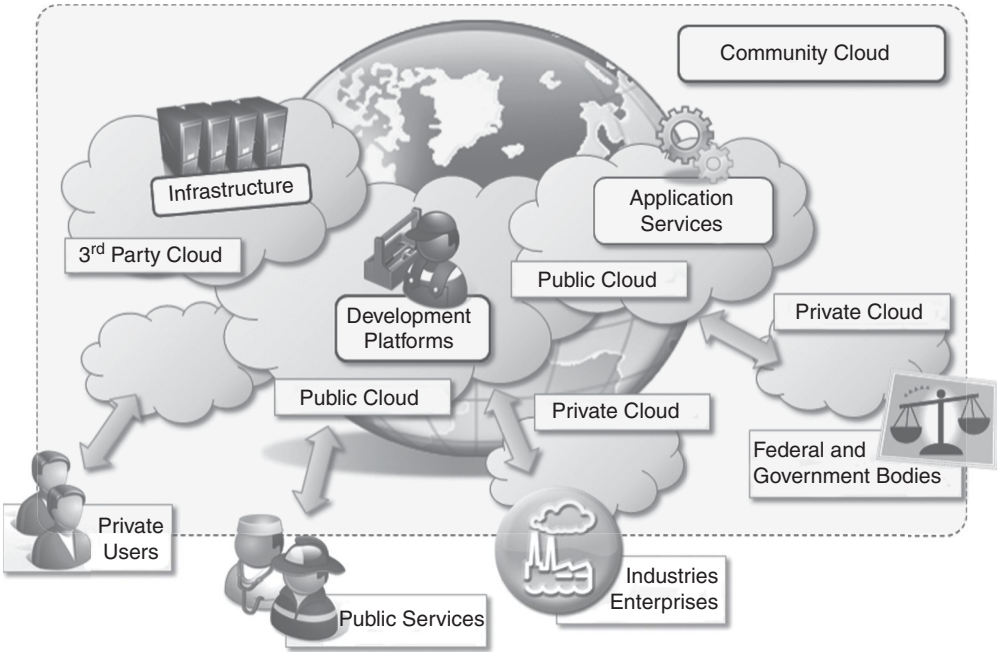


**Fig. 4.6.** Community Cloud.

Candidate sectors for Community Clouds described as above are the following:

**(a) Media Industry.** In the media industry, companies are looking for low-cost, agile, and simple solutions to improve the efficiency of content production. Most of the media productions involve an extended ecosystem of partners. In particular, the creation of digital content is the outcome of a collaborative process including movement of large data, massive compute-intensive rendering tasks, and complex workflows executions. Community Clouds can provide a shared environment where services can facilitate the business-to-business collaboration and offer the horsepower in term of aggregate bandwidth, CPU, and storage required to efficiently support media production.

**(b) Healthcare Industry.** Within the healthcare industry, there are different scenarios where Community Clouds could be of use. In particular, they can provide a global platform where to share information and knowledge without revealing sensitive data maintained within the private infrastructure. The naturally hybrid deployment model of Community Cloud can easily support the storing of patient-related data in a Private Cloud, while using the shared infrastructure to use non-critical services and automate processes within hospitals.

**(c) Energy and Other Core Industries.** In these sectors, Community Clouds can bundle together 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.

**(d) 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 to provide strategic solutions at local, national, and international administrative level.

They involve business-to-administration, citizen-to-administration, and possibly business-to-business processes. Some examples include: invoice approval, infrastructure planning, and public hearing. A Community Cloud can constitute the optimal venue to provide a distributed environment where to create a communication platform for performing such operations.

**(e) 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 Clouds, which arises from the concern over the controls of vendors in Cloud computing and aspires to combine the principles of Digital Ecosystems[27] [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 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:

**(f) Openness.** By removing the dependency on Cloud vendors, Community Clouds are open systems where a fair competition between different solutions can happen.

**(g) Community.** Being based on a collectivity providing resources and services, the infrastructure turns out to be more scalable, because the system can grow simply by expanding its user base.

**(h) Graceful Failures.** Since there is no single provider or vendor in control of the infrastructure, there is no single point of failure.

**(i) 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 operates all the decisions through a collective democratic process.

**(j) Environmental Sustainability.** The Community Cloud is supposed to have a smaller carbon footprint as they harness under-utilized resources. Moreover, they tend to be more organic by growing and shrinking in a symbiotic relationship to support the demand of the community, which in turns sustain it.

This is an alternative vision of Community Cloud, focusing more on the social aspect of the Community Clouds, which are formed as an aggregation of resources of the members of the community. 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 of the 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

[27] Digital Ecosystems are distributed, adaptive, and open socio-technical 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 SMEs (Small-Medium Enterprises).

- Replacing software licensing with subscriptions
- Cutting down 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 upfront and that will contribute over a long term to generate profit. The IT infrastructure and the software are capital assets, because enterprises do require them to conduct their business. At present time, it does not matter whether the principal business of an enterprise is related to IT because it 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 capital costs for any kind of enterprise. It is a good practice trying to keep the capital costs low because they introduce expenses that will generate profit over time. More than that, since they are associated to material things, they are subject to a *depreciation* over time, which in the end reduces the profit of the enterprise, since 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 which need to be replaced because new features are required.

Before Cloud computing diffused within the enterprise, the budget spent in IT infrastructure and software constituted a significant expense for medium and large enterprise. Many enterprises own a small or medium size datacenter, which introduce several operational costs in term of maintenance, electricity, and cooling. Additional operational costs are occurred in maintaining an IT department and IT support centers. 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 such model is that it shifts the capital costs previously located 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 on administrative and maintenance costs. That is, there is no or limited need for having administrative staff taking care of the management of Cloud infrastructure. At the same time, the cost of IT support staff is also reduced. When it comes to the 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 starting its business, 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 managing unplanned capital costs which 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 managing peak loads more efficiently 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 to them disappear. This is the most adopted model of Cloud computing since many enterprises already have IT facilities. Another option is to make a slow transition towards Cloud-based solutions while the capital IT assets get depreciated and need to be replaced. Among these two cases, there is a huge variety of scenarios in which Cloud computing could be of help in generating profit 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 emission. With Cloud computing, enterprise uses software applications on a subscription basis and there is no need of any licensing fee because the property of the software providing the service remains to the provider. Leveraging Infrastructure-as-a-Service solutions, allows room for datacenter consolidation that in the end could result into a smaller carbon footprint. In some countries such as Australia, the carbon-footprint emissions are taxable, by reducing or completely eliminating such emissions, enterprises can pay less taxes.
In terms of the pricing models introduced by Cloud computing, we can distinguish them in to three different strategies, that are adopted by the providers:

**(a) Tiered Pricing.** In this model, Cloud services are offered in several tiers, and each tier 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, speed, and memory) that have a different cost per hour.

**(b) Per-unit Pricing.** This model is more suitable in 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, customer can configure their systems more efficiently according to the application needs. This model is used, for example, by GoGrid where customers pay according to RAM/hour units for the servers deployed in the GoGrid Cloud.

**(c) Subscription-based Pricing.** This is the model used mostly by SaaS providers in which users are paying a periodic subscription fee for the usage 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 spent for 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

Being in its infancy, Cloud computing still presents a lot of challenges for the industry and the academia. There is a significant amount of work in the academia focused on defining the challenges brought by this phenomenon ([46], [47], [48], and [49]). In this section, we highlight the most important ones: the definition and the formalization, 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 in defining Cloud computing and in providing a classification of all the services and technologies identified as such. One of the most comprehensive formalization is noted in the working definition of Cloud computing given by National Institute of Standards and Technologies (NIST) [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 inline with our discussion and is shared by many IT practitioners and academics.

Despite the general agreement upon the NIST definition, there are alternative taxonomies for Cloud services. David Linthicum, founder of BlueMountains Labs, provides a more detailed classification[28], which comprehends ten different classes and better suits the vision of Cloud computing within the

[28] David Linthicum, Cloud Computing Ontology Framework. ***http://Cloudcomputing.sys-con.com/node/811519***

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 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 in providing a more robust interaction model between the different Cloud entities, on both the functional and the semantic level.

These characterizations and taxonomies reflect what is meant by Cloud computing at 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 notice 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 getting refined.

## 4.5.2 Cloud Interoperability and Standards

Cloud computing is a service-based model for delivering IT infrastructure and applications as utilities such as power, water, and electricity. In order to fully realize this, 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 a major fear for 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 costs and requires significant amount of time. This can occur either because the customer wants to find a more suitable solution for his/her needs or because the vendor is not able to provide the required service anymore. The presence of standards, which are actually implemented and adopted within the Cloud computing community, could give room for interoperability and then lessen the risks resulting from vendor lock-in.

At present time, the current state of standards and interoperability in Cloud computing resembles the early Internet era where there was no common agreement on the protocols and the technologies used, and each organization had its own network. Yet, the first steps towards a standardization process have been made and few organizations, such as the Cloud Computing Interoperability Forum (CCIF)[29], the Open Cloud Consortium[30], and the DMTF Cloud Standards Incubator[31], are leading the path. Another interesting initiative is the Open Cloud Manifesto[32], which embodies the point of view of different stakeholders about 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 Infrastructure-as-a-Service market, the use of a proprietary virtual machine format constitutes the major reasons of the vendor lock-in, and efforts in providing 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 the metadata describing a virtual machine imagine. Even though the OVF provides a full specification for packaging and distributing virtual machine images in a 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 complete 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

[29] ***http://www.Cloudforum.org***
[30] ***http://www.openCloudconsortium.org***
[31] ***http://www.dmtf.org/about/Cloud-incubator***
[32] ***http://www.openCloudmanifesto.org***

common set of APIs exposed makes the interaction with Cloud based solution vendor specific. In IaaS market, Amazon Web Services plays a leading role and other IaaS solutions, mostly open source, provide AWS compatible API, 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 done 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. In order 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 who develops, manages, and maintains the Cloud middleware and offers the service to customers. Within 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, which are easy to manage and at the same time, provide a competitive performance.

## 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 by using the most advanced technology in cryptography for protecting data, and then considered safe from any attempt to access it without the required permissions. While this data is processed in the memory, it has to be necessarily decrypted by the legitimate application, but since the application is hosted into a managed virtual environment, it becomes accessible to the virtual machine manager that by program is designed to access the memory pages of such 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 poses also 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 or not 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 by the primary Cloud service provider. In this case, there is a chain of responsibilities in the service delivery that can introduce more vulnerability for the secure management of data, the enforcement of the privacy rules, and the trust given to the service provider. In particular, when a violation of privacy or an illegal access to sensitive information is detected, it could become difficult to identify who is liable for such violations. The challenges within this area are then mostly concerned in devising secure and trustable system from different perspectives: technical, social, and legal.

## 4.5.5 Organizational Aspects

Cloud computing introduces a significant change in the way in which IT services are consumed and managed. More precisely, storage, compute power, network infrastructure and applications are de-

livered as metered services over the Internet. This introduces a new billing model which is new within typical enterprise IT departments, and this 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 when considering the role of the IT department in this new scenario. In particular, the following have to be considered:

- What is the new role of the IT department within an enterprise that completely or significantly relies on the Cloud?
- How compliance department will 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 by often 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 competencies and in general less skills, thus reducing their value. These are the challenges from an organizational point of view that have to be faced, and that will significantly change the relations within the enterprise itself among the different groups of people working together.

# Summary

In this chapter, we have 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 working definition of Cloud computing given by the National Institute of Standards and Technology (NIST), which enucleates 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 a rapid adoption of Cloud computing are the economics and the simplicity of software delivery and operation. It presents considerable opportunity to increase the profit 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 the cost models introduced with Cloud computing.

Although Cloud computing has been rapidly adopted in the industry, there are several open research challenges in areas such as management of Cloud computing systems, their security, and the social and organizational issues. There is significant room for advancement of 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 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 different types of Clouds.
11. Give an example of Public Cloud.
12. Which is the most common scenario for a Private Cloud?
13. What kind of needs is 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 applications and to cut down capital expenses?
16. List some of the challenges in Cloud computing.

# 5 Aneka: Cloud-Application Platform

Aneka is Manjrasoft's solution for developing, deploying, and managing Cloud applications. It 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 to monitor the status of the Cloud, and providing integration with existing Cloud technologies. One of the key advantages of Aneka is its extensible set of 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 Fig. 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 firstly describing the architecture of the system. It introduces the component and the fundamental services building up the Aneka Cloud and discusses some common deployment scenarios.

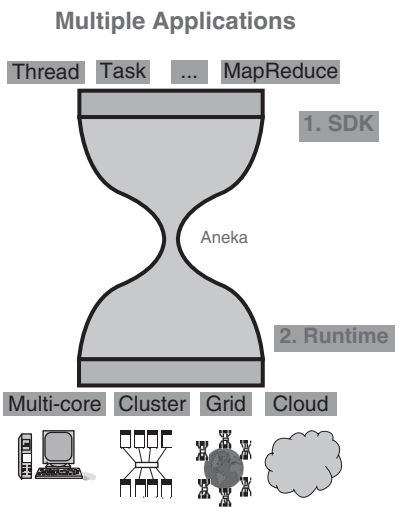


**Fig. 5.1.** Aneka Capabilities at a Glance.

# 5.1 FRAMEWORK OVERVIEW

Aneka is a software platform for developing Cloud computing applications. It allows harnessing disparate computing resources and managing them into a unique virtual domain—the Aneka Cloud—in which applications are executed. According to the Cloud Reference Model in previous chapter, Aneka is a *Pure PaaS* solution for Cloud computing. Aneka is a Cloud middleware that can be deployed on a heterogeneous set of resources: a network of computers, a multi-core server, data centers, virtual Cloud infrastructures, or a mixture of them. The framework provides both a middleware for managing and scaling distributing applications and an extensible set of APIs for developing them.

Figure 5.2 provides a complete overview of the components of the framework. The core infrastructure of the system provides a uniform layer allowing the framework to be deployed over different platform 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 developers. The container features three different classes of services: *Fabric Services, Foundation Services*, and *Execution Services*. These respectively take care of infrastructure management, supporting services for the Cloud, and application management and execution. These services are made available to developers and administrators by the 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.

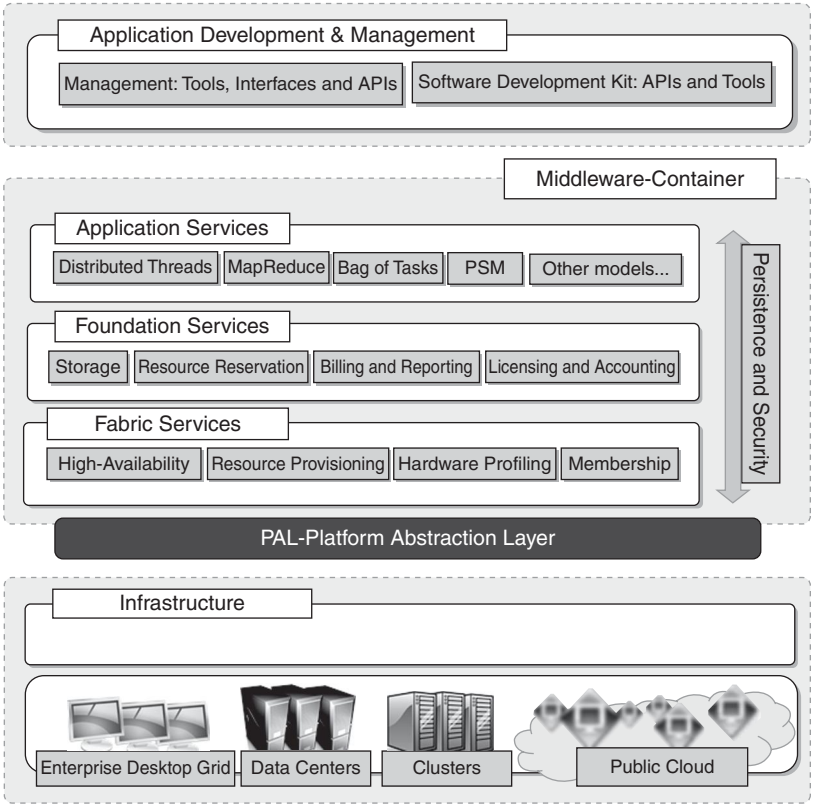


**Fig. 5.2.** Aneka Framework Overview.

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

**(a) Elasticity and Scaling.** With its dynamic provisioning service, Aneka supports dynamically up-sizing and down-sizing of the infrastructure available for applications.

**(b) 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 managing service membership and lookup, infrastructure maintenance, and profiling.

**(c) Resource Management.** Aneka is an elastic infrastructure where resources are added and removed dynamically, according to the application needs and user requirements. In order 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.

**(d) Application Management.** A specific subset of services is devoted to manage applications: these services include scheduling, execution, monitoring, and storage management.

**(e) User Management.** Aneka is a multi-tenant distributed environment where multiple applications, potentially belonging to different users, are executed. The framework provides an extensible user system where 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 the accounting management.

**(f) QoS/SLA Management and Billing.** Within a Cloud environment, application execution is metered and billed. Aneka provides a collection of services that coordinate together for taking into account the usage of resources by each application and billing the owning user accordingly.

All these services are available to specific interfaces and APIs, on top of which the software development kit 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 its status, while the software development kit is more focused on the single application, and provide means to control its execution from a single user. Both of them are meant to provide an easy-to-use interface 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 block of Aneka Clouds and represents the runtime machinery available to services and applications. The container is the unit of deployment in Aneka Clouds, and it 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 where to deploy services and some basic services such as communication channels for interaction 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 towards 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, we discuss in detail the components in each of these layers.

## 5.2.1 From the Ground Up: 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 organization of the file system 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 platform supported.

The PAL is responsible for detecting the supported hosting environment, and providing the corresponding to interact with it for supporting the activity of the container. It 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 comprising 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 system.

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. As an example, these properties can contain the additional information about the processor such as the model, 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 allows for hardware profiling and implement a basic monitoring infrastructure that can be used by all the services installed in the container.

### *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, while 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 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 reservation and scheduling in order to optimize the usage of a heterogeneous infrastructure. As already discussed, the basic information about memory, disk space, CPU and operating system are collected. In addition, additional data are pulled into the "alive" message such as 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 said, 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 components in order to cover a wide variety of hosting environments. While different operating systems are supported with different implementations of the PAL, different node resolvers allow Aneka to capture other types of data which 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, 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 reports its activity. This infrastructure is composed by the *Reporting* and the *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 all the monitored data that has been collected on the node to it. Any service wanting 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 the information about data transfer, such as upload and download times, file names, and sizes.
- The *Resource Provisioning Service* tracks the provisioning and lifetime information of virtual nodes.

All these information can be stored on RDBMS or a flat file, and they can be further analyzed by specific applications. For example, the management console provides a view on such data for administrative purposes.

## 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: *Index Service* (or *Membership Catalogue*), *Reservation Service*, and *Resource Provisioning Service*.

The *Membership Catalogue* is the fundamental component for resource management since 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 by 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 in order to discover the available services and interact with them. In order to speed up and enhance the performance of queries, the membership catalogue works is organized as a distributed database. All the queries that pertain 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 is then sent to the local monitoring service to be persisted on the long term.

Indexing and categorizing resources is fundamental for 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 is performed by the infrastructure management layer, and it is not part of the fabric services.

Dynamic resource provisioning allows the integration and the 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 where 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 in 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 *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 Infrastructure as a Service 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 Quality of Service (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 the resource provisioning capabilities of Aneka by dynamically retrieving a client to the service and interacting with the infrastructure. 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 in order 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 for 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, they 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.

# 1. Storage Management

The management of data is an important aspect in 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 supporting 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 a 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 a more scalable solution.

The centralized storage is implemented through, and managed by, the *Storage Service*. The service constitutes the data staging facilities of Aneka. 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 FTP. In order to support different protocols, the system introduces the concept of *file channel* that identifies a pair of components: a file channel controller and a file channel handler. The former constitutes the server component of the channel, where files are stored and made available, the latter 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 creating 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 of chaining two different channels to move files by using two different protocols. Each file in Aneka contains metadata helping the infrastructure to 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 by using the internal FTP protocol, and then 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*, and *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 where to store files. Files are logically organized into a directory structure, but persisted on the file system by 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 a unique id and stored on different servers, eventually replicated in order to provide high availability and failure tolerance. The model proposed by the Google File System provides an 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 kind of major workloads: large streaming reads and small random reads.
- It is more important to have a sustained bandwidth rather 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 of the Google File System implementation. Aneka provides a simple DFS (distributed file system), which relies on the file system services of the Windows operating system.

## 2. Accounting, Billing, and Resource Pricing

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

The information collected for accounting is primarily related to the usage of the infrastructure and the execution of applications. A complete history of application execution and storage as well as other resource utilization parameter are captured and minded by the accounting services. This information constitutes the foundation on top of which users are charged in Aneka.

Billing is another important feature of accounting. Aneka is a multi-tenant Cloud programming platform where the execution of applications can involve provisioning additional resources from commercial IaaS providers. Aneka billing service provides detailed information about the resource usage of each user with the associated costs. Each resource can be priced differently according to the different 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 on 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 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.

## 3. Resource Reservation

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 Allocation Service. The former keeps track of all the reserved time slots in the Aneka Cloud and provides a unified view of the system, while the latter is installed on each node featuring 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 time frame. If it is possible to satisfy the

request, the *Reservation Service* will return a reservation identifier as a 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 the 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 complete transparent manner and Aneka provides extensible API for supporting advanced services. At the moment, the framework supports three different implementations:

**(a) Basic Reservation.** It features the basic capability of reserving execution slots on nodes and implements the *alternate offers* protocol, which provides alternative options in case the initial reservation requests cannot be satisfied.

**(b) Libra Reservation.** It represents a variation of the previous implementation that features the ability of pricing nodes differently according to their hardware capabilities.

**(c) Relay Reservation.** It constitutes a very thin implementation that allows the resource broker to reserve nodes in Aneka Clouds and control the logic with which these nodes are reserved. This implementation is useful in integration scenario where Aneka operates in an inter-Cloud environment.

Resource reservation is fundamental for ensuring the Quality of Service that is negotiated for applications. It allows having a predictable environment where applications can complete within the deadline or not be executed at all. The assumptions made by the reservation service for accepting reservation request 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 the delay to allocation of nodes to reservation requests as late as possible, in order to cope with temporary failures or limited outages, but in case of serious outages where 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 service level agreement defined for applications. The current implementation of the resource reservation infrastructure leverage 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 both the problem of 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: *Scheduling Service* and *Execution Service*. Moreover, it also defines base implementations that can be extended in order to integrate new models.

### *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 are mostly belonging to the fabric and the foundation layers of the architecture shown in Fig. 5.2. The possibility of having different scheduling engines for different models gives a great freedom in implementing scheduling and resource allocation strategies but, at the same time, requires a careful design on how to use shared resources. In this scenario, common situation that have to be appropriately managed are the following: multiple jobs sent to the same node at the same time; jobs without reservation sent to reserved nodes; and jobs sent to nodes where the required services are not installed. Foundation services of Aneka 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 separated 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.

## 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 the 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*, the local *Allocation* and *Monitoring* Services. Aneka provides a reference implementation of execution services that has a built-in integration with all these services and currently two of the supported programming models specialize the reference implementation.

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

**(a) Task Model.** This model provides the support for independent bag of tasks applications and many tasks computing. 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.

**(b) Thread Model.** This model provides an extension to the classical multi-threaded programming to a distributed infrastructure and uses the abstraction of *Thread* to wrap a method that is executed remotely.

**(c) MapReduce Model.** This is an implementation of MapReduce as proposed by Google on top of Aneka.

**(d) 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 combination 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], *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 to be deployed on, which needs to be managed. Infrastructure management tools are specifically designed for this task, and building Clouds is one of the primary tasks of administrators. Different deployments models for Public, Private and Hybrid Clouds are supported.

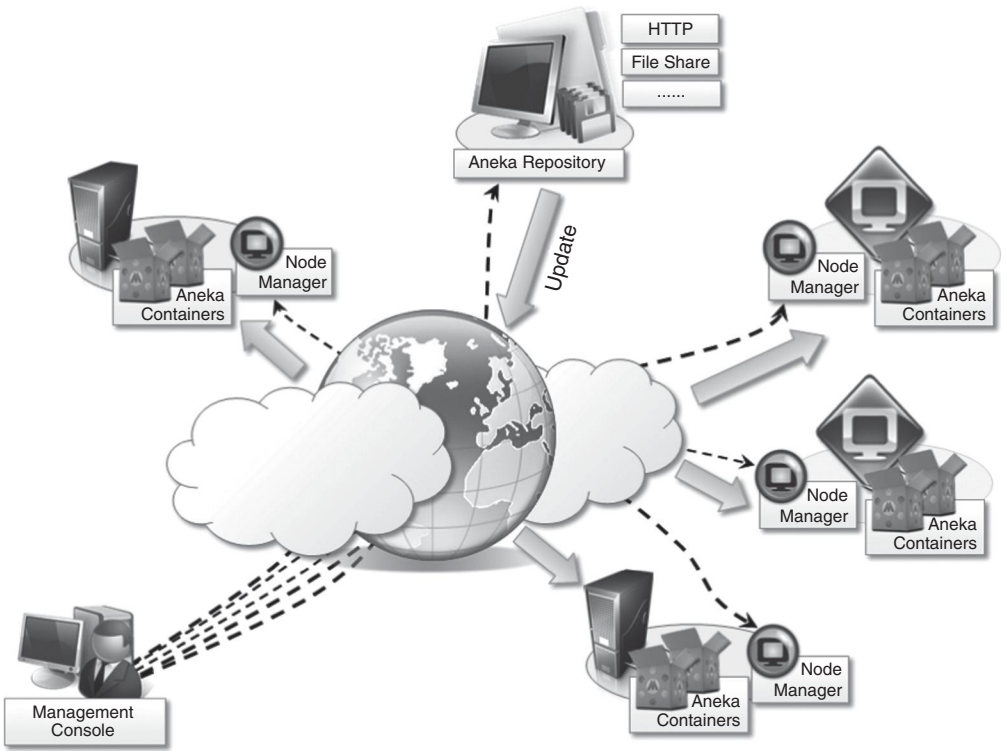


**Fig. 5.3.** Aneka Cloud Infrastructure Overview.

## 5.3.1 Infrastructure Organization

Fig. 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 supported by Aneka. A central role is played by the administrative console that performs all the required management operations. A fundamental element

for Cloud deployment is constituted by *repositories*. A repository provides storage for all the libraries required to layout and to 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 different communication channels, such as HTTP, FTP, and common file share. The management console can manage multiple repositories and select the one which 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 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 needs 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 scenario is using a master-worker configuration with separated nodes for storage as shown in Fig. 5.4.

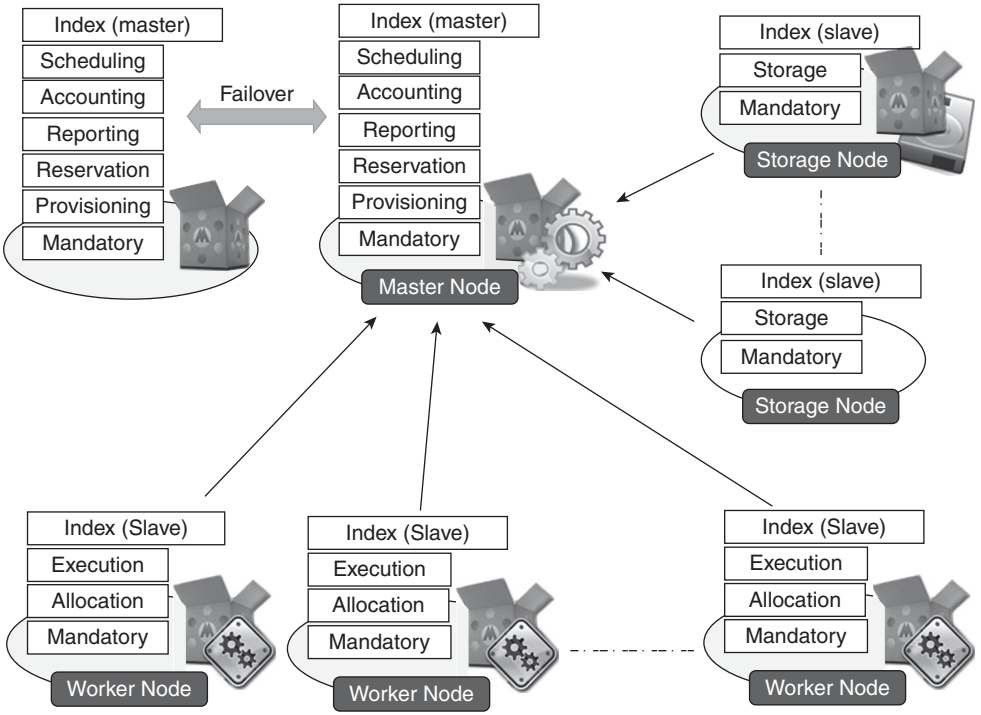


**Fig. 5.4.** Logical Organization of an Aneka Cloud.

The master node features all the services that are most likely to be present in one single copy and provide the intelligence of the Cloud. What specifically characterizes the node as a master node is the

presence of the *index service* (or membership catalogue) configured in the master mode. All the other services, except for those who are mandatory, might be present or located in other nodes. A common configuration of the master node is the following:

- *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*

The master node also provides connection to a 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 normally persists on the RDBMS in order to provide a fault-tolerant infrastructure. The master configuration can then be replicated in several nodes, in order to provide a highly available infrastructure based on the fail over 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 model 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 for providing 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 made by applications. Storage nodes mostly reside on machines that have a considerable disk space in order 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 case of a high-available 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 as desktop machines,

clusters, or workstations. These resources can be partitioned into different groups and Aneka can be configured to leverage these resources according to the need of applications. Moreover, by 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 the case of a Private Cloud. This deployment is acceptable for a scenario where the workload of the system is predictable and the capacity demand in excess can be easily addressed by a local virtual machine manager. Most of the Aneka nodes are constituted by physical nodes having 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 and 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.

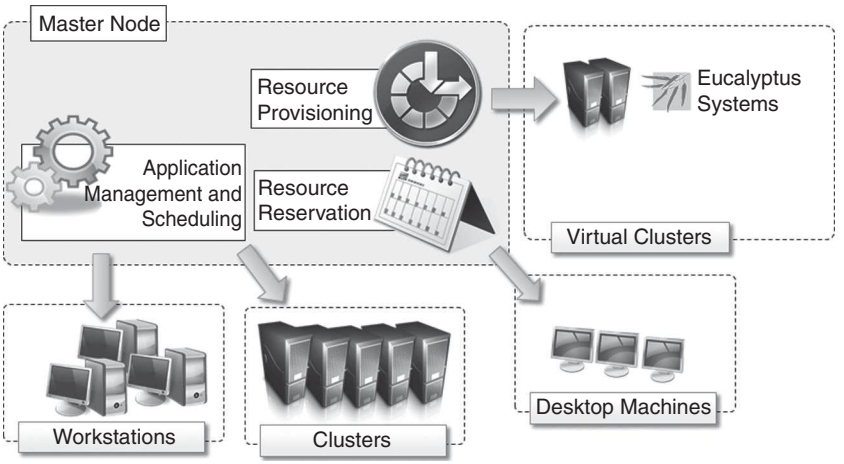


**Fig. 5.5.** Private Cloud Deployment.

## 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 providers such as Amazon EC2 or GoGrid. In this case, it is possible to have both a static deployment where the nodes are provisioned beforehand and used as if 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 reason for this is to minimize the data transfer between different providers, which is generally priced at a higher cost, and have better network performance. In this scenario, it is possible to deploy 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 multi-tenant Cloud where users are billed accordingly to their consumption of Cloud capabilities.

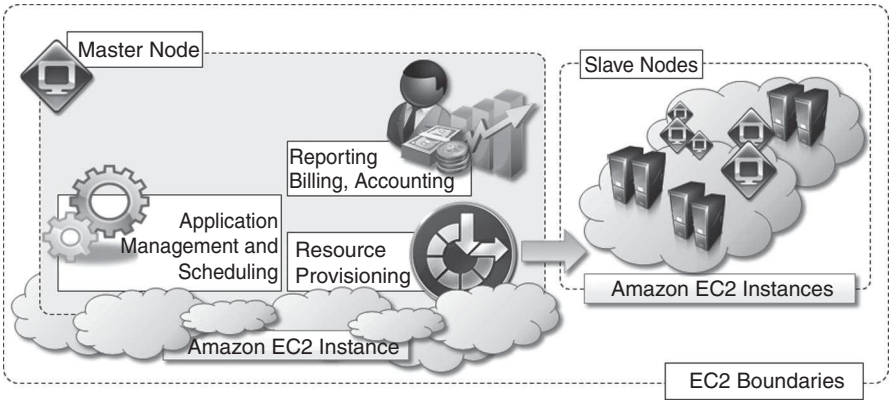


**Fig. 5.6.** Public Cloud Deployment.

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 cases of multi-tenant Clouds, multiple applications can leverage the support for storage. In this scenario, it is then possible to introduce bottlenecks or simply reach the storage quota limits allocated for the storage on the node. Dynamic provisioning can easily solve this issue as it does for increasing the computing capability of Aneka Cloud.

Deployments using different providers are unlikely to happen because of the data transfer costs among providers, but there might be a possible scenario for federated 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

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 the deployment is presented in Fig. 5.7.

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

- *Dynamic resource provisioning*
- *Resource reservation*
- *Workload partitioning*
- *Accounting, monitoring, and reporting*

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 already discussed in case of a Private Cloud deployment, desktop machines can be reserved for low priority

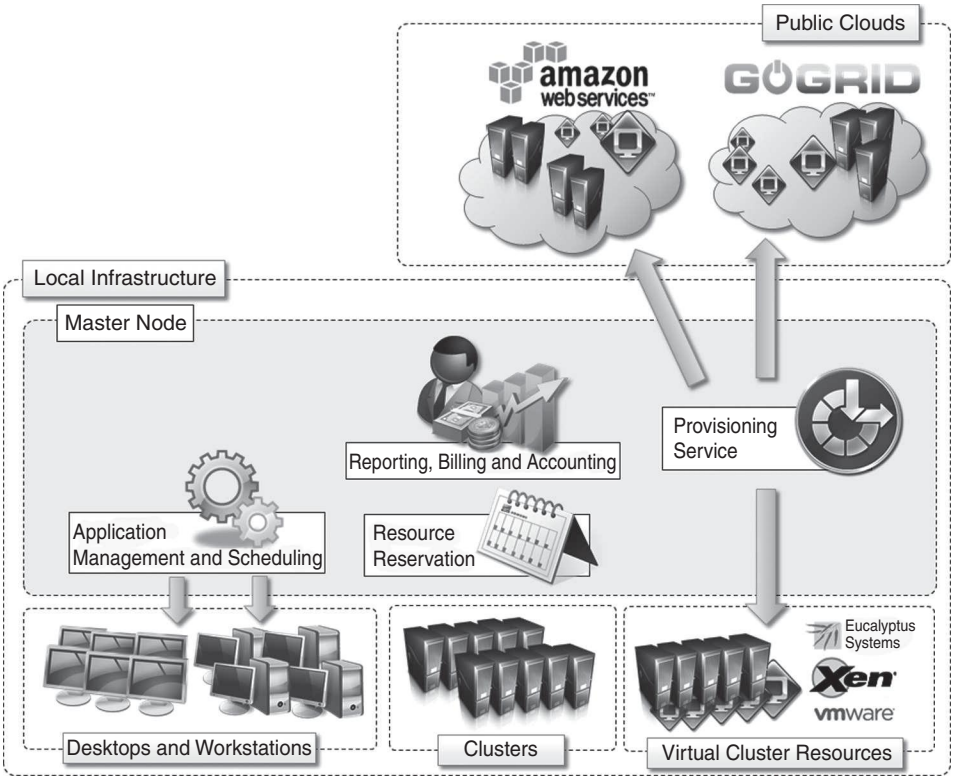


**Fig. 5.7.** Hybrid Cloud Deployment.

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 Public Cloud deployment, this is the case in which it makes more sense to leverage a variety of resource providers in order to provision virtual resources. Since part of the infrastructure is local, there exists a cost in data transfer towards the external IaaS infrastructure that cannot be avoided. It is, then, important to select the most suitable option to address the needs of applications. The *Resource Provisioning Service* implemented in Aneka exposes the capability of leveraging several resource pools at the same time and to configure 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

The primary purpose of Aneka is to provide a scalable middleware where to execute distributed applications. Application development and management constitute the two major features that are exposed to developers and system administrators. In order 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 while 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, while the latter defines the general infrastructure for service development.

### *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 defining the Aneka Application Model is shown in Fig. 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 the application manager used to control the application. Application classes constitute the view that developers have of a distributed application on Aneka Clouds, while application managers are internal components that interact with Aneka Cloud, 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.

Despite the specific model that is used, a distributed application can be conceived as a set of tasks whose collective execution defines the execution of the application on the Cloud. Aneka further specializes applications into two main categories: (i) applications whose tasks are generated by the user, and (ii) 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: *Task Model*, *Thread Model*, and *Parameter Sweep Model*. Applications falling in this category are composed by 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 whose transfer 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 in implementing the *IManualApplicationManager* interface.

The second category covers the case of *MapReduce* and all those other scenarios in which the unit 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 depends on the requirements of the programming model used. For example, in the case of 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 complete the object model. Among them, 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.

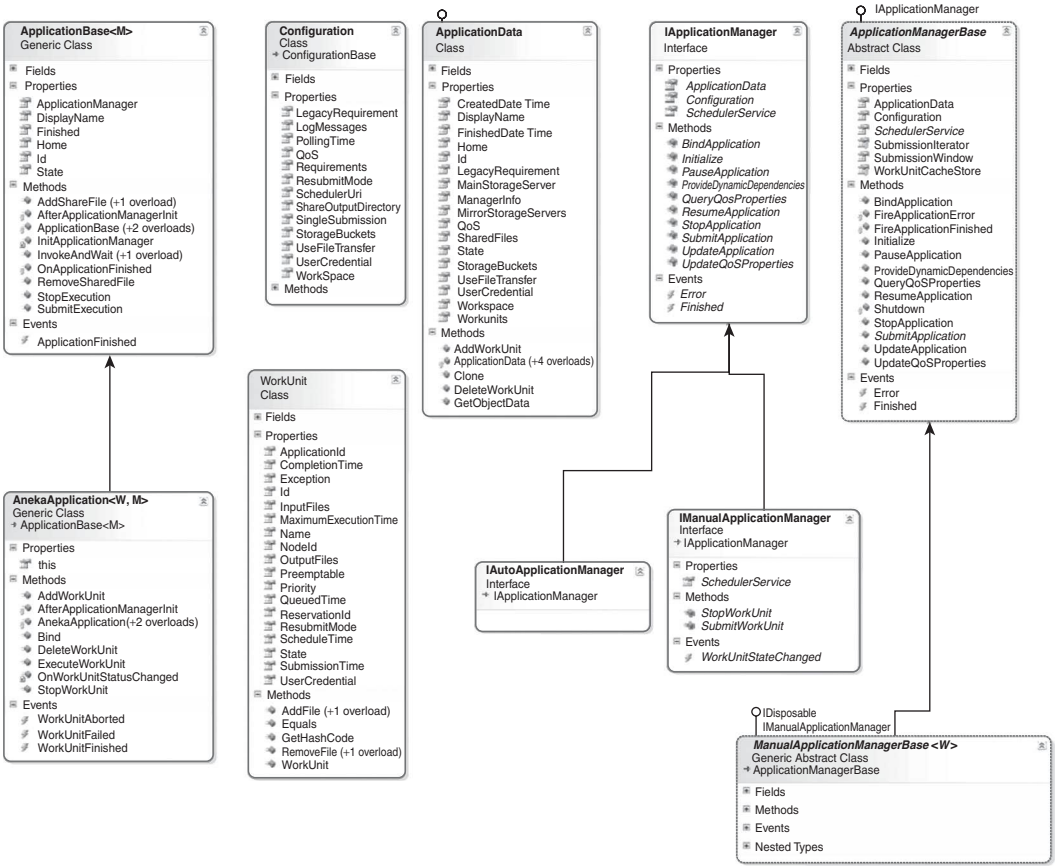


**Fig.5.8.** Aneka Application Model.

Table 5.1 summarizes the features that are available in the Aneka Application Model and how 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 of (or specializing) an existing implementation of a programming model or by using the base classes to define new models and abstractions. As an 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 the end users who just need to define a template task and the parameters that customize it.

**Table.5.1.** Application Model.

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

## 2. Service Model

The Aneka Service Model defines the basic requirements needed to implement a service that can be hosted in the Aneka Cloud. The container defines the runtime environment where 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 help clients if they are meant to directly interact with the end users. Examples of such services might be *Resource Provisioning* and *Resource Reservation* services, which ship their own client 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 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 within the Aneka container. The shaded balloons indicate transient states while the white ones indicate steady states. A service instance can initially be in the *Unknown* or *Initialized* state, this condition 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 *Unknown* state, thus signaling an error. When running, it is possible to pause the activity of a service by calling the *Pause* method and resuming 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 into the *Running* state. Not all the services are needed 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 in order to design and implement services that are compliant with Aneka. In particular, it defines a *ServiceBase* class that can be further extended to 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, where each service defines its own messages, which are, in turn, the only ones that it is able to process. As a result, developers that 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.

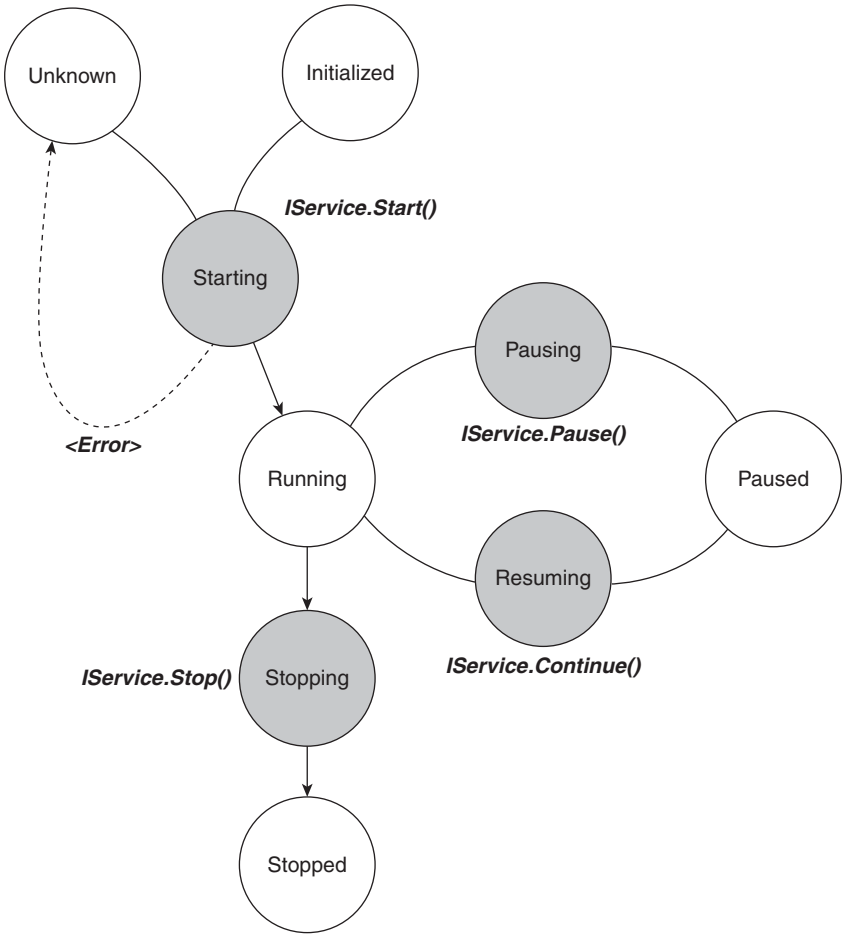


**Fig. 5.9.** Service Life Cycle.

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 feature 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 an application 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 the support for the execution of distributed applications or require access to additional services.

Aneka also provides advanced capabilities for services configuration. Developers can define editors and configuration classes that allow the management tools of Aneka 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 the management layer of Aneka. This layer also includes capabilities for managing services and applications running in the Aneka Cloud.

## 1. Infrastructure Management

Aneka leverages virtual and physical hardware in order to deploy 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 the Aneka on the hardware.

## 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, allowing 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.

## 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 where users are billed for their usage of resources. 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 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 the framework is its configurable runtime environment that 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: monitoring, resource provisioning, hardware profiling, and membership
- foundation services: storage, resource reservation, billing, accounting, and reporting
- application services: scheduling and execution

From an application programming point of view, Aneka provides the capability of supporting different programming models, which allow developers to express distributed applications with different abstractions. The framework currently supports three different models: independent bag of tasks applications, multi-threaded applications, and map reduce.

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 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 the resource provisioning capabilities of Aneka.
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 give for development?
15. Discuss the major features of the Aneka Application Model.
16. Discuss the major features of the Aneka Service Model.
17. Descrise the features of the Aneka management tools in terms of infrastructure, platform, and applications.

# 8 Data-Intensive Computing: Map Reduce Programming

Data-intensive computing focuses on a class of applications that deal with a large amount of data. Several application fields, ranging from computational science to social networking, produce large volumes of data that need to be efficiently stored, made accessible, indexed, and analyzed. These tasks become challenging as the quantity of information accumulates and increases over time at higher rates. Distributed computing is definitely of help in addressing these challenges by providing more scalable and efficient storage architectures and a better performance in terms of data computation and processing. Despite this, the use of parallel and distributed techniques as a support of data intensive computing is not straightforward, but several challenges in the form of data representation, efficient algorithms, and scalable infrastructures need to be faced.

This chapter characterizes the nature of data-intensive computing and presents an overview of the challenges introduced by production of large volumes of data, and how they are handled by storage systems and computing models. It describes *MapReduce*, which is a popular programming model for creating data intensive applications and their deployment on Clouds. Practical examples of MapReduce applications for data-intensive computing are demonstrated by using *Aneka MapReduce Programming Model.*

## 8.1 WHAT IS DATA-INTENSIVE COMPUTING?

Data-intensive computing is concerned with production, manipulation, and analysis of large-scale data in the range of hundreds of megabytes (MB) to petabytes (PB) and beyond [73]. The term "*dataset"* is commonly used to identify a collection of information elements that is relevant to one or more applications. Datasets are often maintained in *repositories*, which are infrastructures supporting the storage, retrieval, and indexing of large amount of information. In order to facilitate the classification and the search of relevant bits of information, *metadata*, are attached to datasets.

Data-intensive computations occur in many application domains. Computational science is one of the most popular ones. Scientific simulations and experiments are often keen to produce, analyze, and process huge volumes of data. Hundreds of gigabytes of data are produced every second by telescopes mapping the sky, and the collection of images of the sky easily reaches the scale of petabytes over a year. Bioinformatics applications mine databases that may end up containing terabytes of data. Earthquake simulators process a massive amount of data, which is produced as a result of recording the vibrations of the Earth across the entire globe.

Besides scientific computing, several IT industry sectors require support for data-intensive computations. A customer data of any telecom company would easily be in the range of 10–100 terabytes. This volume of information is not only processed to generate bill statements but also mined to identify scenarios, trends, and patterns helping these companies to provide a better service. Moreover, it is reported that currently the US handset mobile traffic has reached 8 petabytes per month, and it is expected to grow up to 327 petabytes per month by 2015[41]. The scale of petabytes is even more common when considering IT giants, such as Google, which is reported to process about 24 petabytes of information per day [55] and to sort petabytes of data in hours[42]. Social networking and gaming are two other sectors where data-intensive computing is now a reality. *Facebook* inbox search operations involve crawling about 150 terabytes of data, and the whole uncompressed data stored by the distributed infrastructure reaches to 36 petabytes[43]. *Zynga*, social gaming platform, moves 1 petabyte of data daily, and it has been reported to add 1000 servers every week to sustain and store the data generated by games like Farmville and Frontierville[44].

## 8.1.1 Characterizing Data-Intensive Computations

Data-intensive applications do not only deal with huge volumes of data but, very often, also exhibit compute-intensive properties [74]. Figure 8.1 identifies the domain of data-intensive computing in the two upper quadrants of the graph.

Data-intensive applications handle datasets in the scale of multiple terabytes and petabytes. Datasets are commonly persisted in several formats and distributed across different locations. Such applications process data in multistep analytical pipelines including transformation and fusion stages. The processing requirements scale almost linearly with the data size, and they can be easily processed in parallel. They also need efficient mechanisms for data management, filtering and fusion, and efficient querying and distribution [74].

## 8.1.2 Challenges Ahead

The huge amount of data produced, analyzed, or stored imposes requirements on the supporting infrastructures and middleware that are hardly found in the traditional solutions for distributed computing. For example, the location of data is crucial, as the need for moving terabytes of data becomes an obstacle for high-performing computations. Data partitioning, as well as content replication and scalable algorithms help in improving the performance of data-intensive applications. Open challenges in data-intensive computing given by Ian Gorton et al. [74] are

- Scalable algorithms that can search and process massive datasets
- New metadata management technologies that can scale to handle complex, heterogeneous, and distributed data sources
- Advances in high-performance computing platform aimed at providing a better support for accessing in-memory multi-terabyte data structures
- High-performance, high-reliable, petascale distributed file systems
- Data signature generation techniques for data reduction and rapid processing
- New approaches to software mobility for delivering algorithms able to move the computation where the data is located
- Specialized hybrid interconnection architectures providing a better support for filtering multi-gigabyte data streams coming from high speed networks and scientific instruments
- Flexible and high-performance software integration techniques facilitating the combination of software modules running on different platform to quickly form analytical pipelines.

---

[41] Reference: Coda Research Consultancy, ***http://www.codaresearch.co.uk/usmobileinternet/index.htm.***

[42] Reference: Google's Blog, ***http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html***.

[43] OSCON 2010, David Recordon (Senior Open Programs Manager, Facebook): Today's LAMP Stack, Keynote Speech, Available at: ***http://www.oscon.com/oscon2010/public/schedule/speaker/2442.***

[44] Reference: ***http://techcrunch.com/2010/09/22/zynga-moves-1-petabyte-of-data-daily-adds-1000-servers-a-week/.***

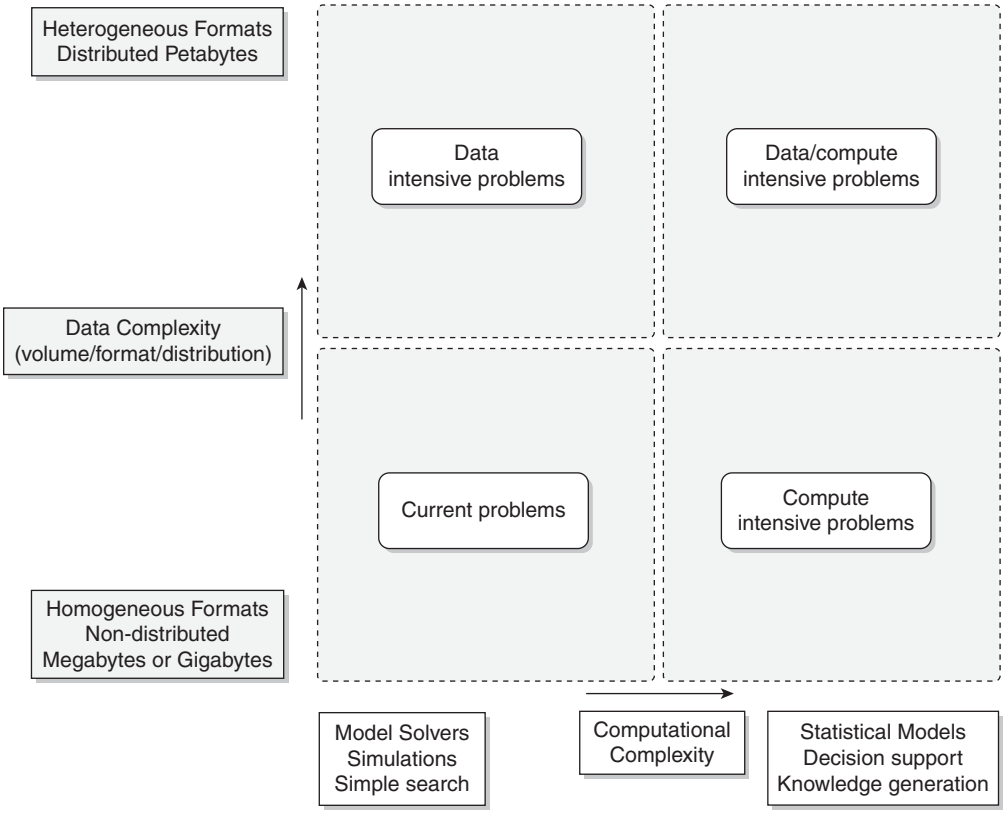


**Fig. 8.1.** Data-Intensive Research Issues.

## 8.1.3 Historical Perspective

Data-intensive computing involves the production, management, and analysis of large volumes of data. Support for data-intensive computations is provided by harnessing storage, networking technologies, algorithms, and infrastructure software all together. We track the evolution of this phenomenon by highlighting the most relevant contributions in the area of storage and networking, and infrastructure software.

### *1. The Early Age: High-Speed Wide Area Networking*

The evolution of technologies, protocols, and algorithms for data transmission and streaming has been an enabler of data-intensive computations [75]. In 1989, the first experiments in high-speed networking as a support of remote visualization of scientific data led the way. Two years later, the potential of using high-speed wide-area networks for enabling high-speed TCP/IP based distributed applications was demonstrated at Supercomputing 1991 (SC91). In that occasion, the remote visualization of large and complex scientific datasets (a high-resolution of MRI scan of the human brain) was set up between the *Pittsburgh Supercomputing Center (PSC)* and Albuquerque, the location of the conference.

A further step was made by the *Kaiser* project [76], which made available as remote data sources high data rate and online instrument systems. The project leveraged the *Wide Area Large Data Object (WALDO)* system [77], which was used to provide the following capabilities: automatic generation of

metadata; automatic cataloguing of data and metadata while processing the data in real time; facilitation of cooperative research by providing data access to local and remote users; and mechanisms to incorporate data into databases and other documents.

The first data-intensive environment is reported to be the *MAGIC* project—a *DARPA* funded collaboration working on distributed applications in large scale, high-speed networks. Within this context, the *Distributed Parallel Storage Systems (DPSS)* was developed, which was later used to support *TerraVision* [78]—a terrain visualization application that lets users explore/navigate a tridimensional real landscape.

Another important milestone was set with the Clipper project[45], which is a collaborative effort of several scientific research laboratories with the goal of designing and implementing a collection of independent but architecturally consistent service components to support data-intensive computing. The challenges addressed by the clipper project included: management of substantial computing resources; generation or consumption of high rate and high-volume data flows; human interaction management; and aggregation of disperse resources (multiple data archives, distributed computing capacity, distributed cache capacity, and guaranteed network capacity). The main focus of Clipper was to develop a coordinated collection of services that can be used by a variety of applications to build on-demand, large-scale, high-performance, wide-area problem-solving environments.

## 2. Data Grids

With the advent of Grid Computing [8], huge computational power and storage facilities could be obtained by harnessing heterogeneous resources across different administrative domains. Within this context, *Data Grids* [79] emerge as infrastructures supporting data-intensive computing. A Data Grid provides services helping users to discover, transfer, and manipulate large datasets stored in distributed repositories and, also, create and manage copies of them. Data Grids offer two main functionalities: high-performance and reliable file transfer for moving large amounts of data; and scalable replica discovery and management mechanisms for an easy access to distributed datasets [80]. As they span across different administration boundaries, access control and security are important concerns.

Data Grids mostly provide storage and dataset management facilities as support of scientific experiments that produce huge volumes of data. The reference scenario might be one depicted in Fig. 8.2. Huge amounts of data are produced by scientific instruments (telescopes, particle accelerators, etc.). The information, which can be locally processed, is then stored in repositories, and made available for experiments and analysis to scientists who can be local or most likely remote. Scientists can leverage specific discovery and information services, which helps in determining where the closest datasets of interest for their experiments are located. Datasets are replicated by the infrastructure in order to provide a better availability. Since processing of this information also requires a large computation power, specific computing sites can be accessed to perform analysis and experiments.

As any other Grid infrastructure, heterogeneity of resources and different administrative domains constitute a fundamental aspect that needs to be properly addressed with security measures and the use of *Virtual Organizations (VO)*. Beside heterogeneity and security, Data Grids have their own characteristics and introduce new challenges [79]:

**(a) Massive Datasets.** The size of datasets can easily be in the scale of gigabytes, terabytes, and beyond. It is, therefore, necessary to minimize latencies during bulk transfers, replicate content with appropriate strategies, and manage storage resources.

**(b) Shared Data Collections.** Resource sharing includes distributed collections of data. For example, repositories can be used to both store and read data.

[45] Reference: ***http://www.nersc.gov/news/annual_reports/annrep98/16clipper.html.***

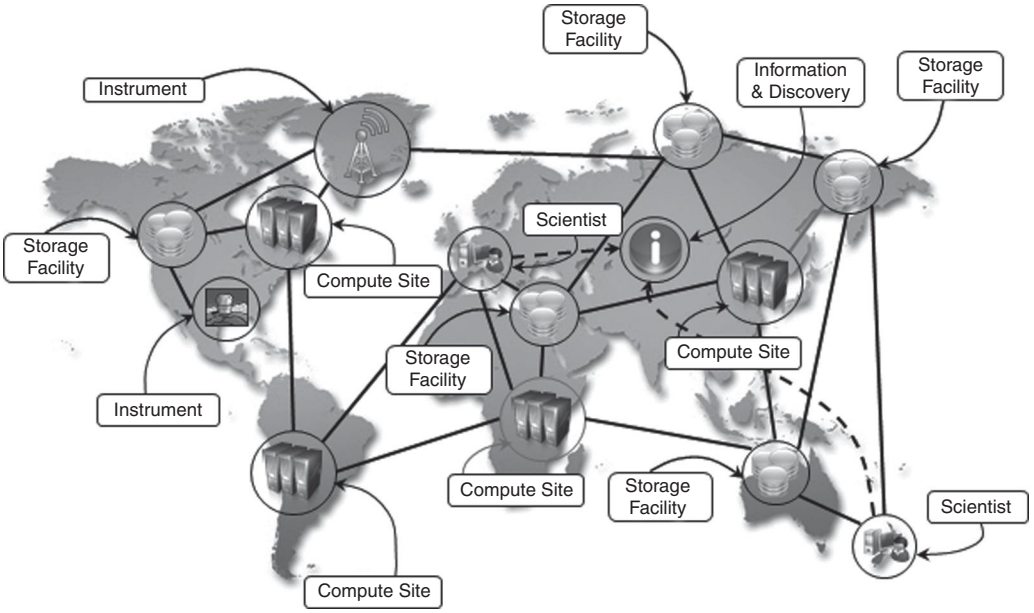


**Fig. 8.2.** Data Grid Reference Scenario.

**(c) Unified Namespace.** Data Grids impose a unified logical namespace where to locate data collections and resources. Every data element has a single logical name, which is eventually mapped to different physical file names for the purpose of replication and accessibility.

**(d) Access Restrictions.** Even though one of the purposes of Data Grids is facilitating the sharing of results and data for experiments, some users might wish to ensure confidentiality for their data and restrict access to them only to their collaborators. Authentication and authorization in Data Grids involve both coarse-grained and fine-grained access control over shared data collections.

With respect to the combination of several computing facilities through high speed networking, Data Grids constitute a more structured and integrated approach for data-intensive computing. As a result, several scientific research fields including high-energy physics, biology, and astronomy leverage Data Grids as briefly discussed below:

*The LHC Grid.* A project funded by the European Union to develop a world-wide Grid computing environment for use by high-energy physics researchers around the world collaborating on the *Large Hadron Collider (LHC) experiment. It supports* storage and analysis of large-scale datasets, from hundreds of terabytes to petabytes, generated by the LHC experiment (*http:///lhc.Web.cern.ch/lhc/)*.

*BioInformatics Research Network (BIRN).* BIRN is a national initiative to advance biomedical research through data sharing and online collaboration. Funded by the National Center for Research Resources (NCRR), a component of the US National Institutes of Health (NIH), BIRN provides data-sharing infrastructure, software tools and strategies, and advisory services (*http://www.birncommunity.org*).

*International Virtual Observatory Alliance (IVOA).* IVOA is an organization aimed at providing improved access to the ever-expanding astronomical data resources available on-line.

It does so by promoting standards for Virtual Observatories, which are a collection of interoperating data archives and software tools utilizing the Internet to form a scientific research environment where astronomical research programs can be conducted. This allows scientists to discover, access, analyze, and combine lab data from heterogeneous data collections (http://www.ivoa.net/).

A complete taxonomy of Data Grids can be found in Venugopal et al. [79].

# 3. Data Clouds and "Big Data"

Large datasets have mostly been the domain of scientific computing. This scenario has recently started to change as massive amount of data are being produced, mined, and crunched by companies providing Internet services such as searching, on-line advertisement, and social media. It is critical for such companies to efficiently analyze these huge datasets as they constitute a precious source of information about their customers. Logs analysis is an example of data-intensive operation that is commonly performed in this context: companies such as Google have a massive amount of data in the form of logs that are daily processed by using their distributed infrastructure. As a result, they settled upon analytic infrastructure that differs from the Grid-based infrastructure used by the scientific community.

Together with the diffusion of Cloud computing technologies supporting data-intensive computations, the term "Big Data" [82] has become popular. This term characterizes the nature of data-intensive computations nowadays and currently identifies datasets that grow so large that they become complex to work with using on-hand database management tools. Relational databases and desktop statistics/visualization packages become ineffective for that amount of information requiring instead "massively parallel software running on tens, hundreds, or even thousands of servers" [82].

*Big Data* problems are found in non-scientific application domains such as Web logs, RFID, sensor networks, social networks, Internet text and documents, Internet search indexing, call detail records, military surveillance, medical records, photography archives, video archives, and large scale e-Commerce. Other than the massive size, what characterizes all these examples is that new data is accumulated with time rather than replacing the old ones. In general, *Big Data* applies to datasets whose size is beyond the ability of commonly used software tools to capture, manage, and process the data within a tolerable elapsed time. Therefore, *Big Data* sizes are a constantly moving target currently ranging from a few dozen terabytes to many petabytes of data in a single dataset [82].

Cloud technologies support data-intensive computing in several ways:

- By providing a large amount of compute instances on demand that can be used to process and analyze large datasets in parallel.
- By providing a storage system optimized for keeping large blobs of data and other distributed data store architectures.
- By providing frameworks and programming APIs optimized for the processing and management of large amount of data. These APIs are mostly coupled with a specific storage infrastructure in order to optimize the overall performance of the system.

A Data Cloud is a combination of these components. An example is the *MapReduce* framework [55], which provides the best performance when leveraging the *Google File System* [54] on top of Google's large computing infrastructure. Another example is the Hadoop system [83]—the most mature, large, and open source Data Cloud. It consists of the *Hadoop Distributed File System (HDFS)* and Hadoop's implementation of *MapReduce*. A similar approach is proposed by *Sector* [84], which consists of the *Sector Distributed File System (SDFS)* and a compute service called *Sphere* [84] that allows users to execute arbitrary *User Defined Functions (UDFs)* over the data managed by SDFS. *Greenplum* uses a shared-nothing Massively Parallel Processing (MPP) architecture based upon commodity hardware. The architecture also integrates MapReduce-like functionality into its platform. A similar architecture has been deployed by *Aster*, which uses MPP-based data warehousing appliance supporting *MapReduce* and targeting 1 PB of data.

## 4. Databases and Data-Intensive Computing

Traditionally, distributed databases [85] have been considered the natural evolution of database management systems as the scale of the datasets becomes unmanageable with a single system. Distributed databases are a collection of data stored at different sites of a computer network. Each site might expose a degree of autonomy providing services for the execution of local applications, but also participates in the execution of a global application. A distributed database can be created by splitting and scattering the data of an existing database over different sites, or by federating together multiple existing databases. These systems are very robust and provide distributed transaction processing, distributed query optimization, and efficient management of resources. However, they are mostly concerned with datasets that can be expressed by using the relational model [86], and the need to enforce ACID properties on data limits their abilities to scale as Data Clouds and Grids do.

# 8.2 TECHNOLOGIES FOR DATA-INTENSIVE COMPUTING

Data-intensive computing concerns the development of applications that are mainly focused on processing large quantities of data. Therefore, storage systems and programming models constitute a natural classification of the technologies supporting data-intensive computing.

## 8.2.1 Storage Systems

Traditionally, database management systems constituted the de-facto storage support for several types of applications. Due to the explosion of unstructured data in the form of blogs, Web pages, software logs, and sensor readings, the relational model in its original formulation, does not seem to be the preferred solution for supporting data analytics at a large scale [88]. Research on databases and the data management industry are indeed at a turning point and new opportunities arise. Some factors contributing to this change are:

- *Growing of popularity of Big Data.* The management of large quantities of data is not anymore a rare case but it has become common in several fields: scientific computing, enterprise applications, media entertainment, natural language processing, and social network analysis. The large volume of data imposes new and more efficient techniques for their management.
- *Growing importance of data analytics in the business chain.* The management of data is not considered anymore a cost but a key element to the business profit. This situation arises in popular social networks such as Facebook, which concentrate their focus on the management of user profiles, interests, and connections among people. This massive amount of data, which is constantly mined, requires new technologies and strategies supporting data analytics.
- *Presence of data in several forms, not only structured.* As previously mentioned, what constitutes relevant information today exhibits a heterogeneous nature and appears in several forms and formats. Structured data is constantly growing as a result of the continuous use of traditional enterprise applications and system, but at the same time the advances in technology and the democratization of the Internet as a platform where everyone can pull information has created a massive amount of information that is unstructured and does not naturally fit into the relational model.
- *New approaches and technologies for computing.* Cloud computing promises access to a massive amount of computing capacity on demand. This allows engineers to design software systems that incrementally scale to arbitrary degrees of parallelism. It is not rare anymore to build software applications and services that are dynamically deployed on hundreds or thousands of nodes, which might belong to the system for few hours or days. Classical database infrastructures are not designed to provide support to such a volatile environment.

All these factors identify the need of new data management technologies. This does not only imply a new research agenda in database technologies and a more holistic approach to the management of information, but also leaves room to alternatives (or complements) to the relational model. In particular,

advances in distributed file systems for the management of raw data in the form of files, distributed object stores, and the spread of the NoSQL movement constitute the major directions towards support for data-intensive computing.

## 1. High-Performance Distributed File Systems and Storage Clouds

Distributed file systems constitute the primary support for the management of data. They provide an interface where to store information in the form of files and later access them for read and write operations. Among the several implementations of file systems, few of them specifically address the management of huge quantities of data on a large number of nodes. Mostly, these file systems constitute the data storage support for large computing clusters, supercomputers, massively parallel architectures, and lately storage/computing Clouds.

**(a) Lustre.** The Lustre file system is a massively parallel distributed file system that covers the needs of a small workgroup of clusters to a large scale computing cluster. The file system is used by several of the top 500 supercomputing systems, including the one rated as the most powerful supercomputer in the June 2012 list[46]. Lustre is designed to provide access to petabytes (PBs) of storage, to serve thousands of clients with an IO throughput of hundreds of gigabytes per second (GB/s). The system is composed by a metadata server containing the metadata information about the file system and a collection of object storage servers that are in charge of providing storage. Users access the file system via a POSIX compliant client, which can be either mounted as a module in the kernel or through a library. The file system implements a robust failover strategy and recovery mechanism, making server failures and recoveries transparent to clients.

**(b) IBM General Parallel File System (GPFS).** GPFS [88] is the high-performance distributed file system developed by IBM providing support for RS/6000 supercomputer and Linux computing clusters. GPFS is a multi-platform distributed file system built over several years of academic research and provides advanced recovery mechanisms. GPFS is built on the concept of shared disks, where a collection of disks is attached to the file system's nodes by means of some switching fabric. The file system makes this infrastructure transparent to users and stripes large files over the disk array also by replicating portion of the file in order to ensure high availability. By means of this infrastructure, the system is able to support petabytes of storage, which is accessed at a high throughput and without losing consistency of data. Compared to other implementations, GPFS distributes also the metadata of the entire file system and provides transparent access to it, thus eliminating a single point of failure.

**(c) Google File System (GFS).** GFS [54] is the storage infrastructure supporting the execution of distributed applications in the Google's computing Cloud. The system has been designed to be a fault tolerant, high available, distributed file system built on commodity hardware and standard Linux operating systems. Rather than a generic implementation of a distributed file system, GFS specifically addresses the needs of Google in terms of distributed storage for applications, and it has been designed with the following assumptions:

- The system is built on top of commodity hardware that often fails.
- The system stores a modest number of large files, multi-GB files are common and should be treated efficiently, and small files must be supported but there is no need to optimize for that.
- The workloads primarily consist of two kinds of reads: large streaming reads and small random reads.
- The workloads also have many large, sequential writes that append data to files.
- High-sustained bandwidth is more important than low latency.

[46] Top500 supercomputers list: ***http://www.top500.org*** (accessed in June 2012).

The architecture of the file system is organized into a single master, containing the metadata of the entire file system, and a collection of chunk servers, which provide storage space. From a logical point of view, the system is composed by a collection of software daemons, which implement either the master server or the chunk server. A file is a collection of chunks whose size can be configured at file system level. Chunks are replicated on multiple nodes in order to tolerate failures. Clients look up the master server and identify the specific chunk of a file they want to access. Once the chunk is identified, the interaction happens between the client and the chunk server. Applications interact through the file system with specific interface supporting the usual operations for file creation, deletion, read, and write. The interface also supports *snapshots* and *record append* operations that are frequently performed by applications. GFS has been conceived by considering that failures in a large distributed infrastructure are common rather than a rarity, therefore a specific attention has been put in implementing a highly available, lightweight, and fault-tolerant infrastructure. The potential single point of failure of the single master architecture has been addressed by giving the possibility of replicating the master node on any other node belonging to the infrastructure. Moreover, a stateless daemon and extensive logging capabilities facilitate the system recovery from failures.

**(d) Sector.** Sector [84] is the storage Cloud supporting the execution of data-intensive applications defined according to the Sphere framework [84]. It is a user space file system that can be deployed on commodity hardware across a wide area network. Compared to other file systems, Sector does not partition a file into blocks but replicates the entire files on multiple nodes allowing the users to customize the replication strategy for a better performance. The architecture of the system is composed by four nodes: a security server, one or more master nodes, slave nodes, and client machines. The security server maintains all the information about access control policies for user and files, while master servers coordinate and serve the IO requests of clients, which ultimately interact with slave nodes for accessing a file. The protocol used to exchange data with slave nodes is UDT [89], which is a lightweight connection-oriented protocol optimized for wide area networks.

**(e) Amazon Simple Storage Service (S3).** Amazon S3 is the on-line storage service provided by Amazon. Even though its internal details are not revealed, the system is claimed to support high availability, reliability, scalability, infinite storage, and low latency at commodity cost. The system offers a flat storage space organized into buckets, which are attached to an Amazon Web Services (AWS) account. Each bucket can store multiple objects, each of them identified by a unique key. Objects are identified by unique URLs and exposed through the HTTP protocol, thus allowing a very simple *get-put* semantics. Because of the use of the HTTP protocol, there is no need of any specific library for accessing the storage system, whose objects can also be retrieved through the Bit Torrent protocol[47]. Despite its simple semantics, a POSIX-like client library has been developed to mount S3 buckets as part of the local file system. Besides the minimal semantics, security is another limitation of S3. The visibility and the accessibility of objects are linked to AWS account, and the owner of a bucket can decide to make it visible to other accounts or to the public. It is also possible to define authenticated URLs, which provide public access to anyone for a limited (and configurable) period of time.

Besides these examples of storage systems, there exist other implementations of distributed file systems and storage Clouds whose architecture is similar to the models discussed here. Except for the S3 service, it is possible to sketch a general reference architecture in all the systems presented that identifies two major roles into which all the nodes can be classified. Metadata or master nodes contain the information about the location of files or file chunks, whereas slave nodes are used for providing direct access to the storage space. The architecture is completed by client libraries which provide a simple interface for accessing the file system that is to some extent, or completely, compliant to the POSIX specification. Variations of the reference architecture can include the ability to support multiple masters, to distribute the metadata over multiple nodes, or to easily interchange the role of nodes.

[47] Bit Torrent is a P2P file sharing protocol used to distribute large amounts of data. The key characteristic of the protocol is the ability to allow users to download a file in parallel from multiple hosts.

The most important aspect common to all these different implementations is the ability to provide fault tolerant and highly available storage systems.

## 2. Not Only SQL (NoSQL) Systems

The term "NoSQL" was originally coined in 1998 to identify a relational database, which did not expose a SQL interface to manipulate and query data, but relied on a set of UNIX shell scripts and commands to operate on text files containing the actual data. In a very strict sense, NoSQL cannot be considered a relational database since it is not a monolithic piece of software organizing the information according to the relational model, but, rather, is a collection of scripts that allow users to manage most of the simplest and more common database tasks by using text files as information store. Later in 2009, the term "NoSQL" was reintroduced with the intent to label all those database management systems that did not use a relational model, but provided simpler and faster alternatives for data manipulation. Nowadays, the term "NoSQL" is a big umbrella encompassing all the storage and database management systems that differ in some way from the relational model. The general philosophy is to overcome the restrictions imposed by the relational model and to provide more efficient systems. This often implies the use of tables without fixed schemas to accommodate a larger range of data types, or avoiding joins to increase the performance and scale horizontally.

Two main reasons have determined the growth of the NoSQL movement: in many cases, simple data models are enough to represent the information used by applications; and the quantity of information contained in unstructured formats has considerably grown in the last decade. These two factors made software engineers look to alternatives that were more suitable to specific application domain they were working on. As a result, several different initiatives explored the use of non-relational storage systems, which considerably differ from each other. A broad classification is reported by Wikipedia[48] which distinguishes NoSQL implementations into:

- *Document stores* (Apache Jackrabbit, Apache CouchDB, SimpleDB, and Terrastore).
- *Graphs* (AllegroGraph, Neo4j, FlockDB, and Cerebrum).
- *Key-value stores*. This is a macro classification that is further categorized into key-value store on disk, key-value caches in RAM, hierarchically key-value stores, eventually consistent key-value stores, and ordered key-value store.
- *Multi-value databases* (OpenQM, Rocket U2, and OpenInsight).
- *Object databases* (ObjectStore, JADE, and ZODB).
- *Tabular stores* (Google BigTable, Hadoop HBase, and Hypertable).
- *Tuple stores* (Apache River).

We now discuss some prominent implementations supporting data-intensive applications.

**(a) Apache CouchDB and MongoDB.** Apache CouchDB [91] and MongoDB [90] are two examples of document stores. Both of them provide a schema-less store where the primary objects are documents, organized into a collection of key-value fields. The value of each field can be of type—string, integer, float, date, or an array of values. The databases expose a RESTful interface and represents data in JSON format. Both of them allow querying and indexing data by using the MapReduce programming model, expose Javascript as a base language for data querying and manipulation rather than SQL, and support large files as documents. From an infrastructure point of view, the two systems support data replication and high-availability. CouchDB ensures ACID properties on data. MongoDB supports *sharding*, which is the ability to distribute the content of a collection among different nodes.

**(b) Amazon Dynamo.** Dynamo [92] is the distributed key-value store supporting the management of information of several of the business services offered by Amazon Inc. The main goal of

[48] ***http://en.wikipedia.com/wiki/NoSQL***

Dynamo is to provide an incrementally scalable and highly available storage system. This goal helps in achieving reliability at a massive scale where thousands of servers and network components build an infrastructure serving 10 million of requests per day. Dynamo provides a simplified interface based on a *get*/*put* semantics, where objects are stored and retrieved with a unique identifier (key). The main goal of achieving an extremely reliable infrastructure has imposed some constraints on the properties these systems have. For example, ACID properties on data have been sacrificed in favor of a more reliable and efficient infrastructure. This creates what is called an *eventually consistent* model, which means that, in the long term, all the users will see the same data.

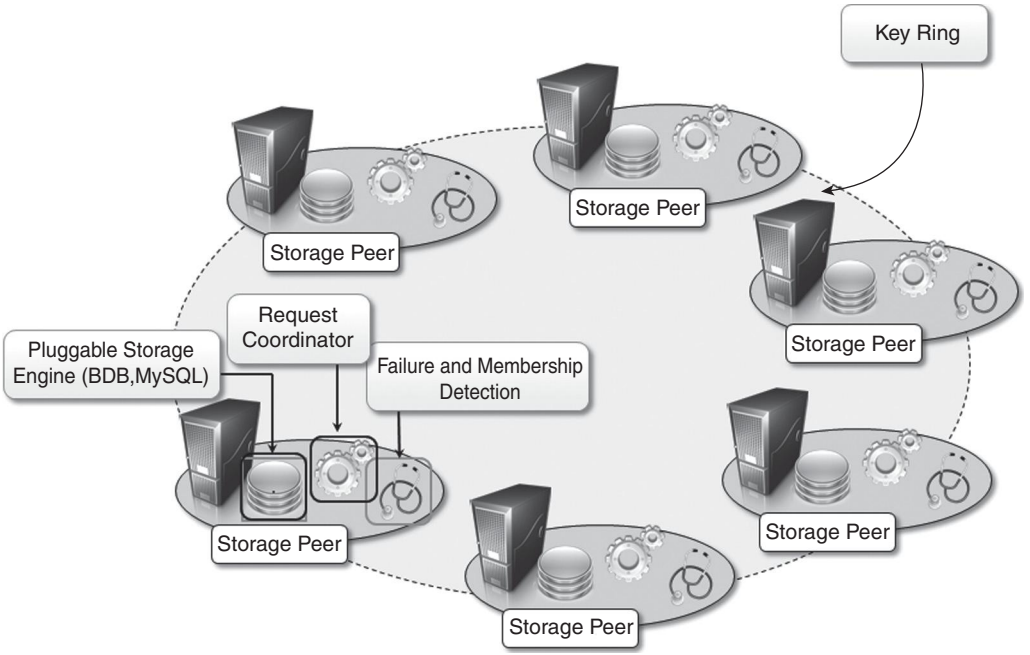


**Fig. 8.3.** Amazon Dynamo Architecture.

The architecture of the Dynamo system, shown in Fig.8.3, is composed of a collection of storage peers organized in a ring, sharing the key space for a given application. The key space is partitioned among the storage peers and the keys are replicated across the ring avoiding adjacent peers. Each peer is configured with access to a local storage facility where original objects and replicas are stored. Also each node provides facilities for distributing the updates among the ring, and to detect failures and unreachable nodes. With some relaxation of the consistency model applied to replicas and the use of object versioning, Dynamo implements the capability of being an *always writable store*, where consistency of data is resolved in background. The downside of such approach is the simplicity of the storage model, which requires applications to build their own data model on top of the simple building blocks provided by the store. For example, there are no referential integrity constraints, relations are not embedded in the storage model, and therefore join operations are not supported. These restrictions are not prohibitive in the case of Amazon services for which the single key-value model is acceptable.

**(c) Google Bigtable.** Bigtable [93] is the distributed storage system designed to scale up to petabytes of data across thousands of servers. Bigtable provides storage support for several Google applications exposing different types of workload: from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The key design goals of Bigtable are wide applicability,

scalability, high performance, and high availability. To achieve these goals, Bigtable organizes the data storage in tables whose rows are distributed over the distributed file system supporting the middleware, which is the Google File System. From a logical point of view, a table is a multidimensional sorted map indexed by a key represented by a string of arbitrary length. A table is organized in rows and columns, columns can be grouped in column family, which allow for specific optimization for better access control, storage and indexing of data. Client applications are provided with very simple data access model that allow them to address data at column level. Moreover, each column value is stored in multiple versions that can be automatically time-stamped by Bigtable or by the client applications.

Besides the basic data access, Bigtable APIs also allow more complex operations such as single row transactions and advanced data manipulation by means of the Sazwall[49] [95] scripting language or the MapReduce APIs.

Figure 8.4 gives an overview of the infrastructure enabling Bigtable. The service is the result of a collection of processes that co-exist with other processes in a cluster-based environment. Bigtable identifies two kinds of processes: master processes and tablet server processes. A tablet server is responsible for serving the requests for a given tablet that is a contiguous partition of rows of a table. Each server can manage multiple tablets (from ten to a thousand commonly). The master server is responsible of keeping track of the status of the tablet servers, and of the allocation of tablets to tablets servers. The server constantly monitors the tablet servers to check whether they are alive, and in case they are not reachable, the allocated tablets are reassigned and eventually partitioned to other servers.

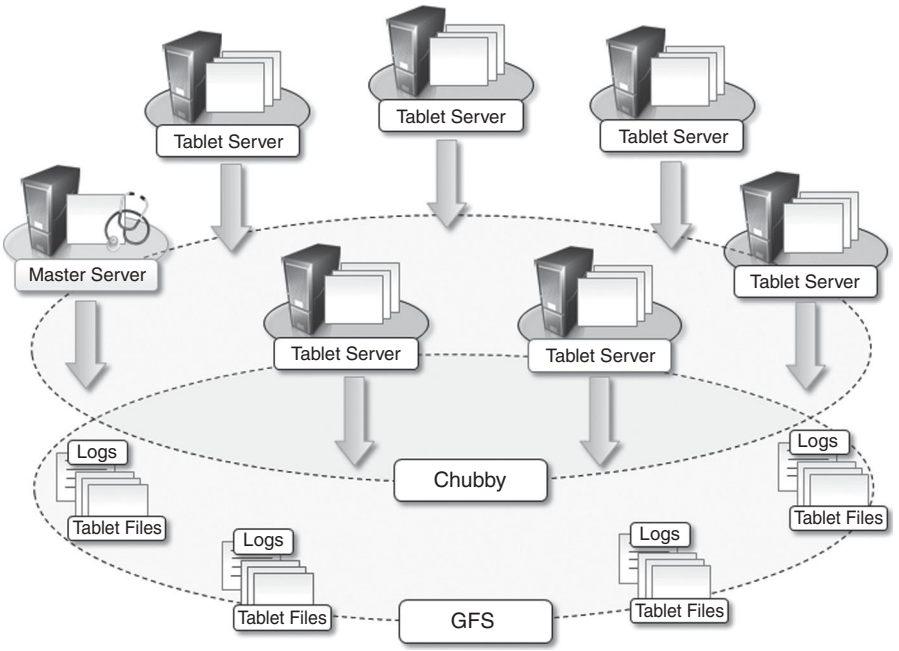


**Fig. 8.4.** Bigtable Architecture.

Chubby [96]—a distributed, highly available and persistent lock service—supports the activity of the master and tablet servers. System monitoring and data access is filtered through Chubby that is also responsible

[49] Sazwall is an interpreted procedural programming language developed at Google for the manipulation of large quantities of tabular data. It includes specific capabilities for supporting statistical aggregation of values read or computed from the input and other features that simplify the parallel processing of petabytes of data.

for managing replicas and providing consistency among them. At the very bottom layer, the data is stored into the Google File System in the form of files and all the update operations are logged into the file for the easy recovery of data, in case of failures or when tablets need to be reassigned to other servers. Bigtable uses a specific file format for storing the data of a tablet, which can be compressed for optimizing the access and the storage of data.

Bigtable is the result of a study of the requirements of several distributed applications in Google. It serves as a storage backend for 60 applications (such as Google Personalized Search, Google Anlytics, Google Finance, and Google Earth), and manages petabytes of data.

**(d) Apache Cassandra.** Cassandra [94] is a distributed object store for managing large amounts of structured data spread across many commodity servers. The system is designed to avoid a single point of failure and offers a highly reliable service. Cassandra was initially developed by Facebook and now it is part of the Apache incubator initiative. Currently, it provides storage support for several very large Web applications such as *Facebook* itself, *Digg*, and *Twitter*. Cassandra is defined as a second generation, distributed database that builds on the concept of *Amazon Dynamo* which follows a fully distributed design and *Google Bigtable,* from which inherits the "Column Family" concept. The data model exposed by Cassandra is based on the concept of table that is implemented as a distributed, multi-dimensional map indexed by a key. The value corresponding to a key is a highly structured object and constitutes the row of a table. Cassandra organizes the row of a table into columns, and sets of columns can be grouped into column families. The APIs provided by the system to access and manipulate the data are very simple: insertion, retrieval, and deletion. The insertion is performed at row level, while retrieval and deletion can operate at column level.

In term of the infrastructure, Cassandra is very similar to Dynamo. It has been designed for incremental scaling and it organizes the collection of nodes sharing a key space into a ring. Each node manages multiple and discontinuous portions of the key space and replicate their data up to N other nodes. Replication uses different strategies and it can be *rack aware*, *data center aware*, or *rack unaware*, meaning that the policies can take into account whether the replication needs to be made within the same cluster, data center, or not to consider the geo-location of nodes. As in Dynamo, node membership information is based on gossip protocols[50]. Cassandra also makes use of this information diffusion mode for other tasks such as disseminating the system control state. The local file system of each node is used for data persistence and Cassandra also makes extensive use of commit logs, which makes the system able to recover from transient failures. Each write operation is applied in memory only after it has been logged on disk so that it can be easily reproduced in case of failures. When the data in memory trespasses a specified size, it is dumped to disk. Read operations are performed in-memory first and then on disk. In order to speed up the process, each file includes a summary of the keys it contains, so that it is possible to avoid unnecessary file scanning to search for a key.

As noted earlier, Cassandra builds on the concepts designed in Dynamo and Bigtable, and puts them together in order to achieve a completely distributed and highly reliable storage system. The largest Cassandra deployment to knowledge manages 100 TB of data distributed over a cluster of 150 machines.

**(e) Hadoop HBase.** HBase is the distributed database supporting the storage needs of the Hadoop distributed programming platform. HBase is designed by taking inspiration from Google Bigtable, and its main goal is to offer real time read/write operations for tables with billions of rows and millions of columns by leveraging clusters of commodity hardware. The internal architecture and logic model of HBase is very similar to Google Bigtable, and the entire system is backed by the Hadoop Distributed File System (HDFS), which mimics the structure and the services of GFS.

In this section, we discussed the storage solutions supporting the management of data-intensive applications, and especially of what is referred as "Big Data". Traditionally, database systems, most

[50] A gossip protocol is a style of communication protocol inspired by the form of gossip seen in social networks. Gossip protocols are used in distributed systems as an efficient alternative to distribute and propagate information, if compared to flooding or other kind of algorithms.

likely based on the relational model, have been the primary solution for handling large quantities of data. As it has been discussed, when it comes to extremely huge quantities of unstructured data, relational databases become not practical and provide poor performance. Alternative and more effective solutions have significantly reviewed the fundamental concepts laying at the basis of distributed file systems and storage systems. The next level is constituted by providing programming platforms that, by leveraging the discussed storage systems, can capitalize the efforts of developers for handling massive amount of data. Among them, MapReduce and all its variations play a fundamental role.

## 8.2.2 Programming Platforms

Platforms for programming data-intensive applications provide abstractions helping to express the computation over a large quantity of information and runtime systems able to manage efficiently huge volumes of data. Traditionally, database-management systems based on the relational model have been used to express the structure and the connections between the entities of a data model. This approach has proven to be not successful in the case of "Big Data" where information is mostly found unstructured or semi-structured, and where data is most likely to be organized in files of large size or a huge number of medium size files, rather than rows in a database. Distributed workflows have been often used to analyze and process large amounts of data [66][67]. This approach introduced a plethora of frameworks for workflow management systems, as discussed in Section 7.2.4, which eventually incorporated capabilities to leverage the elastic features offered by Cloud computing [70]. These systems are fundamentally based on the abstraction of *task*, which puts a lot of burden on the developer who needs to deal with data management and, often, data-transfer issues. Programming platforms for data intensive computing provide higher level abstractions, which focus on the processing of data and move into the runtime system the management of transfers, thus making the data always available where needed. This is the approach followed by the MapReduce [55] programming platform, which expresses the computation in the form of two simple functions—map and reduce—and hides away the complexities of managing large and numerous data files into the distributed file system supporting the platform. In this section, we discuss the characteristics of MapReduce and present some variations of it, which extend its capabilities for wider purposes.

### *1. The MapReduce Programming Model*

MapReduce [55] is a programming platform introduced by Google for processing large quantities of data. It expresses the computation logic of an application into two simple functions: *map* and *reduce*. Data transfer and management is completely handled by the distributed storage infrastructure (i.e., the Google File System), which is in charge of providing access to data, replicating files, and eventually moving them where needed. Therefore, developers do not have to handle anymore these issues, and are provided with an interface that presents data at a higher level: as a collection of key-value pairs. The computation of MapReduce applications is then organized in a workflow of map and reduce operations that is entirely controlled by the runtime system, and developers have only to specify how the map and reduce functions operate on the key value pairs.

More precisely, the model is expressed in the form of the two functions, which are defined as follows:

$$map\ (k1,v1) \rightarrow list(k2,v2)$$

$$reduce(k2,list(v2)) \rightarrow list(v2)$$

The *map* function reads a key-value pair and produces a list of key-value pairs of different types. The *reduce* function reads a pair composed by a key and a list of values, and produces a list of values of the same type. The types (*k1,v1,k2,kv2*) used in the expression of the two functions provide hints on how these two functions are connected and are executed to carry out the computation of a MapReduce job: the output of map tasks is aggregated together by grouping the values according to their corresponding keys and constitute the input of reduce tasks that, for each of the keys found, reduces the list of attached values to a single value. Therefore, the input of a MapReduce computation is expressed as a collection of key-value pairs <*k1,v1*> and the final output is represented by a list values: *list(v2).*

Figure 8.5 depicts a reference workflow characterizing MapReduce computations. As shown, the user submits a collection of files that are expressed in the form of a list of *<k1,v1>* pairs and specifies the map and reduce functions. These files are entered into the distributed file system supporting MapReduce and, if necessary, partitioned in order to be the input of map tasks. Map tasks generate intermediate files that store collections of *<k2, list(v2)>* pairs and these files are saved into the distributed file system. The MapReduce runtime might eventually aggregate the values corresponding to the same keys. These files constitute the input of reduce tasks, which finally produce output files in the form of *list(v2)*. The operation performed by reduce tasks is generally expressed as an aggregation of all the values that are mapped by a specific key. The number of map and reduce tasks to create, the way in which files are partitioned with respect to these tasks, and how many map tasks are connected to a single reduce task are the responsibilities of the MapReduce runtime. Also, the way in which files are stored and moved are the responsibilities of the distributed file system supporting MapReduce.

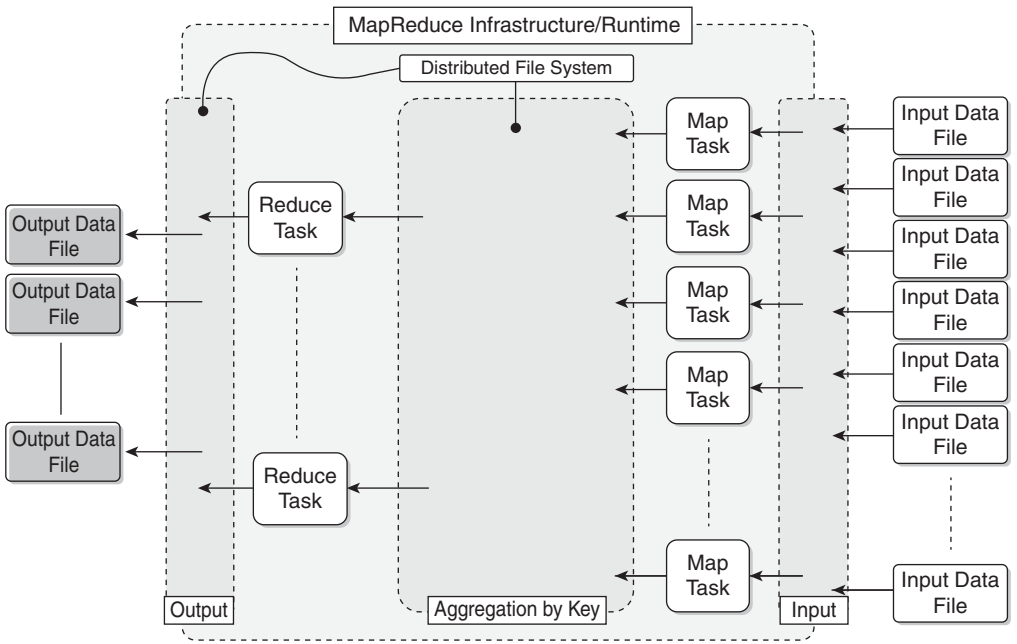


**Fig. 8.5.** MapReduce Computation Workflow.

The computation model expressed by MapReduce is very straightforward and allows a greater productivity for those who have to code the algorithms for processing huge quantities of data. This model has proven to be successful in the case of Google, where the majority of the information that needs to be processed is stored in textual form and represented by Web pages or log files. Some of the examples that show the flexibility of MapReduce are the following [55]:

**(a) Distributed Grep.** The grep operation, which performs the recognition of patterns within text streams, is performed across a wide set of files. MapReduce is leveraged to provide a parallel and faster execution of this operation. In this case, the input file is a plain text file and the map function emits a line into the output each time it recognizes the given pattern. The reduce task aggregates all the lines emitted by the map tasks into a single file.

**(b) Count of URL-Access Frequency.** MapReduce is used to distribute the execution of Web-server log parsing. In this case, the map function takes as input the log of a Web server and

emits into the output file a key-value pair *<URL,1>* for each page access recorded in the log. The reduce function aggregates all these lines by the corresponding URL thus summing the single accesses and outputs a *<URL, total-count>* pair.

**(c) Reverse Web-Link Graph.** The Reverse Web-link graph keeps track of all the possible Web pages that might lead to a given link. In this case, input files are simple HTML pages that are scanned by map tasks emitting *<target, source>* pairs for each of the links found given in the Web page *source*. The reduce task will collate all the pairs that have the same target into a *<target, list(source)>* pair. The final result is given in one or more files containing these mappings.

**(d) Term-Vector per Host.** A term vector recaps the most important words occurring in a set of documents in the form of *list(<word, frequency>)*, where the number of occurrences of a word is taken as a measure of its importance. MapReduce is used to provide a mapping between the origin of a set of document, obtained as the host component of the URL of a document, and the corresponding term vector. In this case, the map task creates a pair *<host, term-vector>* for each text document retrieved, and the reduce task aggregates the term vectors corresponding to documents retrieved from the same host.

**(e) Inverted Index.** The inverted index contains information about the presence of words in documents. This information is useful to allow quick full text searches if compared to direct document scans. In this case, the map task takes as input a document and for each document, it emits a collection of *<word, document-id>*. The reduce function aggregates the occurrences of the same word, producing a pair *<word, list(document-id)>*.

**(f) Distributed Sort.** In this case, MapReduce is used to parallelize the execution of a sort operation over a large number of records. This application mostly relies on the properties of the MapReduce runtime, which sorts and creates partitions of the intermediate files, rather than in the operations performed in the map and reduce tasks. Indeed, these are very simple: the map task extracts the key from a record and emits a <key, record> pair for each record; the reduce task will simply copy through all the pairs. The actual sorting process is performed by the MapReduce runtime which will emit and partition the key value pair by ordering them according to the key.

The examples reported are mostly concerned with text-based processing. MapReduce can also be used, with some adaptation, to solve a wider range of problems. An interesting example is its application in the field of machine learning [97], where statistical algorithms such as Support Vector Machines (SVM), Linear Regression (LR), Naïve Bayes (NB), and Neural Network (NN), are expressed in the form of map and reduce functions. Other interesting applications can be found in the field of compute intensive applications such as the computation of Pi with high degree of precision. It has been reported that the Yahoo Hadoop cluster has been used to compute the $10^{15}$ + 1 bit of Pi[51]. Hadoop is an open-source implementation of the MapReduce platform.

In general, any computation that can be expressed in the form of two major stages can be represented in the terms of MapReduce computation. These stages are the following:

- **Analysis** This phase operates directly to the data input file and corresponds to the operation performed by the map task. Moreover, the computation at this stage is expected to be embarrassingly parallel, since map tasks are executed without any sequencing or ordering.
- **Aggregation** This phase operates on the intermediate results, and it is characterized by operations which are aimed at aggregating, summing, and/or elaborating the data obtained at the previous stage to present it into its final form. This is the task performed by the reduce function.

Adaptations to this model are mostly concerned with: identifying what are the appropriate keys, how to create reasonable keys when the original problem does not have such a model, and how to partition the

[51] The full details of this computation can be found in the Yahoo Developer Network blog in the following blog post: ***http://developer.yahoo.com/blogs/hadoop/posts/2009/05/hadoop_computes_the_10151st_bi/.***

computation between *map* and *reduce* functions. Moreover, more complex algorithms can be decomposed into multiple MapReduce programs where the output of one program constitutes the input of the following program.

The abstraction proposed by MapReduce provides developers with a very minimal interface that is strongly focused on the algorithm to implement, rather than the infrastructure on which it is executed. This is a very effective approach but at the same time, demands a lot of common tasks, which are of concern in the management of a distributed application to the MapReduce runtime, allowing the user only to specify configuration parameters to control the behavior of applications. These tasks are the management of data transfer, and the scheduling of map and reduce tasks over a distributed infrastructure. Fig.8.6 gives a more complete overview of a MapReduce infrastructure, according to the implementation proposed by Google [55].

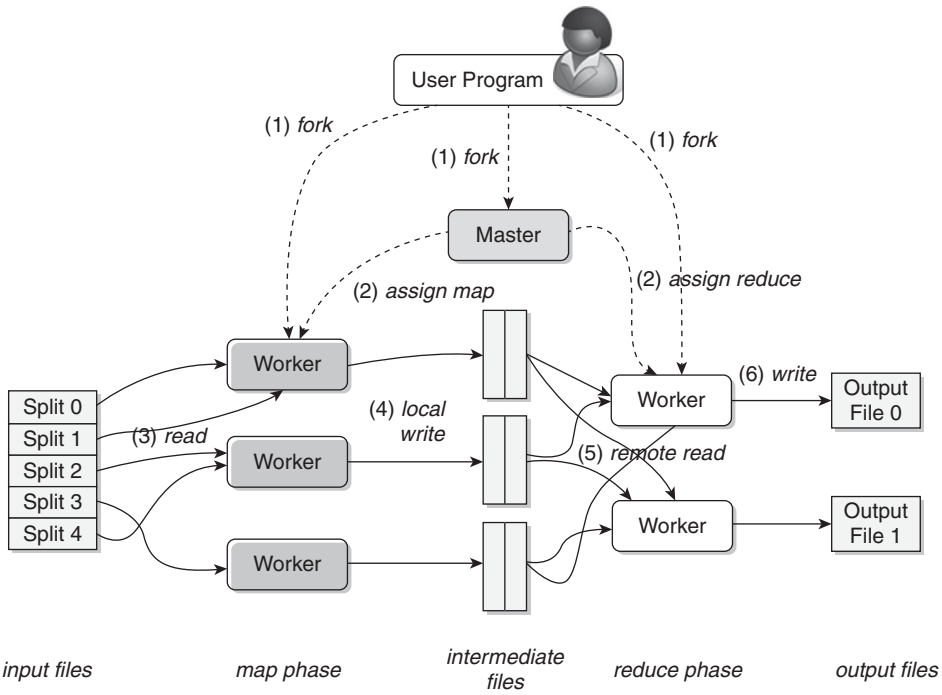


**Fig. 8.6.** Google MapReduce Infrastructure Overview.

As depicted, the user submits the execution of MapReduce jobs by using the client libraries that are in charge of submitting the input data files, registering the map and reduce functions, and returning the control to the user once the job is completed. A generic distributed infrastructure (i.e., a cluster) equipped with job scheduling capabilities and distributed storage can be used to run MapReduce applications. Two different kinds of processes are run on the distributed infrastructure: a master process and a worker process.

The master process is in charge of controlling the execution of map and reduce tasks, partitioning, and reorganizing the intermediate output produced by the map task in order to feed the reduce tasks. The worker processes are used to host the execution of map and reduce tasks, and provide basic I/O facilities that are used to interface the map and reduce tasks with input and output files. In a MapReduce computation, input files are initially divided into splits (generally 16 to 64 MB) and stored in the distributed file system. The master process generates the map tasks and assigns input splits to each of them by balancing the load.

Worker processes have input and output buffers that are used to optimize the performance of map and reduce tasks. In particular, output buffer for map tasks are periodically dumped to disk in order to create intermediate files. Intermediate files are partitioned by using a user-defined function to split evenly the output of map tasks. The locations of these pairs are then notified to the master process, which forwards this information to the reduce task which are able to collect the required input via a remote procedure call in order to read from the map tasks' local storage. The key range is then sorted and all the same keys are grouped together. Finally, the reduce task is executed to produce the final output, which is stored into the global file system. This process is completely automatic, and users may control it through configuration parameters that allow specifying, besides the map and reduce function, the number of map tasks, the number of partitions into which separate the final output, and the partition function for the intermediate key range.

Besides orchestrating the execution of map and reduce tasks as described before, the MapReduce runtime ensures a reliable execution of applications by providing a fault-tolerant infrastructure. Failures of both master and worker processes are handled as well as machine failures that make intermediate outputs not accessible. Worker failures are handled by rescheduling map tasks somewhere else. This is also the technique that is used to address machine failures since the valid intermediate output of map tasks has become inaccessible. Master process failure is instead addressed by using checkpointing, which allow restarting the MapReduce job with a minimum loss of data and computation.

## 2. Variations and Extensions of MapReduce

MapReduce constitutes a simplified model for processing large quantities of data, and imposes constraints on how distributed algorithms should be organized in order to run over a MapReduce infrastructure. Although the model can be applied to several different problem scenarios, it still exhibit limitations mostly given by the fact that the abstractions provided to process data are very simple, and complex problems might require considerable effort to be represented in terms of *map* and *reduce* functions only. Therefore, a series of extensions and variations to the original MapReduce model have been proposed. They aim at extending MapReduce application space and providing an easier interface to developers for designing distributed algorithms. In this section, we briefly present a collection of MapReduce-like framework and discuss how they differ from the original MapReduce model.

**(a) Hadoop.** Apache Hadoop [83] is a collection of software projects for reliable and scalable distributed computing. Taken together, the entire collection is an open-source implementation of the MapReduce framework supported by a GFS-like distributed file system. The initiative consists of mostly two projects: Hadoop Distributed File System (HDFS) and Hadoop MapReduce. The former is an implementation of the Google File System [54], while the latter provides the same features and abstractions of Google MapReduce. Initially developed and supported by Yahoo, Hadoop constitutes now the most mature and large data Cloud application, and has a very robust community of developers and users supporting it. Yahoo runs now the world's largest Hadoop cluster, composed by 40000 machines and more than 300 thousands cores, made available to academic institutions all over the world. Besides the core projects of Hadoop, there is a collection of other projects, somehow related to it, providing services for distributed computing.

**(b) Pig.** Pig[52] is a platform allowing the analysis of large data sets. Developed as an Apache project, it consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs. Pig infrastructure's layer consists of a compiler for a high-level language that produces a sequence of MapReduce jobs that can be run on top of distributed infrastructures such as Hadoop. Developers can express their data analysis programs in a textual language called *Pig Latin*, which exposes a SQL-like interface and it is characterized by major expressiveness, reduced programming effort, and a familiar interface with respect to MapReduce.

[52] ***http://pig.apache.org/***

**(c) Hive.** Hive[53] is another Apache initiative that provides a datawarehouse infrastructure on top of Hadoop MapReduce. It provides tools for easy data summarization, ad-hoc queries, and analysis of large datasets stored in Hadoop MapReduce files. Whereas the framework provides the same capabilities of a classical data warehouse, it does not exhibit the same performances, especially in terms of query latency and, for this reason, does not constitute a valid solution for online transaction processing. The major advantage of Hive resides in the ability to scale out since it is based on the Hadoop framework, and in the ability of providing a data warehouse infrastructure in environments where there is already a Hadoop system running.

**(d) Map-Reduce-Merge.** Map-Reduce-Merge [98] is an extension to the MapReduce model introducing a third phase to the standard MapReduce pipeline—the Merge phase—that allows effciently merging data already partitioned and sorted (or hashed) by map and reduce modules. The Map-Reduce-Merge framework simplifies the management of heterogeneous related datasets and provides an abstraction able to express the common relational algebra operators as well as several join algorithms.

**(e) Twister.** Twister [99] is an extension to the MapReduce model that allows the creation of iterative executions of MapReduce jobs. With respect to the normal MapReduce pipeline, the model proposed by twister proposes the following extensions:

```
1. Configure Map
2. Configure Reduce
3. While Condition Holds True Do
    a. Run MapReduce
    b. Apply Combine Operation to Result
    c. Update Condition
4. Close
```

Besides the iterative MapReduce computation, Twister provides additional features such as the ability for *map* and *reduce* tasks to refer to static and in memory data, the introduction of an additional phase called combine run at the end of the MapReduce job that aggregates the output together, and other tools for management of data.

## 3. Alternatives to MapReduce

Besides MapReduce, there are other abstractions that provide support to process large datasets and execute data-intensive workloads. To a different extent, they exhibit some similarities with the approach presented by MapReduce.

**(a) Sphere.** Sphere [84] is the distributed processing engine that leverages the Sector Distributed File System (SDFS). Rather than being a variation of MapReduce, Sphere implements the stream processing model (*Single Program Multiple Data*), and allows the developer to express the computation in terms of *User Defined Functions (UDF)* which are run against the distributed infrastructure. A specific combination of UDFs allows Sphere to express MapReduce computations. Sphere strongly leverages the Sector distributed file systems and it is built on top of the Sector's API for data access. User-defined functions are expressed in terms of programs that read and write streams. A stream is a data structure that provides access to a collection of data segments mapping one or more files in the SDFS. The collective execution of UDFs is achieved through the distributed execution of *Sphere Process Engines (SPEs)* which are assigned with a given stream segment. The execution model is a master-slave model

[53] ***http://hive.apache.org/***

that is client controlled: a Sphere client sends a request for processing to the master node that returns the list of available slaves and the client will choose the slaves where to execute Sphere processes and orchestrate the entire distributed execution.

**(b) All-Pairs.** All-Pairs [100] is an abstraction and a run-time environment for the optimized execution of data-intensive workloads. It provides a simple abstraction—in terms of the All-pairs function—that is common in many scientific computing domains:

*All-pairs(A:set, B:set, F:function) → M:matrix*

Examples of problems that can be represented into this model can be found in the field of biometrics where similarity matrices are composed as a result of the comparison of several images containing subject pictures. Another example is constituted by several applications and algorithms in data mining. The model expressed by the All-Pairs function can be easily solved by the following algorithm:

```
1.  For each $i in A
2.    For each $j in B
3.      Submit job F $i $j
```

This implementation is quite naïve and produces a poor performance in general. Moreover, other problems such as data distribution, dispatch latency, number of available compute nodes, and probability of failure are not handled specifically. The All-pairs model tries to address these issues by introducing a specification for the nature of the problem and an engine that, according to this specification, optimizes the distribution of tasks over a conventional cluster or grid infrastructure. The execution of a distributed application is controlled by the engine and develops in four stages: i) model the system; ii) distribute the data; iii) dispatch batch jobs; and iv) clean up the system. The interesting aspect of this model is mostly concentrated on the first two phases where the performance model of the system is built, and the data is opportunistically distributed in order to create the optimal number of tasks to assign to each node and optimize the utilization of the infrastructure.

**(c) DryadLINQ.** Dryad [101] is a Microsoft Research project investigating programming model for writing parallel and distributed programs to scale from a small cluster to a large data-center. The aim of Dryad is to provide an infrastructure for automatically parallelizing the execution of application without requiring the developer to know about distributed and parallel programming.

In Dryad, developers can express distributed applications as a set of sequential programs that are connected together by means of channels. More precisely, a Dryad computation can be expressed in terms of a directed acyclic graph where nodes are the sequential programs and vertices represent the channels connecting such programs. Because of this structure, Dryad is considered a superset of the MapReduce model since its general application model allows expressing graphs representing MapReduce computation as well. An interesting feature exposed by Dryad is the capability of supporting dynamic modification of the graph (to some extent) and of partitioning, if possible, the execution of the graph into stages. This infrastructure is used to serve different applications and tools for parallel programming. Among them, DryadLINQ [102] is a programming environment that produces Dryad computations from the Language Integrated Query (LINQ) extensions to C# [103]. The resulting framework provides a solution completely integrated into the .NET framework and able to express several distributed computing models, including MapReduce.

## 8.3 ANEKA MapReduce PROGRAMMING

Aneka provides an implementation of the MapReduce abstractions by following the reference model introduced by Google and implemented by Hadoop. MapReduce is supported as one of the available programming models that can be used to develop distributed applications.

### 8.3.1 Introducing the MapReduce Programming Model

The *MapReduce Programming Model* defines the abstractions and the runtime support for developing MapReduce applications on top of Aneka. Figure 8.7 provides an overview of the infrastructure supporting MapReduce in Aneka. A MapReduce job in Google MapReduce or Hadoop corresponds to the execution of a MapReduce application in Aneka. The application instance is specialized with components that identify the map and reduce functions to use. These functions are expressed in the terms of a Mapper and Reducer class that are extended from the AnekaMapReduce APIs. The runtime support is constituted by three main elements:

- *MapReduce Scheduling Service*, which plays the role of the master process in the Google and Hadoop implementation.
- *MapReduce Execution Service*, which plays the role of the worker process in the Google and Hadoop implementation.
- A specialized distributed file system that is used to move data files.

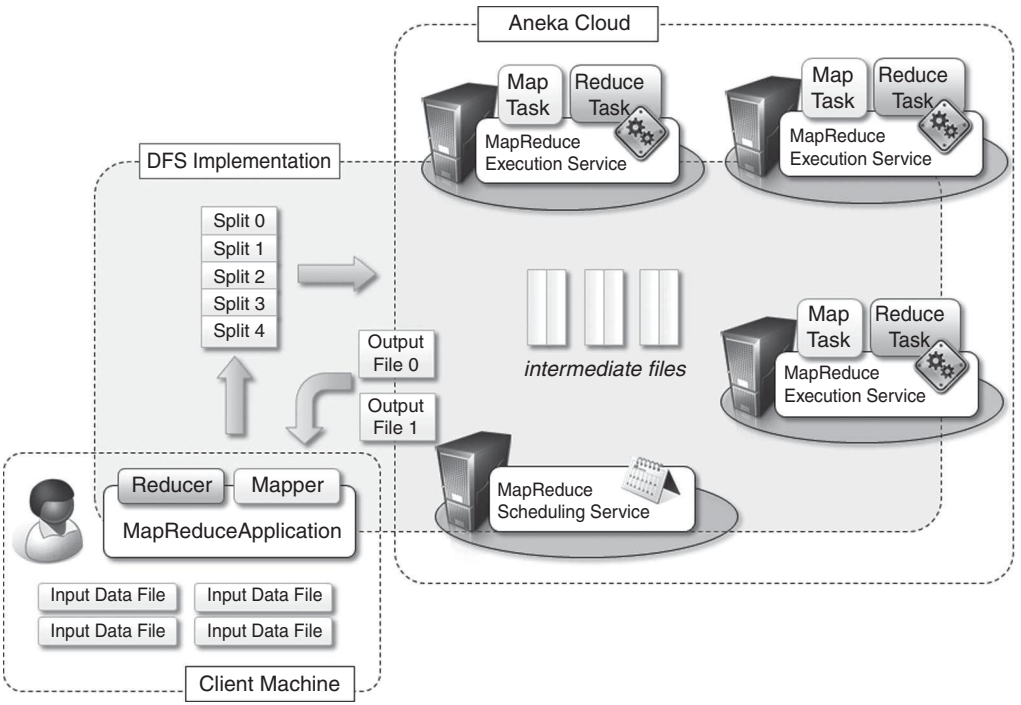


**Fig. 8.7.** Aneka MapReduce Infrastructure.

Client components, namely the *MapReduceApplication*, are used to submit the execution of a MapReduce job, upload data files, and monitor it. The management of data files is transparent: local data files are automatically uploaded to Aneka and output files are automatically downloaded to the client machine, if requested.

In the following, we introduce these major components and describe how they collaborate in order to execute MapReduce jobs.

# 1. Programming Abstractions

Aneka executes any piece of user code within the context of a distributed application. This approach is maintained even in the MapReduce programming model, where there is a natural mapping between the concept of MapReduce job—used in *Google MapReduce* and *Hadoop*—and the Aneka application concept. Different from other programming models, the task creation is not the responsibility of the user but of the infrastructure once the user has defined the *map* and *reduce* functions. Therefore, the Aneka MapReduce APIs provide developers with base classes for developing *Mapper* and *Reducer* types, and use a specialized type of application class—*MapReduceApplication*—that better supports the needs of this programming model.

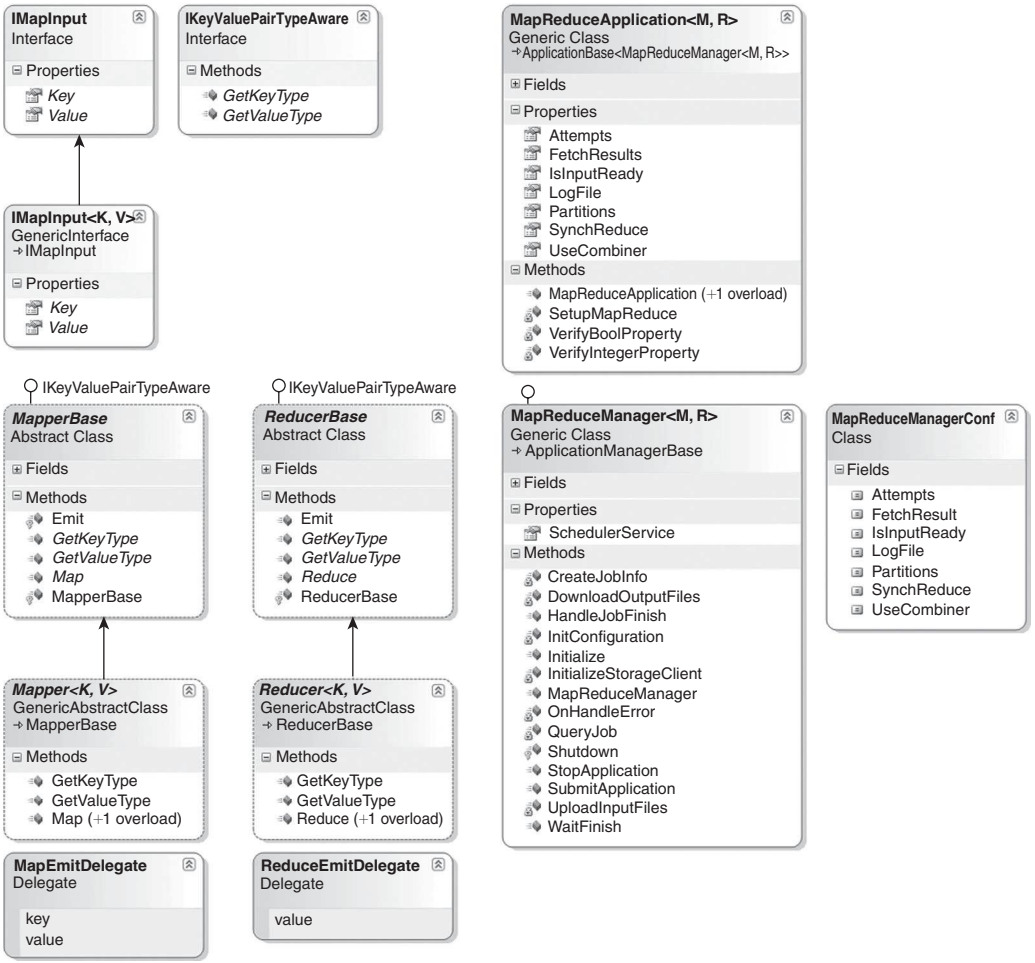


**Fig. 8.8.** MapReduce Abstractions Object Model.

Figure 8.8 provides an overview of the client components defining the MapReduce programming model. Three classes are of interest for application development: *Mapper<K,V>*, *Reducer<K,V>*, and *MapReduceApplication<M,R>*. The other classes are internally used to implement all the functionalities required by the model and expose simple interfaces that require minimum amount of coding for implementing the map and reduce functions, and controlling the job submission. *Mapper<K,V>* and

*Reducer<K,V>* constitute the starting point of the application design and implementation. Template specialization is used to keep track of keys and values types on which these two functions operate. Generics provide a more natural approach in terms of object manipulation from within the map and reduce methods, and simplify the programming by removing the necessity of casting and other type check operations. The submission and execution of a MapReduce job is performed through the class *MapReduceApplication<M,R>*, which provides the interface to the Aneka Cloud to support MapReduce programming model. This class exposes two generic types: *M* and *R*. These two placeholders identify the specific types of *Mapper<K,V>* and *Reducer<K,V>* that will be used by the application.

**Listing 1.** Map Function APIs.

```
using Aneka.MapReduce.Internal;
namespaceAneka.MapReduce
{
    /// <summary>
    /// Interface IMapInput<K,V>. Extends IMapInput and provides a strongly-
    /// typed version of the extended interface.
    /// </summary>
    public interface IMapInput<K,V>: IMapInput
    {
        /// <summary>
        /// Property <i>Key</i> returns the key of key/value pair.
        /// </summary>
        K Key { get; }
        /// <summary>
        /// Property <i>Value</i> returns the value of key/value pair.
        /// </summary>
        V Value { get; }
    }
    /// <summary>
    /// Delegate MapEmitDelegate. Defines the signature of a method
    /// that is used to doEmit intermediate results generated by the mapper.
    /// </summary>
    /// <param name=”key”>The <i>key</i> of the <i>key-value</i> pair.</param>
    /// <param name=”value”>The <i>value</i> of the <i>key-value</i> pair.</param>
    public delegate void MapEmitDelegate(object key, object value);
    /// <summary>
    /// Class Mapper. Extends MapperBase and provides a reference implementation that
    /// can be further extended in order to define the specific mapper for a given
    /// application. The definition of a specific mapper class only implies the
    /// implementation of the Mapper<K,V>.Map(IMapInput<K,V>) method.
    /// </summary>
    public abstract class Mapper<K,V> : MapperBase
    {
        /// <summary>
        /// Emits the intermediate result source by using doEmit.
        /// </summary>
```

```
        /// <param name = "source">An instance implementing IMapInput containing the
        /// <i>key-value</i> pair representing the intermediate result.</param>
        /// <param name = "doEmit">A MapEmitDelegate instance that is used to write
            to the
        /// output stream the information about the output of the Map operation.</
            param>
        public void Map(IMapInput input, MapEmitDelegate emit) { … }
        /// <summary>
        /// Gets the type of the <i>key</i> component of a <i>key-value</i> pair.
        /// </summary>
        /// <returns>A Type instance containing the metadata about the type of the
        /// <i>key</i>.</returns>
        public override Type GetKeyType(){ return typeof(K); }
        /// <summary>
        /// Gets the type of the <i>value</i> component of a <i>key-value</i> pair.
        /// </summary>
        /// <returns>A Type instance containing the metadata about the type of the
        /// <i>value</i>.</returns>
        public overrideType GetValueType(){ return typeof(V); }
        #region Template Methods
        /// <summary>
        /// Function Map is overrided by users to define a map function.
        /// </summary>
        /// <param name = "source">The source of Map function is IMapInput, which
            contains
        /// a key/value pair.</param>
        protected abstract void Map(IMapInput<K, V> input);
        #endregion
    }
}
```

Listing 1 shows in detail the definition of the *Mapper<K,V>* class and of the related types that developers should be aware of for implementing the *map* function. In order to implement a specific mapper, it is necessary to inherit this class and provide actual types for key *K* and the value *V*. The map operation is implemented by overriding the abstract method *void Map(IMapInput<K,V> input)*, while the other methods are internally used by the framework. *IMapInput<K,V>* provides access to the input key-value pair on which the map operation is performed.

**Listing 2.** Simple Mapper<K,V> Implementation.

```
using Aneka.MapReduce;
namespace Aneka.MapReduce.Examples.WordCounter
{
    /// <summary>
    /// Class WordCounterMapper. Extends Mapper<K,V> and provides an
    /// implementation of the map function for the Word Counter sample. This mapper
```

```
    /// emits a key-value pair (word,1) for each word encountered in the input
        line.
    /// </summary>
    public class WordCounterMapper: Mapper<long,string>
    {
        /// <summary>
        /// Reads the source and splits into words. For each of the words found
        /// emits the word as a key with a vaue of 1.
        /// </summary>
        /// <param name = "source">map source</param>
        protected override void Map(IMapInput<long,string> input)
        {
            // we don't care about the key, because we are only interested on
            // counting the word of each line.
            string value = input.Value;
            string[] words = value.Split(" \t\n\r\f\"\'|!-=()[]<>:{}.#".
            ToCharArray(), StringSplitOptions.RemoveEmptyEntries);
            // we emit each word without checking for repetitions. The word becomes
            // the key and the value is set to 1, the reduce operation will take care
            // of merging occurrences of the same word and summing them.
            foreach(string word in words)
            {
                this.Emit(word, 1);
            }
        }
    }
}
```

Listing 2 shows the implementation of the *Mapper<K,V>* component for the Word Counter sample. This sample counts the frequency of words into a set of large text files.The text files are divided into lines, and each of the lines will become the value component of a key-value pair, while the key will be represented by the offset in the file where the line begins. Therefore, the mapper is specialized by using a *long integer* as key type and a *string* for the value. In order to count the frequency of words, the map function will emit a new key-value pair for each of the words contained in the line, by using the word as the key and the number 1 as value. This implementation will emit two pairs for the same word, if the word occurs twice in the line. It will be the responsibility of the reducer to sum appropriately all these occurrences.

**Listing 3.** Reduce Function APIs.

```
using Aneka.MapReduce.Internal;
namespace Aneka.MapReduce
{
    /// <summary>
    /// Delegate ReduceEmitDelegate. Defines the signature of a method that is used
    /// to emit aggregated value of a collection of values matching the same key
    /// and that is generated by a reducer.
    /// </summary>
```

```
/// <param name = "value">The <i>value</i> of the <i>key-value</i> pair.</param>
public delegate void ReduceEmitDelegate(object value);
/// <summary>
/// Class <i>Reducer</i>. Extends the ReducerBase class and provides an
/// implementation of the common operations that are expected from a
    <i>Reducer</i>.
/// In order to define reducer for specific applications developers have to
    extend implementation of the Reduce (IReduceInputEnumerator<V>) method that
/// reduces a this class and provide an collection of <i>key-value</i> pairs as
/// described by the <i>map-reduce</i> model.
/// </summary>
public abstract class Reducer<K,V> : ReducerBase
{
    /// <summary>
    /// Performs the <i>reduce</i> phase of the <i>map-reduce</i> model.
    /// </summary>
    /// <param name = "source">An instance of IReduceInputEnumerator allowing to
    /// iterate over the collection of values that have the same key and will be
    /// aggregated.</param>
    /// <param name = "emit">An instance of the ReduceEmitDelegate that is used to
    /// write to the output stream the aggregated value.</param>
    public void Reduce(IReduceInputEnumerator input, ReduceEmitDelegate emit)
    { … }
    /// <summary>
    /// Gets the type of the <i>key</i> component of a <i>key-value</i> pair.
    /// </summary>
    /// <returns>A Type instance containing the metadata about the type of the
    /// <i>key</i>.</returns>
    public override Type GetKeyType(){return typeof(K);}
    /// <summary>
    /// Gets the type of the <i>value</i> component of a <i>key-value</i> pair.
    /// </summary>
    /// <returns>A Type instance containing the metadata about the type of the
    /// <i>value</i>.</returns>
    public override Type GetValueType(){return typeof(V);}
    #region Template Methods
    /// <summary>
    /// Recuces the collection of values that are exposed by
    /// <paramref name = "source"/> into a single value. This method implements
    /// the <i>aggregation</i> phase of the <i>map-reduce</i> model, where
    /// multiple values matching the same key are composed together to generate
    /// a single value.
    /// </summary>
    /// <param name = "source">AnIReduceInputEnumerator<V> instancethat allows to
```

```
        /// iterate over all the values associated with same key.</param>
        protected abstract void Reduce(IReduceInputEnumerator<V> input);
        #endregion
    }
}
```

The listing above shows the definition of the *Reducer<K,V>* class. The implementation of a specific reducer requires specializing the generic class and overriding the abstract method: *Reduce(IReduceInputEnumerator<V> input)*. Since the reduce operation is applied to a collection of values that are mapped to the same key, the *IReduceInputEnumerator<V>* allows developers to iterate over such collection. Listing 4 shows how to implement the reducer function for the word counter sample.

**Listing 4.** Simple Reducer<K,V> Implementation.

```
using Aneka.MapReduce;
namespace Aneka.MapReduce.Examples.WordCounter
{
    /// <summary>
    /// Class <b><i>WordCounterReducer</i></b>. Reducer implementation for the Word
    /// Counter application. The Reduce method iterates all over values of the
    /// enumerator and sums the values before emitting the sum to the output file.
    /// </summary>
    public class WordCounterReducer: Reducer<string,int>
    {
        /// <summary>
        /// Iterates all over the values of the enumerator and sums up
        /// all the values before emitting the sum to the output file.
        /// </summary>
        /// <param name = "source">reduce source</param>
        protected override void Reduce(IReduceInputEnumerator<int>input)
        {
            int sum = 0;
            while(input.MoveNext())
            {
                int value = input.Current;
                sum += value;
            }
            this.Emit(sum);
        }
    }
}
```

In this case, the *Reducer<K,V>* class is specialized using a *string* as a key type and an *integer* as a value. The reducer simply iterates over all the values that are accessible through the enumerator and sums them. Once the iteration is completed, the sum is dumped to a file.

It is important to notice that there is a link between the types used to specialize the mapper and those used to specialize the reducer. The key and value types used in the reducer are those defining the key-value pair emitted by the mapper. In this case, the mapper generates a key-value pair *(string,int)*, hence the reducer is of type *Reducer<string,int>*.

The *Mapper<K,V>* and *Reducer<K,V>* classes provides facilities for defining the computation performed by a MapReduce job. In order to submit, execute, and monitor its progress, Aneka provides the *MapReduceApplication<M,R>* class. As happens for the other programming models introduced in this book, this class represents the local view of distributed applications on top of Aneka. Due to the simplicity of the MapReduce model, such class provides limited facilities that are mostly concerned with starting the MapReduce job and waiting for its completion. Listing 5 shows the interface of *MapReduceApplication<M,R>*.

**Listing 5.** MapReduceApplication<M,R>.

```
using Aneka.MapReduce.Internal;
namespace Aneka.MapReduce
{
    /// <summary>
    /// Class <b><i>MapReduceApplication</i></b>. Defines a distributed application
    /// based on the MapReduce Model. It extends the ApplicationBase<M> and
        specializes
    /// it with the MapReduceManager<M,R> application manager. A
        MapReduceApplication
    /// is a generic type that is parameterized with a specific
    /// type of MapperBase and a specific type of ReducerBase. It controls the
    /// execution of the application and it is in charge of collecting the results
        or resubmitting the failed tasks.
    /// </summary>
    /// <typeparam name = "M">Placeholder for the mapper type.</typeparam>
    /// <typeparam name = "R">Placeholder for the reducer type.</typeparam>
    public class MapReduceApplication<M, R> : ApplicationBase<MapReduceManager<M, R>>
           where M: MapReduce.Internal.MapperBase
           where R: MapReduce.Internal.ReducerBase
    {
        /// <summary>
        /// Default value for the Attempts property.
        /// </summary>
        public const intDefaultRetry = 3;
        /// <summary>
        /// Default value for the Partitions property.
        /// </summary>
        public const intDefaultPartitions = 10;
        /// <summary>
        /// Default value for the LogFile property.
        /// </summary>
        public const stringDefaultLogFile = "mapreduce.log";
        /// <summary>
        /// List containing the result files identifiers.
```

```
/// </summary>
private List<string>resultFiles = new List<string>();
/// <summary>
/// Property group containing the settings for the MapReduce application.
/// </summary>
private PropertyGroupmapReduceSetup;
/// <summary>
/// Gets, sets an integer representing the number of partions for the key
    space.
/// </summary>
public int Partitions { get { … } set { … } }
/// <summary>
/// Gets, sets an boolean value indicating in whether to combine the result
/// after the map phase in order to decrease the number of reducers used in
/// the reduce phase.
/// </summary>
public bool UseCombiner { get { … } set { … } }
/// <summary>
/// Gets, sets an boolean indicating whether to synchronize the reduce
    phase.
/// </summary>
public bool SynchReduce { get { … } set { … } }
/// <summary>
/// Gets or sets a boolean indicating whether the source files required by
/// the required by the application is already uploaded in the storage or
    not.
/// </summary>
public bool IsInputReady { get { … } set { … } }
/// <summary>
/// Gets, sets the number of attempts that to run failed tasks.
/// </summary>
public int Attempts { get { … } set { … } }
/// <summary>
/// Gets or sets a string value containing the path for the log file.
/// </summary>
public string LogFile { get { … } set { … } }
/// <summary>
/// Gets or sets a boolean indicating whether application should download
/// the result files on the local client machine at the end of the execution
    or not.
/// </summary>
public bool FetchResults { get { … } set { … } }
/// <summary>
/// Creates a MapReduceApplication<M,R> instance and configures it with
/// the given configuration.
```

```
        /// </summary>
        /// <param name = "configuration">A Configuration instance containing the
        /// information that customizes the execution of the application.</param>
        public MapReduceApplication(Configurationconfiguration):
                                    base("MapReduceApplication", configuration){ … }
        /// <summary>
        /// Creates MapReduceApplication<M,R> instance and configures it with
        /// the given configuration.
        /// </summary>
        /// <param name = "displayName">A string containing the friendly name of the
        /// application.</param>
        /// <param name = "configuration">A Configuration instance containing the
        /// information that customizes the execution of the application.</param>
        public MapReduceApplication(string displayName, Configuration configuration):
                                    base(displayName, configuration) { … }
        // here follows the private implementation…
    }
}
```

The interface of the class exhibits only the MapReduce specific settings, while the control logic is encapsulated in the *ApplicationBase<M>* class. From this class, it is possible set the behavior of MapReduce for the current execution. The parameters that can be controlled are the following:

**(a) Partitions.** This property stores an integer number containing the number of partitions into which divide the final results. This value determines also the number of reducer tasks that will be created by the runtime infrastructure. The default value is 10.

**(b) Attempts.** This property contains the number of times that the runtime will retry to execute a task before declaring it failed. The default value is 3.

**(c) UseCombiner.** This property stores a boolean value indicating whether the MapReduce runtime should add a combiner phase to the map task execution in order to reduce the number of intermediate files that are passed to the reduce task. The default value is set to *true*.

**(d) SynchReduce.** This property stores a boolean value indicating whether to synchronize the reducers or not. The default value is set to *true,* and currently is not used to determine the behavior of MapReduce.

**(e) IsInputReady.** This is a boolean property indicating whether the input files are already stored into the distributed file system or have to uploaded by the client manager before executing the job. The default value is set to *false*.

**(f) FetchResults.** This is a boolean property indicating whether the client manager needs to download to the local computer the result files produced by the execution of the job. The default value is set to *true*.

**(g) LogFile.** This property contains a string defining the name of the log file used to store the performance statistics recorded during the execution of the MapReduce job. The default value is *mapreduce.log*.

The core control logic governing the execution of a MapReduce job resides within the *MapReduceApplicationManager<M,R>*, which interacts with the MapReduce runtime. Developers can control the application by using the methods and the properties exposed by the *ApplicationBase<M>* class. Listing 6 displays the collection of methods that are of interest in this class for the execution of MapReduce jobs.

**Listing 6.** ApplicationBase<M >.

```
Namespace Aneka.Entity
{
    /// <summary>
    /// Class <b><i>ApplicationBase<M></i></b>. Defines the base class for the
    /// application instances for all the programming model supported by Aneka.
    /// </summary>
    public class ApplicationBase<M> where M : IApplicationManager, new()
    {
        /// <summary>
        /// Gets the application unique identifier attached to this instance. The
        /// application unique identifier is the textual representation of a System.
        /// Guid instance, therefore is a globally unique identfier. This identifier is
        /// automatically created when a new instance of an application is created.
        /// </summary>
        public string Id { get { … } }
        /// <summary>
        /// Gets the unique home directory for the AnekaApplication<W,M>.
        /// </summary>
        public string Home { get { … } }
        /// <summary>
        /// Gets the current state of the application.
        /// </summary>
        public ApplicationState State{get{ … }}
        /// <summary>
        /// Gets a boolean value indicating whether the application is terminated.
        /// </summary>
        public bool Finished { get { … } }
        /// <summary>
        /// Gets the underlying IApplicationManager that ismanaging the execution
        /// of the application instanceon the client side.
        /// </summary>
        public M ApplicationManager { get { … } }
        /// <summary>
        /// Gets, sets the application display name. This is a friendly name which is
        /// to identify an application by means of a textual and human intelligible
        /// sequence of characters, but it is NOT a unique identifier and no check
            about
        /// uniqueness of the value of this property is done. For a unique
            indentifier
        /// please check the Id property.
```

```
/// </summary>
public string DisplayName { get { … } set { … } }
/// <summary>
/// Occurs when the application instance terminates its execution.
/// </summary>
public event EventHandler<ApplicationEventArgs> ApplicationFinished;
/// <summary>
/// Creates an application instance with the given settings and sets the
/// application display name to null.
/// </summary>
/// <param name = "configuration">Configuration instance specifying the
/// application settings.</param>
public ApplicationBase(Configuration configuration): this(null, configuration)
{ … }
/// <summary>
/// Creates an application instance withthe given settings and display
/// name. As a result of the invocation, a new application unique identifier
/// is created and the underlying application manager is initialized.
/// </summary>
/// <param name = "configuration">Configurationinstance specifying the
    application
/// settings.</param>
/// <param name = "displayName">Application friendly name.</param>
public ApplicationBase(string displayName, Configuration configuration){ … }
/// <summary>
/// Starts the execution of the application instanceon Aneka.
/// </summary>
public void SubmitExecution() { … }
/// <summary>
/// Stops the execution of the entire application instance.
/// </summary>
public void StopExecution() { … }
/// <summary>
/// Invoke the application and wait until the application finishes.
/// </summary>
public void InvokeAndWait() { this.InvokeAndWait(null); }
/// <summary>
/// Invoke the application and wait until the application finishes, then
/// invokes the given callback.
/// </summary>
/// <param name = "handler">A pointer to a method that is executed at the
    end of
/// the application.</param>
```

```
        public void InvokeAndWait(EventHandler<ApplicationEventArgs> handler) { … }
        /// <summary>
        /// Adds a shared file to the application.
        /// </summary>
        /// <param name="file">A string containing the path to the file to add.</
            param>
        public virtual void AddSharedFile(string file) { … }
        /// <summary>
        /// Adds a shared file to the application.
        /// </summary>
        /// <param name="file">A FileData instance containing the information about
            the
        /// file to add.</param>
        public virtual void AddSharedFile(FileData fileData) { … }
        /// <summary>
        /// Removes a file from the list of the shared files of the application.
        /// </summary>
         /// <param name="file">A string containing the path to the file to
        /// remove.</param>
        public virtual void RemoveSharedFile(string filePath) { … }
        // here come the private implementation.
    }
}
```

Besides the constructors and the common properties that are of interest in all the applications, the two methods in bold are those that are most commonly used to execute MapReduce jobs. These are two different overloads of the *InvokeAndWait* method: the first one simply starts the execution of the MapReduce job and returns upon its completion, while the second one executes a client supplied callback at the end of the execution. The use of *InvokeAndWait* is blocking, therefore, it is not possible to stop the application by calling *StopExecution* within the same thread. If it is necessary to implement a more sophisticated management of the MapReduce job, it is possible to use the *SubmitExecution* method, which submits the execution of the application and returns without waiting for its completion.

In terms of management of files, the MapReduce implementation will automatically upload all the files that are found in the *Configuration.Workspace* directory and will ignore the files added by the *AddSharedFile* methods.

Listing 7 shows how to create a MapReduce application for running the Word Counter sample defined by the previous *WordCounterMapper* and *WordCounterReducer* classes.

**Listing 7.** WordCounter Job.

```
using System.IO;
using Aneka.Entity;
using Aneka.MapReduce;
namespace Aneka.MapReduce.Examples.WordCounter
{
    /// <summary>
    /// Class <b><i>Program<M></i></b>. Application driver for the Word Counter
        sample.
```

```
/// </summary>
public class Program
{
    /// <summary>
    /// Reference to the configuration object.
    /// </summary>
    private static Configuration configuration = null;
    /// <summary>
    /// Location of the configuration file.
    /// </summary>
    private static string confPath = "conf.xml";
    /// <summary>
    /// Processes the arguments given to the application and according
    /// to the parameters read runs the application or shows the help.
    /// </summary>
    /// <param name="args">program arguments</param>
    private static void Main(string[] args)
    {
        try
        {
            Logger.Start();
            // get the configuration
            configuration = Configuration.GetConfiguration(confPath);
            // configure MapReduceApplication
            MapReduceApplication<WordCountMapper, WordCountReducer> application =
                new MapReduceApplication<WordCountMapper, WordCountReducer>
                    ("WordCounter", configuration);
            // invoke and wait for result
            application.InvokeAndWait(new
                                EventHandler<ApplicationEventArgs>(OnDone));
            // alternatively we can use the following call
           // application.InvokeAndWait();
        }
        catch(Exception ex)
        {
            Usage();
            IOUtil.DumpErrorReport(ex, "Aneka WordCounter Demo - Error Log");
        }
        finally
        {
            Logger.Stop();
```

```
            }
        }
        /// <summary>
        /// Hooks the ApplicationFinished events and Process the results
        /// if the application has been successful.
        /// </summary>
        /// <param name = "sender">event source</param>
        /// <param name = "e">event information</param>
        private static void OnDone(object sender, ApplicationEventArgs e) { … }
        /// <summary>
        /// Displays a simple informative message explaining the usage of the
        /// application.
        /// </summary>
        private static void Usage() { … }
    }
}
```

The lines of interest are those put in evidence in the *try* { … } *catch* { … } *finally* { …} block. As it can be seen, the execution of a MapReduce Job requires only three lines of code, where the user reads the configuration file, creates a *MapReduceApplication<M,R>* instance and configures it, and then starts the execution. All the rest of the code is mostly concerned with setting up the logging and handling exceptions.

## 2. Runtime Support

The runtime support for the execution of MapReduce jobs is constituted by the collection of services that deal with the scheduling and the execution of MapReduce tasks. These are *MapReduce Scheduling Service* and the *MapReduce Execution Service*. These two services integrate with the existing services of the framework in order to provide persistence, application accounting, and the features available for the applications developed with other programming models.

**(a) Job and Task Scheduling.** The scheduling of jobs and tasks is the responsibility of the *MapReduce Scheduling Service*, which covers the same role of the master process in the Google MapReduce implementation. The architecture of the scheduling service is organized into two major components: the *MapReduceSchedulerService* and the *MapReduceScheduler*. The former is a wrapper around the scheduler implementing the interfaces required by Aneka to expose a software component as a service, while the latter controls the execution of jobs and schedules tasks. Therefore, the main role of the service wrapper is to translate messages coming from the Aneka runtime or the client applications into calls or events directed to the scheduler component and vice versa. The relationship of two components is depicted in Fig. 8.9.

The core functionalities for job and tasks scheduling are implemented in the *MapReduceScheduler* class. The scheduler manages multiple queues for several operations such as: uploading input files into the distributed file system; initializing jobs before scheduling; scheduling map and reduce tasks; keeping track of unreachable nodes; re-submitting failed tasks; and reporting execution statistics. All these operations are performed asynchronously and triggered by events happening in the Aneka middleware.

**(b) Task Execution.** The execution of tasks is controlled by the *MapReduce Execution Service*. This component plays the role of the worker process in the Google MapReduce implementation. The service manages the execution of map and reduce tasks, and also performs other operations such as sorting and merging of intermediate files. The service is internally organized as described in Fig. 8.10.

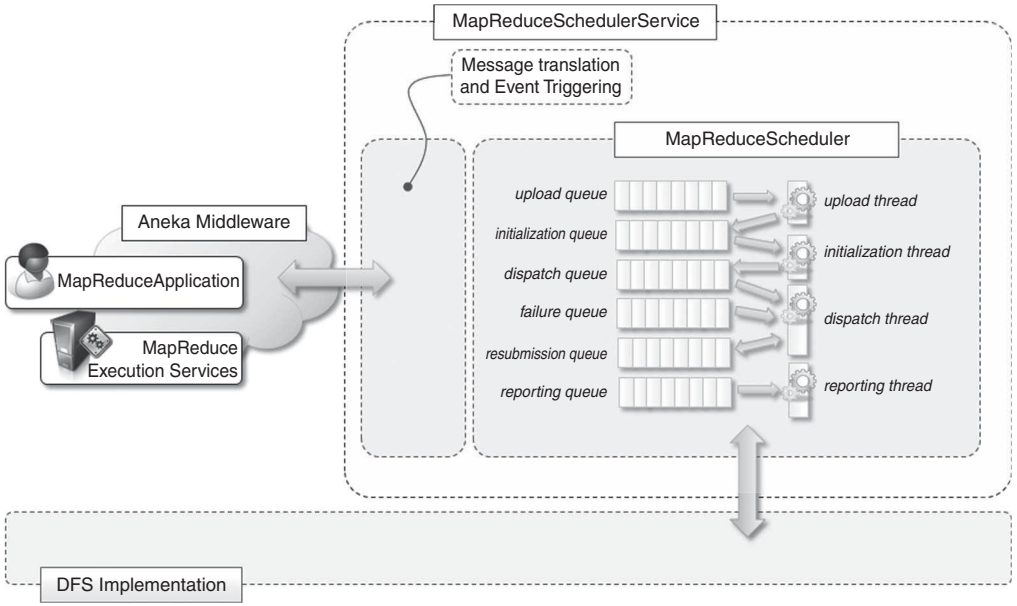


**Fig. 8.9.** MapReduce Scheduling Service Architecture.

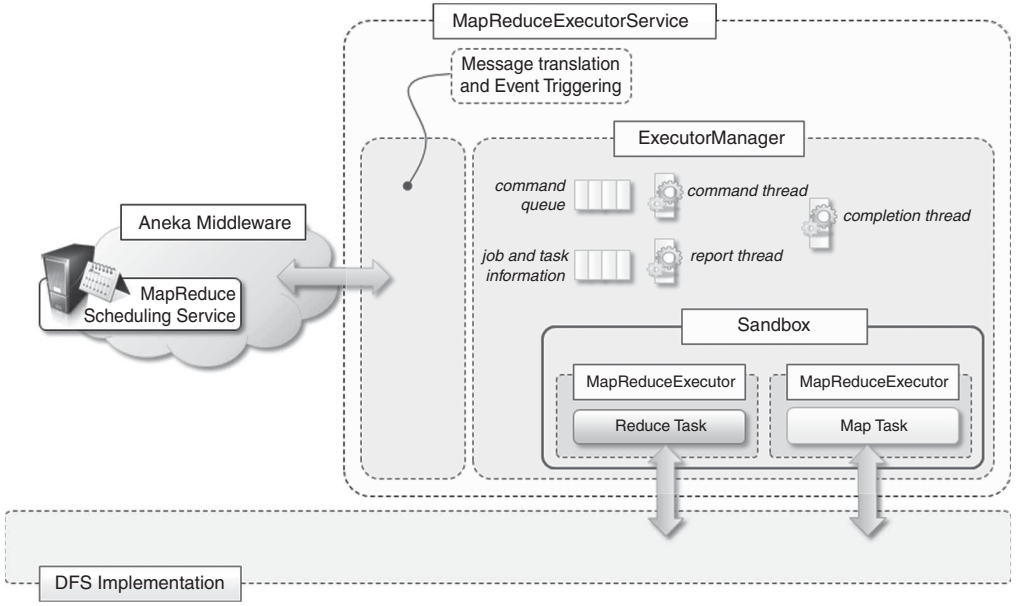


**Fig. 8.10.** MapReduce Execution Service Architecture.

There are three major components that coordinate together for executing tasks: *MapReduce-SchedulerService*, *ExecutorManager*, and *MapReduceExecutor*. The *MapReduceSchedulerService* interfaces the *ExecutorManager* with the Aneka middleware, while the *ExecutorManager* is in charge of keeping track of the tasks being executed by demanding the specific execution of a task to the *MapReduceExecutor* and of sending the statistics about the execution back to the scheduler service.

It is possible to configure more than one *MapReduceExecutor* instances, and this is helpful in case of multi-core nodes where more than one task can be executed at the same time.

## 3. Distributed File System Support

Different from the other programming models supported by Aneka, the MapReduce model does not leverage the default storage service for storage and data transfer but uses a distributed file system implementation. The reason for this is because the requirements in terms of file management are significantly different with respect to the other models. In particular, MapReduce has been designed to process large quantities of data stored in files of large dimensions. Therefore, the support provided by a distributed file system, which can leverage multiple nodes for storing data, is more appropriate. Distributed file system implementations guarantee high availability and a better efficiency by means of replication and distribution. Moreover, the original MapReduce implementation assumes the existence of a distributed and reliable storage; hence, the use of a distributed file system for implementing the storage layer is natural.

Aneka provides the capability of interfacing with different storage implementations as described in earlier chapter (Section 5.2.3), and it maintains the same flexibility for the integration of a distributed file system. The level of integration required by MapReduce requires the ability to perform the following tasks:

- Retrieving the location of files and file chunks
- Accessing a file by means of a stream

The first operation is useful to the scheduler for optimizing the scheduling of *map* and *reduce* tasks according to the location of data, while the second operation is required for the usual I/O operations to and from data files. In case of a distributed file system, the stream might also access the network, if the file chunk is not stored on the local node. Aneka provides interfaces that allow performing such operations and the capability to plug different file systems behind them by providing the appropriate implementation. The current implementation provides bindings to HDFS.

On top of these low-level interfaces, the MapReduce programming model offers classes to read from and write to files in a sequential manner. These are the classes *SeqReader* and *SeqWriter*. They provide sequential access for reading and writing key-value pairs and they expect a specific file format, which is described in Fig. 8.11.

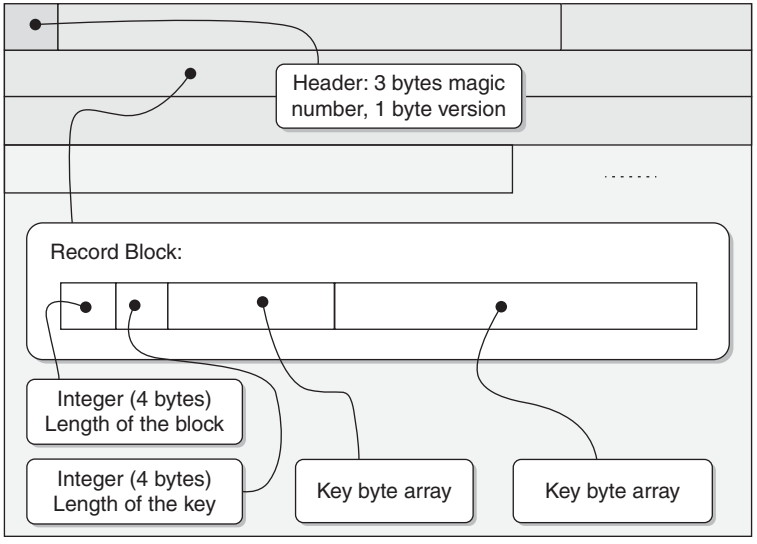


**Fig. 8.11.** Aneka MapReduce Data File Format.

An Aneka MapReduce file is composed by a header used to identify the file and a sequence of record blocks, each of them storing a key-value pair. The header is composed of 4 bytes: the first three bytes represent the character sequence SEQ, and the fourth byte identifies the version of the file. The record block is composed as follows: the first 8 bytes are used to store two integers representing the length of the rest of the block and the length of the key section, which is immediately following. The remaining part of the block stores the data of the value component of the pair. The *SeqReader* and *SeqWriter* classes are designed to read and write files in this format by transparently handling the file format information, and translating key and value instances to and from their binary representation. All the .NET built-in types are supported. Since MapReduce jobs operate very often with data represented in common text files, a specific version of the *SeqReader* and *SeqWriter* classes have been designed to read and write text files as a sequence of key-value pairs. In the case of the read operation, each value of the pair is represented by a line in the text file, while the key is automatically generated and assigned to the position in bytes where the line starts in the file. In case of the write operation, the writing of the key is skipped and the values are saved as single lines.

Listing 8 shows the interface of the *SeqReader* and *SeqWriter* classes. The *SeqReader* class provides an enumerator-based approach, through which it is possible to access the key and the value sequentially by calling the *NextKey()* and the *NextValue()* methods respectively. It is also possible to access the raw byte data of keys and values by using the *NextRawKey*() and *NextRawValue*(). The *HasNext()* returns a Boolean indicating whether there are more pairs to read or not. The *SeqWriter* class exposes different versions of the *Append* method.

**Listing 8.** SeqReader and SeqWriter Classes.

```
using Aneka.MapReduce.Common;
namespace Aneka.MapReduce.DiskIO
{
    /// <summary>
    /// Class <b><i>SeqReader</i></b>. This class implements a file reader for the
    /// sequence file, which isa standard file split used by MapReduce.NET to store a
    /// partition of a fixed size of a data file. This classprovides an interface for
    /// exposing the content of a file split as an enumeration of key-value pairs
    /// and offersfacilities for both accessing keys and values as objects and
        their corresponding binary values.
    /// </summary>
    public class SeqReader
    {
        /// <summary>
        /// Creates a SeqReader instance and attaches it to the given file. This
        /// constructor initializes the instance with the default value for the
        /// internal buffers and does not set any information about the types of
        /// the keys and values read from the file.
        /// </summary>
        public SeqReader(string file) : this(file, null, null) { … }
        /// <summary>
        /// Creates a SeqReader instance, attaches it to the given file, and sets
        /// the internal buffer size to bufferSize. This constructor does not
        /// provide any information about the types of the keys and values read
            from the file.
        /// </summary>
```

```
public SeqReader(string file, int bufferSize) :
this(file,null,null,bufferSize) { … }
/// <summary>
/// Creates a SeqReader instance, attaches it to the given file, and
/// provides metadata information about the content of the file in the form
/// of keyType and valueType. The internal buffers are initialized with the
    default dimension.
/// </summary>
public SeqReader(string file, Type keyType, Type valueType)
    : this(file, keyType, valueType, SequenceFile.DefaultBufferSize) { … }
/// <summary>
/// Creates a SeqReader instance, attaches it to the given file, and
/// provides metadata information about the content of the file in the form
/// of keyType and valueType. The internal buffers are initialized with the
    bufferSize dimension.
/// </summary>
public SeqReader(string file, Type keyType, Type valueType, int bufferSize)
{ … }
/// <summary>
/// Sets the metadata information about the keys and the values contained
/// in the data file.
/// </summary>
public void SetType(Type keyType, Type valueType) { … }
/// <summary>
/// Checks whether there is another record in the data file and moves the
/// current file pointer to its beginning.
/// </summary>
public bool HaxNext() { … }
/// <summary>
/// Gets the object instance corresponding to the next key in the data file.
/// in the data file.
/// </summary>
public object NextKey() { … }
/// <summary>
/// Gets the object instance corresponding to the next value in the data.
/// file in the data file.
/// </summary>
public object NextValue() { … }
/// <summary>
/// Gets the raw bytes that contain the value of the serializedinstance of
/// the current key.
/// </summary>
public BufferInMemory NextRawKey() { … }
/// <summary>
/// Gets the raw bytes that contain the value of the serialized instance of
```

```
        /// the current value.
        /// </summary>
        public BufferInMemory NextRawValue() { … }
        /// <summary>
        /// Gets the position of the file pointer as an offset from its beginning.
        /// </summary>
        public long CurrentPosition() { … }
        /// <summary>
        /// Gets the size of the file attached to this instance of SeqReader.
        /// </summary>
        public long StreamLength() { … }
        /// <summary>
        /// Moves the file pointer to position. If the value of position is 0 or
        /// negative, returns the current position of the file pointer.
        /// </summary>
        public long Seek(long position) { … }
        /// <summary>
        /// Closes the SeqReader instanceand releases all the resources that have
        /// been allocated to read fromthe file.
        /// </summary>
        public void Close() { … }
        // private implementation follows
    }
    /// <summary>
    /// Class SeqWriter. This class implements a file writer for the sequence
    /// sequence file, which is a standard file split used by MapReduce.NET to store a
    /// partition of a fixed size of a data file. This classprovides an interface to
    /// add a sequence of key-value pair incrementally.
    /// </summary>
    public class SeqWriter
    {
        /// <summary>
        /// Creates a SeqWriter instance for writing to file. This constructor
        /// initializes the instance with the default value for the internal
            buffers.
        /// </summary>
        public SeqWriter(string file) : this(file, SequenceFile.DefaultBufferSize){ … }
        /// <summary>
        /// Creates a SeqWriter instance, attachesit to the given file, and sets the
        /// internal buffer size to bufferSize.
        /// </summary>
        public SeqWriter(string file, int bufferSize) { … }
        /// <summary>
        /// Appends a key-value pair to the data file split.
```

```
        /// </summary>
        public void Append(object key, object value) { … }
        /// <summary>
        /// Appends a key-value pair to the data file split.
        /// </summary>
        public void AppendRaw(byte[] key, byte[] value) { … }
        /// <summary>
        /// Appends a key-value pair to the data file split.
        /// </summary>
        public void AppendRaw(byte[] key, int keyPos, int keyLen,
                              byte[] value, int valuePos, int valueLen) { … }
        /// <summary>
        /// Gets the length of the internal buffer or 0 if no buffer has been
            allocated.
        /// </summary>
        public longLength() { … }
        /// <summary>
        /// Gets the length of data file split on disk so far.
        /// </summary>
        public long FileLength() { … }
        /// <summary>
        /// Closes the SeqReader instance and releases all the resources that have
        /// been allocated to write to the file.
        /// </summary>
        public void Close() { … }
        // private implementation follows
    }
}
```

Listing 9 shows a practical use of the *SeqReader* class by implementing the callback used in the Word Counter sample. In order to visualize the results of the application, we use the *SeqReader* class to read the content of the output files and dump it into a proper textual form that can be visualized with any text editor, such as the Notepad application.

**Listing 9.** WordCounter Job (…continues).

```
using System.IO;
using Aneka.Entity;
using Aneka.MapReduce;
namespace Aneka.MapReduce.Examples.WordCounter
{
    /// <summary>
    /// Class Program. Application driver for the Word Counter sample.
    /// </summary>
    public class Program
    {
```

```
/// <summary>
/// Reference to the configuration object.
/// </summary>
private static Configuration configuration = null;
/// <summary>
/// Location of the configuration file.
/// </summary>
private static string confPath = "conf.xml";
/// <summary>
/// Processes the arguments given to the application and according
/// to the parameters read runs the application or shows the help.
/// </summary>
/// <param name = "args">program arguments</param>
private static void Main(string[] args)
{
    try
    {
        Logger.Start();
        // get the configuration
        Program.configuration = Configuration.GetConfiguration(confPath);
        // configure MapReduceApplication
        MapReduceApplication<WordCountMapper, WordCountReducer> application =
           new MapReduceApplication<WordCountMapper, WordCountReducer>
           ("WordCounter",configuration);
        // invoke and wait for result
        application.InvokeAndWait(newEventHandler<ApplicationEventArgs>
        (OnDone));
        // alternatively we can use the following call
        // application.InvokeAndWait();
    }
    catch(Exception ex)
    {
        Program.Usage();
        IOUtil.DumpErrorReport(ex, "Aneka WordCounter Demo - Error Log");
    }
    finally
    {
        Logger.Stop();
    }
}
/// <summary>
/// Hooks the ApplicationFinished events and process the results
/// if the application has been successful.
```

```
/// </summary>
/// <param name = "sender">event source</param>
/// <param name = "e">event information</param>
private static void OnDone(object sender, ApplicationEventArgs e)
{
    if (e.Exception != null)
    {
      IOUtil.DumpErrorReport(e.Exception, "Aneka WordCounter Demo -
      Error");
    }
    else
    {
        string outputDir = Path.Combine(configuration.Workspace, "output");
        try
        {
            FileStream resultFile = new FileStream("WordResult.
            txt",FileMode.Create, FileAccess.Write);
            Stream WritertextWriter = new StreamWriter(resultFile);
            DirectoryInfo sources = new DirectoryInfo(outputDir);
            FileInfo[] results = sources.GetFiles();
            foreach(FileInfo result in results)
            {
                SeqReader seqReader = newSeqReader(result.FullName);
                seqReader.SetType(typeof(string), typeof(int));
                while(seqReader.HaxNext() == true)
                {
                   object key = seqReader.NextKey();
                   object value = seqReader.NextValue();
                   textWriter.WriteLine("{0}\t{1}", key, value);
                }
                seqReader.Close();
            }
            textWriter.Close();
            resultFile.Close();
            // clear the output directory
            sources.Delete(true);
            Program.StartNotePad("WordResult.txt");
        }
        catch(Exception ex)
        {
            IOUtil.DumpErrorReport(e.Exception, "Aneka WordCounter Demo -
            Error");
        }
```

```
            }
        }
        /// <summary>
        /// Starts the notepad process and displays the given file.
        /// </summary>
        private static voidStartNotepad(string file) { … }
        /// <summary>
        /// Displays a simple informative message explaining the usage of the
        /// application.
        /// </summary>
        private static void Usage() { … }
    }
}
```

The *OnDone* callback checks whether the application has terminated successfully. In case there are no errors, it iterates over the result files downloaded in the workspace output directory. By default, the files are saved in the *output* subdirectory of the workspace directory. For each of the result files, it opens a *SeqReader* instance on it and dumps the content of the key-value pair into a text file, which can be opened by any text editor.

## 8.3.2 Example Application

MapReduce is a very useful model for processing large quantities of data, which in many cases are maintained in a semi-structured form such as logs or Web pages. In order to demonstrate how to program real applications with Aneka MapReduce, we consider a very common task: log parsing. We design a MapReduce application that processes the logs produced by the Aneka container in order to extract some summary information about the behavior of the Cloud. In this section, we describe in detail the problem to be addressed and design the Mapper and Reducer classes that are used to execute log parsing and data extraction operations.

### *1. Parsing Aneka Logs*

Aneka components (daemons, container instances, and services) produce a lot of information that are stored in the form of log files. The most relevant information is stored into the container instances logs, which store the information about the applications that are executed on the Cloud. In this example, we parse these logs in order to extract useful information about the execution of applications and the usage of services in the Cloud.

The entire framework leverages the *log4net* library for collecting and storing the log information. In the case of Aneka containers, the system is configured to produce a log file that is partitioned in chunks every time the container instance restarts. Moreover, the information contained in the log file can be customized in their appearance and currently the default layout is the following:

*DD MMM YY hh:mm:ss level –message*

Some examples of formatted log messages are the following:

*15 Mar 2011 10:30:07 DEBUG - SchedulerService:HandleSubmitApplication - SchedulerService: …*
*15 Mar 2011 10:30:07 INFO - SchedulerService: Scanning candidate storage …*
*15 Mar 2011 10:30:10 INFO - Added [WU: 51d55819-b211-490f-b185-8a25734ba705, 4e86fd02…*
*15 Mar 2011 10:30:10 DEBUG - StorageService:NotifyScheduler - Sending FileTransferMessage…*
*15 Mar 2011 10:30:10 DEBUG - IndependentSchedulingService:QueueWorkUnit–Queueing…*

*15 Mar 2011 10:30:10 INFO - AlgorithmBase::AddTasks[64] Adding 1 Tasks*
*15 Mar 2011 10:30:10 DEBUG - AlgorithmBase:FireProvisionResources - Provision Resource: 1*

In the content of the sample log lines, we observe that the message part of almost all the log lines exhibit a similar structure, and they start with the information about the component that enters the log line. This information can be easily extracted by looking at the first occurrence of the ':' character following a sequence of characters that do not contain spaces.

Possible information that we might want to extract from such logs is the following:

- The distribution of log messages according to the level.
- The distribution of log messages according to the components.

This information can be easily extracted and composed into a single view by creating *Mapper* tasks that count the occurrences of log levels and component names, and emit one simple key-value pair in the form *(level-name, 1)* or *(component-name, 1)* for each of the occurrences. The *Reducer* task will simply sum up all the key-value pairs that have the same key. For both problems, the structure of the *map* and *reduce* functions will be the following:

*map: (long, string) => (string, long)*

*reduce: (string, long) => (string, long)*

The *Mapper* class will then receive a key-value pair containing the position of the line inside the file as a key and the log message as the value component. It will produce a key-value pair containing a string representing the name of the log level or the component name and 1 as value. The *Reducer* class will sum up all the key value pairs that have the same name. By modifying the canonical structure discussed above, we can perform both of the analyses at the same time instead of developing two different MapReduce jobs. It can be noticed that the operation performed by the Reducer class is the same in both cases, while the operation of the *Mapper* class changes but the type of the key-value pair that is generated is the same for the two jobs. Therefore, it is possible to combine the two tasks performed by the map function into one single Mapper class that will produce two key-value pairs for each input line. Moreover, by differentiating the name of Aneka components from the log level names by using an initial underscore character, it will be very easy to post process the output of the reduce function in order to present and organize data.

## 2. Mapper Design and Implementation

The operation performed by the *map* function is a very simple text extraction that identifies the level of the logging and the name of the component entering the information in the log. Once this information is extracted, a key-value pair (string, long) is emitted by the function. Since we decided to combine the two MapReduce jobs into one single job, each *map* task will at most emit two key-value pair. This is because some of the log lines do not record the name of the component that entered the line, and for these lines only, the key-value pair corresponding to the log level will be emitted.

**Listing 10.** Log Parsing Mapper Implementation.

```
using Aneka.MapReduce;
namespace Aneka.MapReduce.Examples.LogParsing
{
    /// <summary>
    /// Class LogParsingMapper. Extends Mapper<K,V> and provides an
    /// implementation of the map function for parsing the Aneka container log files.
    /// This mapper emits a key-value (log-level, 1) and potentially another
```

```
    /// key-value (_aneka-component-name,1) if it is able to extract such information
    /// from the input.
    /// </summary>
    public class LogParsingMapper: Mapper<long,string>
    {
        /// <summary>
        /// Reads the input and extracts the information about the log level and if
        /// found the name of the aneka component that entered the log line.
        /// </summary>
        /// <param name = "input">map input</param>
        protected override void Map(IMapInput<long,string>input)
        {
            // we don't care about the key, because we are only interested on
            // counting the word of each line.
            string value = input.Value;
            long quantity = 1;
            // first we extract the log level name information. Since the date is
            // reported in the standard format DD MMM YYYY mm:hh:ss it is possible
            // to skip the first 20 characters (plus one space) and then extract the
            // next following characters until the next position of the space
               character.
            int start = 21;
            int stop = value.IndexOf(' ', start);
            string key = value.Substring(start, stop - start);
            this.Emit(key, quantity);
            // now we are looking for the Aneka component name that entered the log
            // line if this is inside the log line it is just right after the log
               level preceeded
            // by the character sequence <space><dash><space> and terminated by the
               <colon>
            // character.
            start = stop + 3; // we skip the <space><dash><space> sequence.
            stop = value.IndexOf(':', start);
            key = value.Substring(start, stop - start);
            // we now check whether the key contains any space, if not then it is the
            // name of an Aneka component and the line does not need to be skipped.
            if (key.IndexOf(' ') == -1)
            {
                 this.Emit("_" + key, quantity);
            }
         }
    }
}
```

Listing 10 shows the implementation of the mapper class for the log parsing task. The Map method simply locates the position of the log level label into the line, extracts it and emits a corresponding key-value pair *(label, 1)*. It then tries to locate the position of the name of the Aneka component that entered the log line by looking for a sequence of characters that is limited by a colon. If such a sequence does not contain spaces, it then represents the name of the Aneka component. In this case, another key-value pair *(component-name, 1)* is emitted. As already discussed, in order to differentiate the log level labels from component names, an underscore is pre-fixed to the name of the key in this second case.

## 3. Reducer Design and Implementation

The implementation of the *reduce* function is even more straightforward; the only operation that needs to be performed is to add all the values that are associated to the same key and emit a key-value pair with the total sum. The infrastructure will already aggregate all the values that have been emitted for a given key. Therefore, we simply need to iterate over the collection of values and sum them up.

**Listing 11.** Aneka Log Parsing Reducer Implementation.

```
using Aneka.MapReduce;
namespace Aneka.MapReduce.Examples.LogParsing
{
    /// <summary>
    /// Class <b><i>LogParsingReducer</i></b>. Extends Reducer<K,V> and provides an
    /// implementation of the redcuce function for parsing the Aneka container log
    /// files. The Reduce method iterates all over values of the enumerator and sums
    /// the values before emitting the sum to the output file.
    /// </summary>
    public class LogParsingReducer: Reducer<string,long>
    {
        /// <summary>
        /// Iterates all over the values of the enumerator and sums up
        /// all the values before emitting the sum to the output file.
        /// </summary>
        /// <param name = "input">reduce source</param>
        protected override void Reduce(IReduceInputEnumerator<long>input)
        {
            long sum = 0;
            while(input.MoveNext())
            {
                long value = input.Current;
                sum += value;
            }
            this.Emit(sum);
        }
    }
}
```

As it can be noticed in Listing 11, the operation to perform is very simple and actually the same for both of the two different key-value pairs extracted from the log lines. It will be the responsibility of the driver program to differentiate among the different type of information assembled in the output files.

## 4. Driver Program

The *LogParsingMapper* and *LogParsingReducer* constitute the core functionality of the *MapReduce* job, which only requires to be properly configured in the main program in order to process and produce text tiles. Moreover, since we have designed the mapper component to extract two different types of information, another task that is performed in the driver application is the separation of these two statistics into two different files for further analysis.

**Listing 12.** Driver Program Implementation

```
using System.IO;
using Aneka.Entity;
using Aneka.MapReduce;
namespace Aneka.MapReduce.Examples.LogParsing
{
    /// <summary>
    /// Class Program. Application driver. This class sets up the MapReduce
    /// job and configures it with the <i>LogParsingMapper</i> and
    /// <i>LogParsingReducer</i> classes. It also configures the MapReduce runtime
    /// in order sets the appropriate format for input and output files.
    /// </summary>
    public class Program
    {
        /// <summary>
        /// Reference to the configuration object.
        /// </summary>
        private static Configuration configuration = null;
        /// <summary>
        /// Location of the configuration file.
        /// </summary>
        private static string confPath = "conf.xml";
        /// <summary>
        /// Processes the arguments given to the application and according
        /// to the parameters read runs the application or shows the help.
        /// </summary>
        /// <param name = "args">program arguments</param>
        private static void Main(string[] args)
        {
            try
            {
                Logger.Start();
                // get the configuration Program.configuration = Program.
                   Initialize(confPath);
                // configure MapReduceApplication
                MapReduceApplication<LogParsingMapper, LogParsingReducer>
                application = new MapReduceApplication<LogParsingMapper, LogPars
                ingReducer>("LogParsing", configuration);
```

```
        // invoke and wait for result
        application.InvokeAndWait(newEventHandler<ApplicationEventArgs>
        (OnDone));
        // alternatively we can use the following call
        // application.InvokeAndWait();
    }
    catch(Exception ex)
    {
        Program.ReportError(ex);
    }
    finally
    {
        Logger.Stop();
    }
    Console.Readline();
}
/// <summary>
/// Initializes the configuration and ensures that the appropriate input
/// and output formats are set.
/// </summary>
/// <param name = "configFile">A string containing the path to the config
    file.</param>
/// <returns>An instance of the configuration class.</returns>
private static Configuration Initialize(stringconfigFile)
{
    Configuration conf = Configuration.GetConfiguration(confPath);
    // we ensure that the input and the output formats are simple
    // text files.
    PropertyGroup mapReduce = conf["MapReduce"];
    if (mapReduce == null)
    {
        mapReduce = newPropertyGroup("MapReduce");
        conf.Add("MapReduce") = mapReduce;
    }
    // handling input properties
    PropertyGroup group = mapReduce.GetGroup("Input");
    if (group == null)
    {
        group = newPropertyGroup("Input");
        mapReduce.Add(group);
    }
    string val = group["Format"];
    if (string.IsNullOrEmpty(val) == true)
```

```
    {
        group.Add("Format","text");
    }
    val = group["Filter"];
    if (string.IsNullOrEmpty(val) == true)
    {
        group.Add("Filter","*.log");
    }
    // handling output properties
    group = mapReduce.GetGroup("Output");
    if (group == null)
    {
        group = newPropertyGroup("Output");
        mapReduce.Add(group);
    }
    val = group["Format"];
    if (string.IsNullOrEmpty(val) == true)
    {
        group.Add("Format","text");
    }
    return conf;
}
/// <summary>
/// Hooks the ApplicationFinished events and process the results
/// if the application has been successful.
/// </summary>
/// <param name = "sender">event source</param>
/// <param name = "e">event information</param>
private static void OnDone(object sender, ApplicationEventArgs e)
 {
    if (e.Exception != null)
    {
        Program.ReportError(ex);
    }
    else
    {
        Console.Write("Aneka Log Parsing-Job Terminated: SUCCESS");
        FileStream logLevelStats = null;
        FileStream componentStats = null;
        string workspace = Program.configuration.Workspace;
        string outputDir = Path.Combine(workspace, "output");
        DirectoryInfo sources = new DirectoryInfo(outputDir);
```

```
FileInfo[] results = sources.GetFiles();
try
{
    logLevelStats = new FileStream(Path.Combine(workspace,
    "loglevels.txt"), FileMode.Create, FileAccess.Write));
    componentStats = new FileStream(Path.Combine(workspace,
    "components.txt"), FileMode.Create, FileAccess.Write));
    using(StreamWriter logWriter = new StreamWriter(logLevelStats))
    {
        using(StreamWritercompWriter = newStreamWriter(component
        Stats))
        {
            foreach(FileInfo result in results)
            {
                using(StreamReader reader =
                     new StreamReader(result.OpenRead()))
                {
                    while(reader.EndOfStream == false)
                    {
                        string line = reader.ReadLine();
                        if (line != null)
                        {
                            if (line.StartsWith("_") == true)
                            {
                                compWriter.WriteLine(line.
                                Substring(1));
                            }
                            else
                            {
                                logWriter.WriteLine(line);
                            }
                        }
                    }
                }
            }
        }
    }
    // clear the output directory
    sources.Delete(true);
    Console.WriteLine("Statistics saved to:[loglevels.txt,
    components.txt]");
    Environment.ExitCode = 0;
}
catch(Exception ex)
```

```
                {
                    Program.ReportError(ex);
                }
                Console.WriteLine("<Press Return>");
            }
        }
        /// <summary>
        /// Displays a simple informative message explaining the usage of the
        /// application.
        /// </summary>
        private static void Usage()
        {
            Console.WriteLine("Aneka Log Parsing - Usage Log.Parsing.Demo.Console.
            exe" + "[conf.xml]");
        }
        /// <summary>
        /// Dumps the error to the console, sets the exit code of the application
            to -1
        /// and saves the error dump into a file.
        /// </summary>
        /// <param name="ex">runtime exception</param>
        private static void ReportError(Exception ex)
        {
            IOUtil.DumpErrorReport(Console.Out, ex, "Aneka Log Parsing-Job
            Terminated: " + "ERROR");
            IOUtil.DumpErrorReport(ex, "Aneka Log Parsing-Job Terminated: ERROR");
            Program.Usage();
            Environment.ExitCode = -1;
        }
    }
}
```

Listing 12 shows the implementation of the driver program. With respect to the previous examples, there are three things to be noticed:

- The configurarion of the MapReduce job
- The post processing of the result files
- The management of errors.

The configuration of the job is performed in the *Initialize* method. This method reads the configuration file from the local file system and ensures that the input and output formats of files are set to *text*. *MapReduce* jobs can be configured by using a specific section of the configuration file named *MapReduce*. Within this section, two sub-sections control the properties of input and output files, and are named Input and Output respectively. The input and output sections may contain the following properties:

*Format (string).* Defines the format of the input file. If this property is set, the only supported value is text.

*Filter (string).* Defines the search pattern to be used to filter the input files to process in the workspace directory. This property only applies for the Output properties group.

*NewLine (string).* Defines the sequence of characters that is used to detect (or write) a new line in the text stream. This value is meaningful when the input/output format is set to text, and the default value is selected from the execution environment if not set.

*Separator (character).* This property is only present in the Output section and defines the character that needs to be used to separate the key from the value in the output file. As the previous property, this value is meaningful when the input/output format is set to text.

Besides the specific setting for input and output files, it is possible to control other parameters of a *MapReduce* job. These parameters are defined in the main *MapReduce* configuration section and their meaning has been already discussed in Section 8.3.1.

Instead of a programming approach for the initialization of the configuration, it is also possible to embed these settings into the standard Aneka configuration file as demonstrated in the following listing.

**Listing 13.** Driver Program Configuration File (conf.xml)

```
<?xml version = "1.0" encoding = "utf-8" ?>
<Aneka>
  <UseFileTransfervalue = "false" />
  <Workspacevalue = "Workspace" />
  <SingleSubmissionvalue = "AUTO" />
  <PollingTimevalue = "1000"/>
  <LogMessagesvalue = "false" />
  <SchedulerUrivalue = "tcp://localhost:9090/Aneka" />
  <UserCredential type = "Aneka.Security.UserCredentials" assembly = "Aneka.dll">
    <UserCredentials username="Administrator" password=""/>
  </UserCredentials>
  <Groups>
    <Group name = "MapReduce">
      <Groups>
        <Group name = "Input">
        <Property name = "Format" value = "text" />
        <Property name = "Filter" value="*.log" />
      </Group>
      <Group name = "Output">
        <Property name = "Format" value = "text" />
      </Group>
    </Groups>
    <Property name = "LogFile" value = "Execution.log"/>
    <Property name = "FetchResult" value = "true" />
    <Property name = "UseCombiner" value = "true" />
    <Property name = "SynchReduce" value = "false" />
    <Property name = "Partitions" value = "1" />
```

```
    <Property name = "Attempts" value = "3" />
  </Group>
    </Groups>
</Aneka>
```

As demonstrated, it is possible to open a <Group name="MapReduce">…</Group/> tag and enter all the properties that are required for the execution. The Aneka configuration file is based on a flexible framework that allows to simply enter groups of name-value properties. The *Aneka.Property* and *Aneka.PropertyGroup* classes also provide facilities for converting the strings representing the value of a property into a corresponding built-in type if possible. This simplifies the task of reading from and writing to configuration objects.

The second element shown in Listing 12 is represented by the post processing of the output files. This operation is performed in the *OnDone* method, whose invocation is triggered either if an error occurs during the execution of the MapReduce job or if it completes successfully. This method separates the occurrences of log level labels from the occurrences of Aneka component names by saving them into two different files: *loglevels.txt* and *components.txt* under the workspace directory, and then deletes the output directory where the output files of the reduce phase have been downloaded. These two files contain the aggregated result of the analysis and can be used to extract statistic information about the content of the log files, and display in a graphical manner as shown in the next section.

A final aspect that can be considered is the management of errors. Aneka provides a collection of APIs that are contained in the *Aneka.Util* library and represent utility classes for automating tedious tasks such as the appropriate collection of stack trace information associated to an exception or the information about the types of the exception thrown. In our example, the reporting features, which are triggered in case of exceptions, are implemented into the *ReportError* method. This method utilizes the facilities offered by the *IOUtil* class in order to dump a simple error report to both the console and a log file that is named using the following pattern: *error.YYYY-MM-DD_hh-mm-ss.log*.

# 5. Running the Application

Aneka produces a considerable amount of logging information. The default configuration of the logging infrastructure creates a new log file for each activation of the Container process or as soon as the dimension of the log file goes beyond 10 MB. Therefore, by simply keeping an Aneka Cloud running for a few days, it is quite easy to collect enough data to mine for our sample application. Moreover, this scenario also constitutes a real case study for MapReduce since one of its most common practical applications is extracting semi-structured information from logs and traces of execution.

In the execution of the test, we used a distributed infrastructure consisting of 7 worker nodes and one master node interconnected through a LAN. We processed 18 log files of several sizes for a total aggregate size of 122 MB. The execution of the MapReduce job over the collected data produced the results, which are stored in the *loglevels.txt* and *components.txt* files, and are represented graphically in Fig. 8.12 and Fig. 8.13 respectively.

The two graphs show that there is a considerable amount of unstructured information in the log files produced by the Container processes. In particular, about 60% of the log content is skipped during the classification. This content is more likely due to the result of stack trace dumps into the log file, which produces—as a result of ERROR and WARN entries—a sequence of lines that are not recognized. Figure 8.13 shows the distribution among the components that make use of the logging APIs. This distribution is computed over the data recognized as a valid log entry and the graph shows that just about 20% of these entries have not been recognized by the parser implemented in the map function. It can be then inferred that the meaningful information extracted from the log analysis constitutes about 32% (80% of 40% of the total lines parsed) of the entire log data.

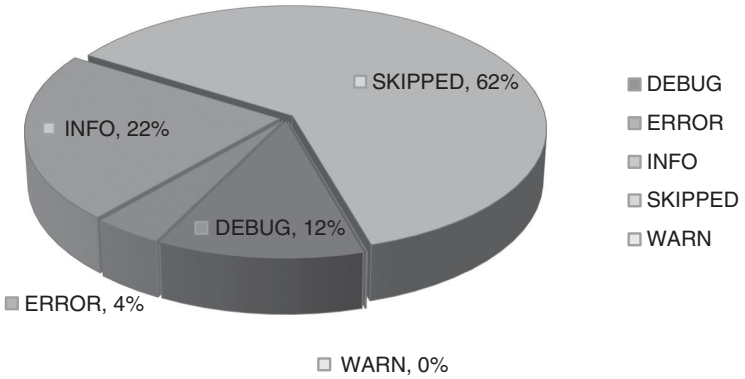


**Fig. 8.12.** Log Levels Entries Distribution.

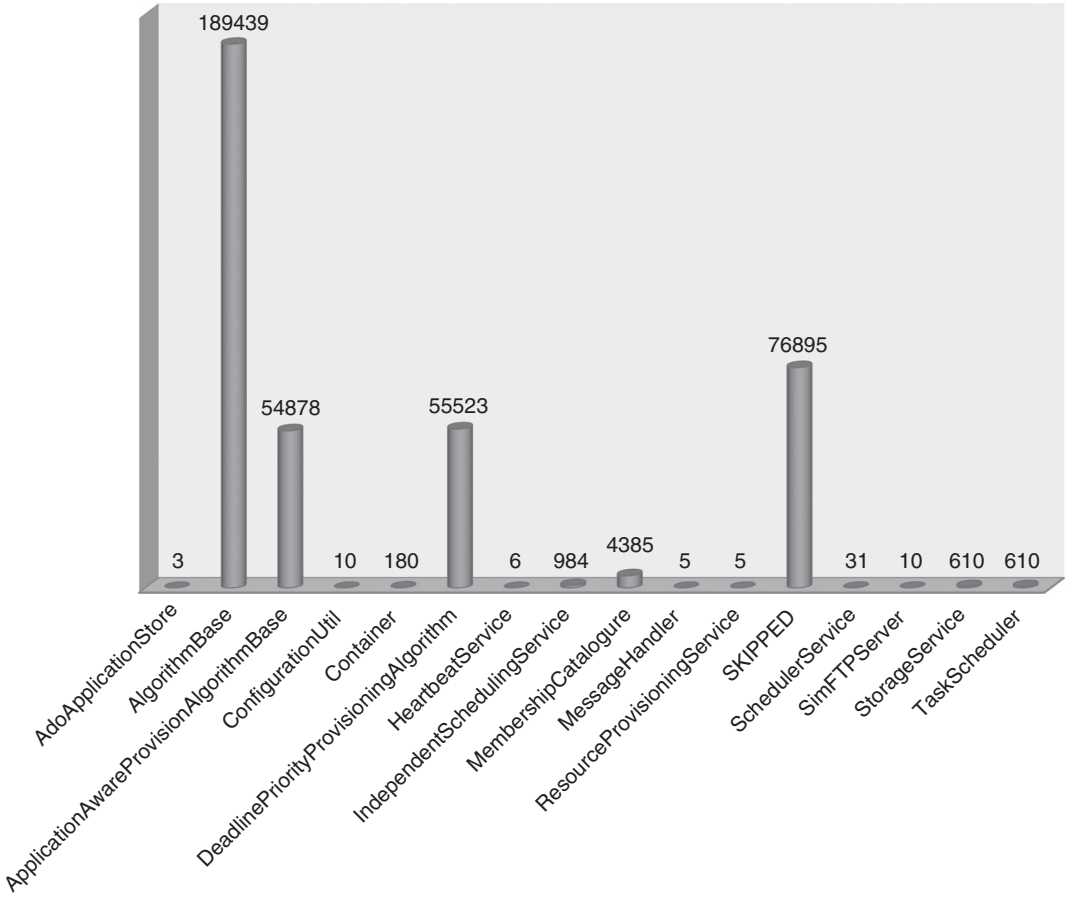


**Fig. 8.13.** Components Entries Distribution.

Despite the simplicity of the parsing function implemented in the *map* task, this practical example shows how the Aneka MapReduce programming model can be used to easily perform massive data analysis

tasks. The purpose of the case study was not to create a very refined parsing function but to demonstrate how to logically and programmatically approach a realistic data analysis case study with MapReduce and how to implement it on top of the Aneka APIs.

# Summary

In this chapter, we introduced the main characteristics of *data-intensive computing*. Data-intensive applications process or produce high volumes of data and may also exhibit compute intensive properties. The size of data that has triggered the definition of "data-intensive computation" has changed over time and, together with it, also the technologies and the programming and storage models used for data-intensive computing. Data-intensive computing is a field originally prominent in high-speed, wide area network applications. It is now the domain of Storage Clouds, where the dimension of data reaches the size of terabytes, if not petabytes, and it is referred as *Big Data*. This term identifies the massive amount of information that is produced, processed, and mined not only by scientific applications, but also by companies providing Internet services such as search, on-line advertisement, social media, and social networking.

One of the interesting characteristics of *Big Data* world is that the data is represented in a semi-structured or unstructured form. Therefore, traditional approaches based on relational databases, are not capable of efficiently supporting data-intensive applications. New approaches and storage models have been investigated to address these challenges. In the context of storage systems, the most significant efforts have been directed towards the implementation of high-performance distributed file systems, storage Clouds, and *NoSQL*-based systems. For the support of programming data-intensive applications, the most relevant innovation has been the introduction of *MapReduce* together with all its variations aiming at extending the applicability of the proposed approach to a wider range of scenarios.

*MapReduce* has been proposed by Google and provides a simple approach for processing large quantities of data based on the definition of two functions—*map* and *reduce*—that are applied to the data in a two phase process. First, the map phase extracts the valuable information from the data and stores it into key-value pairs, which are eventually aggregated together in the reduce phase. Such a model, even though constraining, has proven to be successful in several application scenarios.

We discussed the reference model of MapReduce as proposed by Google and provided pointers to relevant variations. We described the implementation of MapReduce in Aneka. Similar to Thread and Task programming models in Aneka, we discussed the programming abstractions supporting the design and the implementation of MapReduce applications. We presented the structure and the organization of runtime services of Aneka for the execution of MapReduce jobs. We included step-by-step examples on how to design and implement applications using Aneka MapReduce APIs.

# Review Questions

1. What is *Data-Intensive Computing*? Describe the characteristics that define this term.
2. Provide an historical perspective of the most important technologies supporting data-intensive computing.
3. What are the characterizing features of the so called *Big Data*?

4. List some of the important storage technologies supporting Data-Intensive Computing and describe one of them.
5. Describe the architecture of the Google File System.
6. What does the term “NoSQL” mean?
7. Describe the characteristics of Amazon Simple Storage Service (S3).
8. What is Google *Bigtable*?
9. What are the requirements of a programming platform supporting data-intensive computations?
10. What is MapReduce?
11. Describe what kinds of problems MapReduce can solve and give some real examples.
12. List some of the variations or extensions to MapReduce.
13. What are the major components of the Aneka *MapReduce Programming Model*?
14. How does the MapReduce model differ from the other models supported by Aneka and discussed in this book?
15. Describe the components of the Scheduling and Execution services that constitute the runtime infrastructure supporting MapReduce.
16. Describe the architecture of the data storage layer designed for Aneka MapReduce and the I/O APIs for handling MapReduce files.
17. Design and implement a simple program that uses MapReduce for the computation of Pi.

# 9 Cloud Platforms in Industry

Cloud computing allows end users and developers to leverage large distributed computing infrastructures. This is made possible thanks to infrastructure management software and distributed computing platforms offering on demand compute, storage, and, on top of these, more advanced services. There are several different options for building enterprise Cloud computing applications or for using Cloud computing technologies to integrate and extend existing industrial applications. An overview of a few prominent Cloud computing platforms and a brief description of the type of service offered by them are shown in Table 9.1. A Cloud computing system can be developed by using either a single technology and vendor or a combination of them.

This chapter presents some of the representative Cloud computing solutions offered as Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) services in market. It provides some insights and practical issues around the architecture of the major Cloud computing technologies and their service offerings.

## 9.1 AMAZON WEB SERVICES

Amazon Web Services (AWS) is a platform allowing the development of flexible applications by providing solutions for elastic infrastructure scalability, messaging, and file and data storage. The platform is accessible through SOAP or RESTful Web service interfaces and provides a Web based console where users can administrate and monitor the resources required as well as their expenses computed on a pay as you go basis.

**Table 9.1.** Some Examples of Cloud-Computing Offerings.

| *Vendor / Product* | *Service Type* | *Description* |
|---|---|---|
| *Amazon Web Services* | IaaS, PaaS, SaaS | Amazon Web Services (AWS) is a collection of Web services providing developers with compute, storage, and more advanced services. AWS is mostly popular for IaaS services and primarily for its elastic compute service EC2. |

(*Continued*)

**Table 9.1.** Continued.

| *Vendor / Product* | *Service Type* | *Description* |
|---|---|---|
| *Google AppEngine* | PaaS | Google AppEngine is a distributed and scalable runtime for developing scalable Web applications based on Java and Python runtime environments. These are enriched with access to services that simplify the development of applications in a scalable manner. |
| *Microsoft Azure* | PaaS | Microsoft Azure is a Cloud operating system providing services for developing scalable applications based on the proprietary Hyper-V virtualization technology and the .NET framework. |
| *SalesForce.com and Force.com* | SaaS, PaaS | SalesForce.com is a Software-as-a-Service solution allowing prototyping CRM applications. It leverages the Force.com platform that is made available for developing new component and capabilities for CRM applications. |
| *Heroku* | PaaS | Heroku is a scalable run-time environment for building applications based on Ruby. |
| *RightScale* | IaaS | Rightscale is Cloud management platform, which provides facilities for building a scalable computing infrastructure. It provides templates that facilitate the deployment of Web applications in the Cloud. |

Figure 9.1 shows all the services available in the AWS ecosystem. At the basis of the solution stack, there are services that provide raw compute and raw storage: *Amazon Elastic Compute* (*EC2*) and *Amazon Simple Storage Service* (*S3*). These are the two most popular services, which are generally complemented with other offerings for building a complete system. At the higher level, *Elastic MapReduce* and *AutoScaling* provide additional capabilities for building smarter and elastic computing system. On the data side, *Elastic Block Store* (*EBS*), *Amazon SimpleDB*, *Amazon RDS*, and *Amazon ElastiCache* provide solutions for reliable data snapshots and the management of structured and semi-structured data. Communication needs are covered at the networking level by *Amazon Virtual Private Cloud* (*VPC*), *Elastic Load Balancing*, *Amazon Route 53*, and *Amazon Direct Connect*. More advanced services for connecting applications are *Amazon Simple Queue Service* (*SQS*), *Amazon Simple Notification Service* (*SNS*), and *Amazon Simple E-mail Service* (*SES*). Other services include

- *Amazon CloudFront*: content delivery network solution
- *Amazon CloudWatch*: monitoring solution for several Amazon services
- *Amazon Elastic BeanStalk* and *CloudFormation*: flexible application packaging and deployment

As shown, AWS comprise a wide set of services. We discuss the most important services by discussing the solutions proposed by AWS for what regards compute, storage, communication, and complementary services.

## 9.1.1 Compute Services

Compute services constitute the fundamental element for Cloud computing systems. The fundamental service in this space is Amazon EC2, which delivers an Infrastructure-as-a-Service solution, and it has represented a reference model for several offerings from other vendors in the same market segment. Amazon EC2 allows deploying servers in the form of virtual machines created as instances of

a specific image. Images come with already preinstalled operating system and a software stack, and instances can be configured for what regards memory, number of processors, and storage. Users are provided with credentials to remotely access the instance and further configure or install software, if needed.

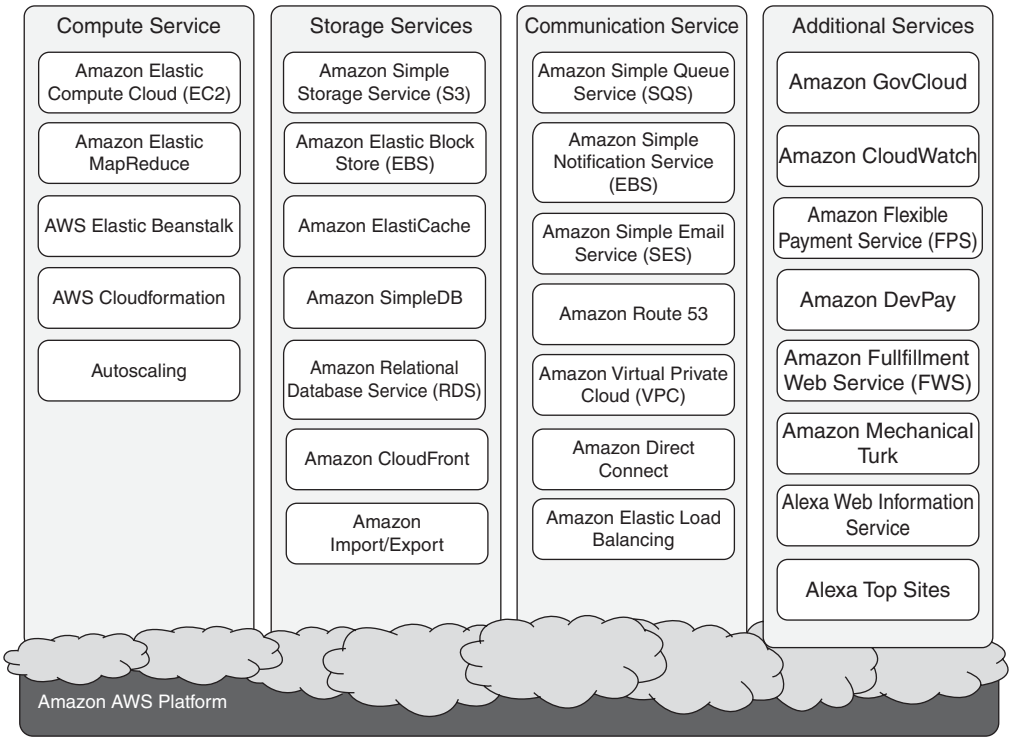


**Fig. 9.1.** Amazon Web Services Ecosystem.

## 1. Amazon Machine Image (AMI)

AMIs are templates from which it is possible to create a virtual machine. They are stored into the Amazon S3 and identified by a unique identifier in the form of *ami-xxxxxx* and a manifest XML file. An AMI contains a physical file system layout with a predefined operating system installed. These two are specified by the *Amazon Ramdisk Image* (*ARI*, id: *ari-yyyyyy*) and the *Amazon Kernel Image* (*AKI*, id: *aki-zzzzzz*), which are part of the configuration of the template. AMIs are either created from scratch or "bundled" from existing EC2 instances running. A common practice is to prepare new AMIs to create an instance from a pre-existing AMI, log into it once it is booted and running, and install all the software needed and, by using the tools provided by Amazon, the instance is converted into a new image. Once an AMI is created, it is stored into an S3 bucket and the user can decide whether to make it available to other users or keep it for personal use. Finally, it is also possible to associate a product code with a given AMI, thus allowing the owner of the AMI to get revenue every time this AMI is used to create EC2 instances.

## 2. EC2 Instance

EC2 instances represent virtual machines. They are created by using AMI as templates, which are specialized by selecting the number of cores, their computing power, and the installed memory. The

processing power is expressed in terms of virtual cores and EC2 Compute Units (ECUs). The ECU is a measure of the computing power of a virtual core and it is used to express a predictable quantity of real CPU power that can is allocated to an instance. By using compute units instead of real frequency values, Amazon can change over time the mapping of such units to the underlying real amount of computing power allocated, thus keeping the performance of EC2 instances consistent with standard set by the times. Over time, the hardware supporting the underlying infrastructure will be replaced by more powerful ones and the use of ECUs helps to give users a consistent view of the performance offered by EC2 instances. Since users rent computing capacity rather than buying hardware, this approach is reasonable. At the time of writing, 1 ECU is defined as giving the same performance of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor[54].

Table 9.2 shows all the currently available configurations for EC2 instances. We can identify six major categories:

**(a) Standard Instances.** This class offers a set of configurations that are suitable for most applications. EC2 provides three different categories of increasing computing power, storage, and memory.

**(b) Micro Instances.** This class is suitable for those applications that consume a limited amount of computing power and memory, and occasionally need burst in CPU cycles to process surges in the workload. Micro instances can be used for small Web applications with limited traffic.

**(c) High-Memory Instances.** This class targets applications that need to process huge workloads and require large amount of memory. Three-tier Web applications characterized by high traffic are the target profile. Three categories are available of increasing memory and CPU, with memory proportionally larger than computing power.

**(d) High-CPU Instances.** This class targets compute-intensive applications. Two configurations are available where computing power proportionally increases more than memory.

**(e) Cluster Compute Instances.** This class is used to provide virtual cluster services. Instances in this category are characterized by high CPU compute power, large memory, and an extremely high I/O and network performance, which makes it suitable for HPC applications.

**(f) Cluster GPU Instances.** This class provides instances featuring graphic processing units (GPUs) and high compute power, large memory, and extremely high I/O and network performance. This class is particularly suited for cluster applications that perform heavy graphic computations, such as rendering clusters. Since GPU can be used for general purpose computing, users of such instances can benefit from additional computing power, which makes them suitable for HPC applications.

EC2 instances are priced hourly according to the category they belong to. At the beginning of every hour of usage the user will be charged with the cost of the entire hour. The hourly expense charged for one instance is constant. Instance owners are responsible to provide their own backup strategies, since there is no guarantee that the instance will run for the entire hour. Another alternative is represented by *Spot Instances*. These instances are much more dynamic in terms of pricing and lifetime, since they are made available to the user according to the load of EC2 and the availability of resources. Users define an upper bound for a price they want to pay these instances; as long as the current price (spot price) remains under the given bound, the instance is kept running. The price is sampled at the beginning of each hour. Spot instances are more volatile than normal instances: whereas for normal instances, EC2 will try as much as possible to keep them active, there is no such guarantee for spot instances. Therefore, the need of implementing backup and checkpointing strategies is more important.

[54] Reference: ***http://aws.amazon.com/ec2/faqs/#What_is_an_EC2_Compute_Unit_and_why_did_you_introduce_it***

EC2 instances can be run either by using the command line tools provided by Amazon, which connects the Amazon Web service providing remote access to the EC2 infrastructure, or by the AWS console, which allows the management of other services, such as S3. By default, an EC2 instance is created with the kernel and the disk associated to the AMI. These define the architecture (32 bit or 64 bit) and the space of disk available to the instance. This is an ephemeral disk: once the instance is shutdown, the content of the disk will be lost. Alternatively, it is possible to attach an EBS volume to the instance, whose content will be stored in S3. If the default AKI and ARI are not suitable, EC2 provides capabilities to run EC2 instance by specifying a different AKI and ARI, thus giving flexibility in the creation of instances.

**Table 9.2.** Amazon EC2 (On-Demand) Instances Characteristics.

| *Instance Type* | *ECU* | *Platform* | *Memory* | *Disk Storage* | *Price (US East)* | *(USD/hour)* |
|---|---|---|---|---|---|---|
| *Standard Instances* | | | | | | |
| *Small* | 1(1x1) | 32 bit | 1.7 GB | 160 GB | $0.085 Linux | $0.12 Windows |
| *Large* | 4(2x2) | 64 bit | 7.5 GB | 850 GB | $0.340 Linux | $0.48 Windows |
| *Extra Large* | 8(4x2) | 64 bit | 15 GB | 1690 GB | $0.680 Linux | $0.96 Windows |
| *Micro Instances* | | | | | | |
| *Micro* | <=2 | 32/64 bit | 613 MB | EBS Only | $0.020 Linux | $0.03 Windows |
| *High Memory Instances* | | | | | | |
| *Extra Large* | | | | | | |
| *Double Extra Large* | 6.5 (2x3.25) | 64 bit | 17.1 GB | 420 GB | $0.500 Linux | $0.62 Windows |
| *Quadruple Extra Large* | 13 (4x3.25) | 64 bit | 34.2 GB | 850 GB | $1.000 Linux | $1.24 Windows |
| | 26 (8x3.25) | 64 bit | 68.4 GB | 1690 GB | $2.000 Linux | $2.48 Windows |
| *High CPU Instances* | | | | | | |
| *Medium* | 5 (2x2.5) | 32 bit | 1.7 GB | 350 GB | $0.170 Linux | $0.29 Windows |
| *Extra Large* | 20 (8x2.5) | 64 bit | 7 GB | 1690 GB | $0.680 Linux | $1.16 Windows |
| *Cluster Instances* | | | | | | |
| *Quadruple Extra Large* | 33.5 | 64 bit | 23 GB | 1690 GB | $1.600 Linux | $1.98 Windows |
| *Cluster GPU Instances* | | | | | | |
| *Quadruple Extra Large* | 33.5 | 64 bit | 22 GB | 1690 GB | $2.100 Linux | $2.60 Windows |

## 3. EC2 Environment

EC2 instances are executed within a virtual environment, which provides them with the services they require to host applications. The EC2 environment is in charge of allocating addresses, attaching storage volumes, and configuring the security in terms of access control and network connectivity.

By default, instances are created with an internal IP address, which makes them capable to communicate within the EC2 network and access the internet as clients. It is possible to associate to them an *Elastic IP*, which can be remapped to a different instance over time. Elastic IPs allows instances running in EC2 to act as servers reachable from the Internet, and since they are not strictly bound to specific instance, to implement failover capabilities. Together with an external IP, EC2 instances are also given a domain name that generally is in the form *ec2-xxx-xxx-xxx.compute-x.amazonaws.com*, where *xxx-xxx-xxx* normally represents the four parts of the external IP address separated by a dash and *compute-x* gives information about the availability zone where instances

are deployed. At the time of writing, there are 5 availability zones that are priced differently: 2 in the United States (Virginia and Northern California), 1 in Europe (Ireland), and 2 in Asia Pacific (Singapore and Tokyo).

Instance owners can partially control where to deploy instances. Instead, they have a finer control on the security of the instances as well as their network accessibility. Instance owners can associate a key-pair to one or more instances when these instances are created. A key pair allows the owner to remotely connect to the instance once this is running and gain root access to it. Amazon EC2 controls the accessibility of a virtual instance with basic firewall configuration allowing the specification of source address, port, and protocols (TCP, UDP, ICMP). Rules can also be attached to security groups and instances can be made part of one or more groups before their deployment. Security groups and firewall rules constitute a flexible way of providing basic security for EC2 instances, which has to be complemented by appropriate security configuration within the instance itself.

## 4. Advanced Compute Services

EC2 instances and AMIs constitute the basic blocks for building an Infrastructure-as-a-Service computing Cloud. On top of these, Amazon Web Services provide more sophisticated services that allow the easy packaging of application and their deployment, and a computing platform supporting the execution of MapReduce-based applications.

**(a) AWS CloudFormation.** AWS CloudFormation constitutes an extension of the simple deployment model that characterizes EC2 instances. CloudFormation introduces the concepts of templates, which are JSON formatted text files describing the resources needed to run an application or a service in EC2 together with the relations between them. CloudFormation allows easily and explicitly linking EC2 instances together and introducing dependencies among them. Templates provide a simple and declarative way to build complex systems and integrate EC2 instances with other AWS services such as S3, SimpleDB, SQS, SNS, Route 53, Elastic Beanstalk and others.

**(b) AWS Elastic Beanstalk.** AWS Elastic Beanstalk constitutes a simple and easy way to package applications and deploy them on the AWS Cloud. This service simplifies the process of provisioning instances and deploying application code, and providing appropriate access to them. Currently, this service is available only for Web applications developed with the Java/Tomcat technology stack: developers can conveniently package their Web application into a WAR file and use Beanstalk to automate its deployment on the AWS Cloud.

With respect to other solutions that automate the Cloud deployment, Beanstalk simplifies tedious tasks without negating to the user the capability of accessing—and taking over the control of—the underlying EC2 instances that make up the virtual infrastructure on top of which the application is running. With respect to AWS CloudFormation, AWS Elastic Beanstalk provides a higher-level approach for application deployment on the Cloud, which does not require the user to specify the infrastructure in terms of EC2 instances and their dependencies.

**(c) Amazon Elastic MapReduce.** Amazon Elastic MapReduce provides AWS users with a Cloud computing platform for MapReduce applications. It utilizes Hadoop as MapReduce engine, deployed on to a virtual infrastructure composed by EC2 instances and uses Amazon S3 for storage needs.

Besides supporting all the application stack connected to Hadoop (Pig, Hive, etc.), it introduces elasticity and allows users to dynamically size the Hadoop cluster according to their needs, as well as selecting the appropriate configuration of EC2 instances to compose the cluster (small, high-memory, high-CPU, cluster compute, and cluster GPU). On top of these services, basic Web applications allowing users to quickly run data intensive application without writing code, are offered.

## 9.1.2 Storage Services

AWS provides a collection of services for data storage and information management. The core service in this area is represented by Amazon *Simple Storage Service (S3)*. This is a distributed object store that allows users to store information in different. The core components of S3 are two: *buckets* and *objects*. Buckets represent virtual containers where to store objects, while objects represent the content that is actually stored. Objects can also be enriched with metadata that can be used to tag the content stored with additional information.

### *1. S3 Key Concepts*

As the name suggest, S3 has been designed to provide a simple storage service accessible through a REST interface, which is quite similar to a distribute file system but that presents some important differences that allow the infrastructure to be highly efficient:

*The storage is organized in a two level hierarchy.* S3 organizes its storage space into buckets that cannot be further partitioned. This means that it is not possible to create directories or other kinds of physical groupings for objects stored in a bucket. Despite this, there are less limitations in naming objects, and this allows users to simulate directories and create logical groupings.

*Objects stored cannot be manipulated like standard files.* S3 has been designed to essentially provide storage for objects that will not change over time. Therefore, it does not allow renaming, modifying, or relocating an object. Once an object has been added to a bucket, its content and position is immutable, and the only way to change one of them is to remove the object from the store and add it again.

*Content is not immediately available to users.* The main design goal of S3 is to provide an eventually consistent data store. As a result, being a large distributed storage facility, changes are not immediately reflected. For instance, S3 uses replication to provide redundancy and efficiently serve objects across the globe; this introduces latencies when adding objects to the store—especially large ones—which are not available instantly across the entire globe.

*Request will occasionally fail.* Due to the large distribute infrastructure being managed, requests for object may occasionally fail. Under certain conditions, S3 can decide to drop a request by returning an internal server error. Therefore, it is expected to have a small failure rate during day-to-day operations, which generally it does not identify a persistent failure.

Access to S3 is provided with RESTful Web services. These express all the operations that can be performed against the storage in the form of HTTP requests (GET, PUT, DELETE, HEAD, and POST), which operate differently according to the element they address. As a rule of thumb, PUT/POST requests add new content to the store, GET/HEAD requests are used to retrieve content and information, while DELETE requests are used to remove elements or information attached to it.

**(a) Resource Naming.** Buckets, objects, and attached metadata are made accessible through a REST interface. Therefore, they are represented by *Uniform Resource Identifiers (URIs)* under the *s3.amazonaws.com* domain. All the operations are then performed by expressing the entity they are directed to in the form of a request for a URI.

Amazon offers three different ways of addressing a bucket:

*Canonical Form: http://s3.amazonaws.com/bukect_name/.* The bucket name is expressed as a path component of the domain name s3.amazonaws.com. This is the naming convention that has less restriction in terms of allowed characters, since all the characters that are allowed for a path component can be used.

*Subdomain form: http://bucketname.s3.amazon.com/.* Alternatively, it is also possible to reference a bucket as a subdomain of s3.amazonaws.com. In order to express a bucket name into this form, the name has to:

- be between 3 to 63 characters long;
- contain only letters, numbers, periods, and dashes;
- start with a letter or a number;
- contain at least one letter; and
- have no fragments between periods that start with a dash, end with a dash, or are an empty string.

This form is equivalent to the previous one when it can be used but it is the one to be preferred since it works more effectively for all the geographical locations serving resources stored in S3.

*Virtual hosting form: http://bucket-name.com/.* Amazon also allows referencing its resources with custom URLs. This is accomplished by entering a CNAME record into the DNS that points to the subdomain form of the bucket URI.

Since S3 is logically organized as flat data store, all the buckets are managed under the s3.amazonaws.com domain. Therefore, the names of buckets must be unique across all the users.

Objects are always referred as resources local to a given bucket. Therefore, they always appear as part of the resource component of a URI. Since bucket can be expressed in three different ways, objects indirectly inherit this flexibility:

- *Canonical form: http://s3.amazonaws.com/bukect_name/object_name.*
- *Subdomain form: http://bucket-name/s3.amzonaws.com/object_name.*
- *Virtual hosting form: http://bucket-name.com/object_name.*

Except for the '?', which separates the resource path of a URI from the set of parameters passed with the request, all the characters that follow the '/' after the bucket reference constitute the name of the object. For instance, path separator characters expressed as part of the object name do not have corresponding physical layout within the bucket store. Despite this, these can still be used to create logical groupings that look like directories.

Finally, specific information about a given object, such as the access control policy of an object or the server logging settings defined for a bucket, can be referenced by using specific parameters. More precisely:

- *Object ACL: http://s3.amazonaws.com/bukect_name/object_name?acl.*
- *Bucket Server Logging: http://s3.amzonaws.com/bucket_name?logging.*

Object metadata are not directly accessible through a specific URI but they are manipulated by adding attributes in the request of the URL and are not part of the identifier.

**(b) Buckets.** A bucket is a container of objects. It can be thought as a virtual drive hosted on the S3 distributed storage, which provides users with a flat store where to add objects. Buckets are top-level elements of the S3 storage architecture and do not support nesting. This means that is not possible to create "sub-buckets" or other kinds of physical divisions.

A bucket is located into a specific geographic location and eventually replicated for fault tolerance and better content distribution. Users can select the location where to create the bucket, which by default are created in US datacenters. Once a bucket is created, all the objects that belong to the bucket will be stored in the same availability zone of the bucket. Users create a bucket by sending a PUT request to *http://s3.amazonaws.com/* with the name of the bucket, and if they want to specify the availability zone, additional information about the preferred location. The content of a bucket can be listed by sending a GET request by specifying the name of the bucket. Once created, the bucket cannot be renamed or re-located.

If it is necessary to do so, the bucket needs to be deleted and recreated. The deletion of a bucket is performed by a DELETE request, which turns to be successful if and only if the bucket is empty.

**(c) Objects and Metadata.** Objects constitute the content elements stored in S3. Users either store files or push to s3 text stream representing the object's content. An object is identified by a name that needs to be unique within the bucket in which the content is stored. The name cannot be longer than 1024 bytes when encoded in UTF-8 and it allows almost any character. Since buckets do not support nesting, even characters normally used as path separators are allowed. This actually compensates the lack of a structured file system, since directories can be emulated by properly naming objects.

Users create objects via a PUT request that specifies the name of the object together with the bucket name, its content, and additional properties. The maximum size of an object is 5 GB. Once an object is created it cannot be modified, renamed, or moved into another bucket. In order to retrieve an object, it is possible to retrieve it via a GET request, while its deletion is performed via a DELETE request.

Objects can be tagged with metadata, which are passed as properties of the PUT request. Such properties are retrieved either with a GET request or with a HEAD request, which only returns the object's metadata without the content. Metadata are both system- and user-defined: the first ones are used by S3 to control the interaction with the object, while the second ones are meaningful to the user, who can store up to 2KB per metadata property represented by a key-value pair of strings.

**(d) Access Control and Security.** Amazon S3 allows controlling the access to buckets and objects by means of *Access Control Policies (ACPs)*. An ACP is a set of *grant permissions* that are attached to a resource expressed by means of an XML configuration file. A policy allows defining up to 100 access rules, each of them granting one of the available permissions to a grantee. Currently, five different permissions can be used:

- *READ:* allows the grantee to retrieve an object and its metadata, and to list the content of a bucket as well as getting its metadata.
- *WRITE:* allows the grantee to add an object to a bucket as well as modify and remove it.
- *READ_ACP:* allows the grantee to read the ACP of a resource.
- *WRITE_ACP:* allows the grantee to modify the ACP of a resource.
- *FULL_CONTROL:* all of the above.

Grantees can be either single users or groups. Users can be identified by their canonical id or the email provided while signing up for S3. In case of groups, only three options are available: all users, authenticated users, and log delivery users[55].

Once a resource is created, S3 attaches a default ACP granting full control permissions to its owner only. Changes to the ACP can be made by using the request to the resource URI followed by "*?acl*". A GET method allows retrieving the ACP, while a PUT method allows uploading a new ACP replacing the existing one. Alternatively, it is possible to set the ACP at the time of creating a resource by using a predefined set of permissions called *canned policies*. These policies represent the most common access patterns for S3 resources.

ACPs provide a set of powerful rules to control the access to resources by S3 users, while they do not exhibit fine grain in case of non-authenticated users, who cannot be differentiated and are considered as a group. In order to provide a finer grain in this scenario, S3 allows defining *signed URIs*, which grant access to a resource for a limited amount of time to all the requests that can provide a temporary access token.

**(e) Advanced Features.** Besides the management of buckets, objects, and ACPs, S3 offers other additional features, which can be helpful. These are: server access logging and integration with the *BitTorrent* file sharing network.

[55] This group identifies a specific group of accounts that are used by automated process to perform bucket access logging.

Server access logging allows bucket owners to obtain detailed information about the request made for the bucket and all the objects it contains. By default, this feature is turned off and can be activated by issuing a put request to the bucket URI followed by *?logging*. The request should include an XML file specifying the target bucket where to save the logging files and the file name prefix. A GET request to the same URI allows retrieving the existing logging configuration for the bucket.

The second feature of interest is represented by the capability of exposing S3 objects to the *BitTorrent* network, thus allowing files stored in S3 to be downloaded by using the *BitTorrent* protocol. This is done by appending *?torrent* to the URI of the S3 object. In order to actually download the object, its ACP must grant read permission to everyone.

## 2. Amazon Elastic Block Store (EBS)

Amazon EBS allows AWS users to provide EC2 instances with persistent storage in the form of volumes that can be mounted at instance start-up. They accommodate up to 1 TB of space and are accessed through a block device interface, thus allowing users to format them according to the needs of the instance they are connected to (raw storage, file system, or other). The content of an EBS volume survives the instance lifecycle and it is persisted into S3. EBS volumes can be cloned, used as boot partitions, and constitute a durable storage since they rely on S3 and it is possible to take incremental snapshots of their content.

EBS volumes normally reside within the same availability zone of the EC2 instances that will use them in order to maximize the I/O performance. It is also possible to connect volumes located in different availability zones. Once mounted as volumes, their content is lazily loaded in background and according to the request made by the operating system. This reduces the amount of I/O requests that go to the network. Volume images cannot be shared among instance, but multiple (separate) active volumes can be created from them. Also, it is possible to attach multiple volumes to a single instance, or create a volume from a given snapshot and modify its size, if the formatted file system allows such operation.

The expense related to a volume is composed by the cost generated by the amount of storage occupied in S3 and by the number of I/O requests performed against the volume. At the time of writing, Amazon charges $0.10/GB/month of allocated storage and $0.10 per 1 million of requests made to the volume.

## 3. Amazon ElastiCache

ElastiCache is an implementation of an elastic in-memory cache based on a cluster of EC2 instances. It provides a fast data access from other EC2 instances through a Memcached compatible protocol so that existing applications based on such technology do not need to be modified and can transparently migrate to ElastiCache.

ElastiCache is based on a cluster of EC2 instances running the caching software, which is made available through Web services. An ElastiCache cluster can be dynamically resized according to the demand of the client applications. Also, automatic patch management and failure detection and recovery of cache nodes allow the cache cluster to keep running, without administrative intervention from AWS users, who have only to elastically size the cluster when needed.

ElastiCache nodes are priced according to the EC2 costing model, with a small price difference due to the use of the caching service installed on such instances. It is possible to choose between different type of instances, and the table below provides an overview of the different pricing options.

**Table 9.3.** Amazon EC2 (On-Demand) Cache Instances Characteristics.

| *Instance Type* | *ECU* | *Platform* | *Memory* | *I/O Capacity* | *Price (US East) (USD/hour)* |
|---|---|---|---|---|---|
| *Standard Instances* | | | | | |
| *Small* | 1(1x1) | 64 bit | 1.3 GB | Moderate | $0.095 |
| *Large* | 4(2x2) | 64 bit | 7.1 GB | High | $0.380 |
| *Extra Large* | 8(4x2) | 64 bit | 14.6 GB | High | $0.760 |

(*Continued*)

**Table 9.3.** Continued.

| | | | | | |
|---|---|---|---|---|---|
| *High Memory Instances* | | | | | |
| *Extra Large* | 6.5 (2x3.25) | 64 bit | 16.7 GB | High | $0.560 |
| *Double Extra Large* | 13 (4x3.25) | 64 bit | 33.8 GB | High | $1.120 |
| *Quadruple Extra Large* | 26 (8x3.25) | 64 bit | 68 GB | High | $2.240 |
| *High CPU Instances* | | | | | |
| *Extra Large* | 26 (8x3.25) | 64 bit | 6.6 GB | High | $0.760 |

The prices indicated in the table are related to the Amazon offering during 2011–2012 and the amount of memory specified represents the memory available after taking system software overhead into account.

## 4. Structured Storage Solutions

Enterprise applications quite often rely on databases to store data in a structured form, index, and perform analytics against it. Traditionally, RDBMS have been the common data backend for a wide range of applications, even though recently more scalable and lightweight solutions have been proposed. Amazon provides applications with structured storage services in three different forms: preconfigured EC2 AMIs, *Amazon Relational Data Storage (RDS)*, and *Amazon SimpleDB*.

**(a) Preconfigured EC2 AMIs.** These are predefined templates featuring an installation of a given database management system. EC2 instances created from these AMI can be completed with an EBS volume for storage persistence. Available AMIs include installations of IBM DB2, Microsoft SQL Server, MySQL, Oracle, PostgreSQL, Sybase, and Vertica. Instances are priced hourly according to the EC2 cost model. This solution poses most of the administrative burden on the EC2 user that has to configure, maintain, and manage the relational database but offers the greatest variety of products to choose from.

**(b) Amazon RDS.** RDS is relational database service relying on the EC2 infrastructure and managed by Amazon. Developers do not have to worry about the configuring the storage for high availability, designing failover strategies, or keeping the servers up-to-date with patches. Moreover, the service provides users with automatic backups, snapshots, point-in-time recoveries, and facilities for implementing replications. These and the common database management services are available through the AWS console or a specific Web service. Two relational engines are available: MySQL and Oracle.

Two key advanced features of RDS are *multi-AZ deployment* and *read replicas*. The first option provides users with a failover infrastructure for their RDBMS solution. The high availability solution is implemented by keeping in stand-by synchronized copies of the services in different availability zones that are activated in case the primary service goes down. The second option provides users with increased performance for applications that are heavily based on database reads. In this case, Amazon deploys copies of the primary service that are only available for database reads, thus cutting down the response time of the service.

The available options and the relative pricing of the service during 2011–2012 are shown in Table 9.4. The table shows the costing details of the on-demand instances. There is also the possibility of using reserved instance for long terms (1 to 3 years) by paying upfront at discounted hourly rates.

With respect to the previous solution, users are not responsible for managing, configuring, and patching the database management software, but thee operations are performed by the AWS. Also, support for elastic management of servers is simplified. Therefore, this solution is optimal for applications based on the Oracle and MySQL engines, which are migrated on the AWS infrastructure and require a scalable database solution.

**Table 9.4.** Amazon RDS (On-Demand) Instances Characteristics.

| *Instance Type* | *ECU* | *Platform* | *Memory* | *I/O Capacity* | *Price (US East) (USD/hour)* |
|---|---|---|---|---|---|
| *Standard Instances* | | | | | |
| *Small* | 1(1x1) | 64 bit | 1.7 GB | Moderate | $0.11 |
| *Large* | 4(2x2) | 64 bit | 7.5 GB | High | $0.44 |
| *Extra Large* | 8(4x2) | 64 bit | 15 GB | High | $0.88 |
| *High Memory Instances* | | | | | |
| *Extra Large* | 6.5 (2x3.25) | 64 bit | 17.1 GB | High | $0.65 |
| *Double Extra Large* | 13 (4x3.25) | 64 bit | 34 GB | High | $1.30 |
| *Quadruple Extra Large* | 26 (8x3.25) | 64 bit | 68 GB | High | $2.60 |

**(c) Amazon SimpleDB.** Amazon SimpleDB is a lightweight, highly scalable, and flexible data storage solution for applications that do not require a fully relational model for their data. SimpleDB provides support for semi-structured data, whose model is based on the concept of *domains*, *items*, and *attributes*. With respect to the relational model, this model provides less constraints on the structure of data entries, thus obtaining improved performance on querying large quantities of data. As happens for Amazon RDS, this service frees AWS users from performing configuration, management, and high-availability design for their data store.

SimpleDB uses *domains* as top-level elements to organize the data store. These are roughly comparable to tables in the relational model. Different from tables, they allow items to not have all the same column structure; each item is, therefore, represented as a collection of attributes expressed in the form of a key-value pair. Each domain can grow up to 10 GB of data and by default, a single user can allocate a maximum of 250 domains. Clients can create, delete, modify, and snapshot domains. They can insert, modify, delete, and query items and attributes. Batch insertion and deletion are also supported. The capability of querying data is one of the most relevant functions of the model and the *select* clause supports the following test operator: *=*, *!=*, *<*, *>*, *<=*, *>=*, *like*, *not like*, *between*, *is null*, *is not null*, and *every()*. A simple example on how to query data is given below:

*select * from domain_name where every(attribute_name) = 'value'*

Moreover, the select operator can extend its query beyond the boundaries of a single domain, thus allowing users to query effectively a large amount of data.

In order to efficiently provide AWS users with a scalable and fault tolerant service, SimpleDB implements a relaxed constraint model, which leads to *eventually consistent* data. The adverb eventually denotes the fact that multiple accesses on the same data might not read the same value in the very short term, but they will eventually converge over time. This is because SimpleDB does not lock all the copies of the data during an update, which is propagated in the background. Therefore, there is a transient period of time in which different clients can access different copies of the same data that have different values. This approach results very scalable with minor drawbacks, and it is also reasonable since the application scenario for SimpleDB is mostly characterized by querying and indexing operations on data. Alternatively, it is possible to change the default behavior and ensure that all the readers are blocked during an update.

Even though SimpleDB is not a transactional model, it allows clients to express conditional insertions or deletions, which are useful to prevent lost updates in multiple-writer scenarios. In this case, the operation is executed if and only if the condition is verified. This condition can be used to check pre-existing values of attributes for an item.

**Table 9.5.** Amazon SimpleDB Data Transfer Charges.

| *Instance Type* | *Price (US East) (USD)* |
|---|---|
| *Data Transfer In* | |
| *All data transfer in* | $0.000 |
| *Data Transfer Out* | |
| *1st GB / month* | $0.000 |
| *Up to 10 TB / month* | $0.120 |
| *Next 40 TB / month* | $0.090 |
| *Next 100 TB / month* | $0.070 |
| *Next 350 TB / month* | $0.050 |
| *Next 524 TB / month* | Special Arrangements |
| *Next 4 PB / month* | Special Arrangements |
| *Greater than 5 PB / month* | Special Arrangements |

Table 9.5 provides an overview of the pricing options for the SimpleDB service during 2011–2012 for data transfer. The service charges either for data transfer or stored data. Data transfer within the AWS network is not charged. In addition, SimpleDB charges users also for machine usage: the first 25 SimpleDB instances/month are free, and after this threshold, there is an hourly charge ($0.140 hour in US East region).

If we compare this cost model with the one characterizing S3, it becomes evident that S3 is a cheaper option for storing large objects. This is useful for clarifying the different nature of SimpleDB with respect to S3: the former has been designed to provide fast access to semi-structured collections of small objects and not for being a long-term storage for large objects.

## 5. Amazon CloudFront

CloudFront an implementation of a content delivery network on top of the Amazon distributed infrastructure. It leverages a collection of edge servers strategically located on the globe to better serve requests of static and streaming Web content, so that the transfer time is reduced as much as possible.

AWS provides users with simple Web service APIs to manage CloudFront. In order to make available content through CloudFront it is necessary to create a distribution. This identifies an origin server, which contains the original version of the content being distributed, and it is referenced by a DNS domain under the *Cloudfront.net* domain name (i.e., my-distribution.Cloudfront.net). It is also possible to map a given domain name to a distribution. Once the distribution is created it is sufficient to reference the distribution name and the CloudFront engine will redirect the request to closest replica and eventually download the original version from the origin server, if the content is not found or expired on the selected edge server.

The content that can be delivered through CloudFront is static (HTTP and HTTPS) or streaming (RMTP). The origin server hosting the original copy of the distributed content can be an S3 bucket, an EC2 instance, or a server external to the Amazon network. Users can restrict the access to the distribution to only one or few of the available protocols or they can set up access rules for a finer control. It is also possible to invalidate content to remove it from the distribution of force its update before expiration.

Table 9.6 provides a breakdown of the pricing during 2011–2012. It can be noticed that CloudFront is cheaper than S3. This reflects its different purpose: CloudFront is designed to optimize the distribution of very popular content that gets frequently downloaded, potentially from the entire globe and not only the Amazon network.

**Table 9.6.** Amazon Cloud Front On-Demand Pricing.

| *Pricing Item* | *United States* | *Europe* | *Hong Kong and Singapore* | *Japan* | *South America* |
|---|---|---|---|---|---|
| *Request* | | | | | |
| *Per 10000 HTTP Requests* | $0.0075 | $0.0090 | $0.0090 | $0.0095 | $0.0160 |
| *Per 10000 HTTPS Requests* | $0.0100 | $0.0120 | $0.0120 | $0.0130 | $0.0220 |
| *Regional Data Transfer Out* | | | | | |
| *First 10 TB / month* | $0.120 / GB | $0.120 / GB | $0.190 / GB | $0.201 / GB | $0.250 / GB |
| *Next 40 TB / month* | $0.080 / GB | $0.080 / GB | $0.140 / GB | $0.148 / GB | $0.200 / GB |
| *Next 100 TB / month* | $0.060 / GB | $0.060 / GB | $0.120 / GB | $0.127 / GB | $0.180 / GB |
| *Next 350 TB / month* | $0.040 / GB | $0.040 / GB | $0.100 / GB | $0.106 / GB | $0.160 / GB |
| *Next 524 TB / month* | $0.030 / GB | $0.030 / GB | $0.080 / GB | $0.085 / GB | $0.140 / GB |
| *Next 4 PB / month* | $0.025 / GB | $0.025 / GB | $0.070 / GB | $0.075 / GB | $0.130 / GB |
| *Greater than 5 PB / month* | $0.020 / GB | $0.020 / GB | $0.060 / GB | $0.065 / GB | $0.125 / GB |

## 9.1.3 Communication Services

Amazon provides facilities to structure and facilitate the communication among existing applications and services residing within the AWS infrastructure. These facilities can be organized into two major categories: *virtual networking* and *messaging*.

### *1. Virtual Networking*

Virtual networking comprises a collection of services allowing AWS users to control the connectivity to compute-and-storage services and between them. *Amazon Virtual Private Cloud (VPC)* and *Amazon Direct Connect* provide connectivity solutions in terms of infrastructure, while *Route 53* facilitates connectivity in terms of naming.

Amazon VPC provides a great degree of flexibility in creating virtual private networks within the Amazon infrastructure and beyond. The service provides either prepared templates covering most of the usual scenarios or a fully customizable network service for advanced configurations. Prepared templates include public subnets, isolated networks, private networks accessing Internet through NAT, and hybrid networks including AWS resources and private resources. Also, it is possible to control connectivity between different services (EC2 instances and S3 buckets), by using the *Identity Access Management (IAM)* service. During 2011, the cost of Amazon VPC was $0.50 per connection-hour.

Amazon Direct Connect allows AWS users to create dedicated networks between the user private network and Amazon Direct Connect locations, called *ports*. This connection can be further partitioned in multiple logical connections and give access to the public resources hosted on the Amazon infrastructure. The advantage of using Direct Connect versus other solutions is the consistent performance of the connection between the users' premises and the Direct Connect locations. This service is compatible with other services such as EC2, S3, and Amazon VPC and can be used in scenarios requiring high bandwidth between the Amazon network and the outside world. At the time of writing, there are only two available ports located in the US, but users can leverage external providers offering guaranteed high bandwidth to these ports. Two different bandwidths can be chosen: 1 Gpbs, priced at $0.30 dollars/hour, and 10 Gbps, priced at $2.25 dollars/hour. Inbound traffic is free, while outbound traffic is priced at $0.02 dollars/GB.

*Amazon Route* 53 implements dynamic DNS services allowing AWS resources to be reached through domain names different from the *amazon.com* domain. By leveraging the large and globally distributed network of Amazon DNS servers, AWS users can expose EC2 instances or S3 buckets as resources

under a domain of their property, for which Amazon DNS servers become authoritative[56]. EC2 instances are likely to be more dynamic than the physical machines and S3 buckets might also exist for limited time. In order to cope with such volatile nature, the service provides AWS users with the capability of dynamically mapping names to resources as instances are launched on EC2 or new buckets are created in S3. By interacting with Route 53 Web service, the user can manage a set of *hosted zones*, which represent the user domains controlled by the service, and edit the resources made available through it. At the time of writing, a single user can have up to 100 zones. The costing model includes a fixed amount ($1 dollars/zone/month) and a dynamic component depending on the number of queries resolved by the service for the hosted zones ($0.50 dollars/million queries for the first billion of queries a month, $0.25 dollars/million queries over 1 billion of queries a month).

## 2. Messaging

Messaging services constitute the next step in connecting applications by leveraging AWS capabilities. The three different types of messaging services offered are: *Amazon Simple Queue Service (SQS)*, *Amazon Simple Notification Service (SNS)*, and *Amazon Simple Email Service (SES)*.

Amazon SQS constitutes a disconnected model for exchanging messages between applications by means of message queues, hosted within the AWS infrastructure. By using the AWS console or directly the underlying Web service, AWS users can create an unlimited number of message queues and configure them to control their access. Applications can send an unlimited number of messages to any queue they have access to. These messages are securely and redundantly stored within the AWS infrastructure for a limited period of time, in which they can access by other (authorized) applications. While a message is being read, it is kept locked to avoid spurious processing from other applications. Such a lock will expire after a given period.

Amazon SNS provides publish-subscribe method for connecting heterogeneous applications. With respect to Amazon SQS, where it is necessary to continuously poll a given queue for new message to process, Amazon SNS allows applications to be notified when new content of interest is available. This feature is accessible through a Web service where AWS users can create a topic, which other applications can subscribe to. At any time, applications can publish content on a given topic and subscribers get automatically notified. The service provides subscribers with different notification models (HTTP/HTTPS, E-mail/E-mail JSON, and SQS).

Amazon SES provides AWS users with a scalable email service that leverages the AWS infrastructure. Once users are signed up with the service, they have to provide an email that will be used by SES to send emails on their behalf. To activate the service, SES will send a email to verify the given address and provide the users with the information necessary for the activation. Upon verification, the user is given a SES sandbox to test the service, and access to the production version can be requested. By using SES, it is possible to send either SMTP compliant emails or raw emails by specifying emails headers and MIME types. Emails are queued for delivery and the users are notified of any failed delivery. SES also provides a wide range of statistics helping users to improve their email campaign for effective communications with customers.

With regards to the costing, all of the three services do not require a minimum commitment but are based on pay-as-you go model. Currently, users are not charged until they reach a minimum threshold. All the data transfer in does not incur in any charge, while data transfer out is charged by ranges.

# 9.1.4 Additional Services

Besides compute, storage, and communication services, AWS provides a collection of services allowing users to exploit at best Amazon Cloud computing offering as a whole. Among these two are particularly relevant: *Amazon CloudWatch* and *Amazon Flexible Payment Service (FPS)*.

[56] A DNS server is responsible for resolving a name to a corresponding IP address, since DNS servers implement a distributed database without a single global control. A single DNS server does not have the complete knowledge of all the mappings between names and IP addresses, but it has direct knowledge only of a small subset of them. Such DNS server is therefore authoritative for these names, because it can resolve directly the names. For resolving the other names, the nearest authoritative DNS is contacted.

Amazon CloudWatch is a service that provides a comprehensive set of statistics helping developers to understand and optimize the behavior of their application hosted on AWS. CloudWatch collect information from several other AWS services: EC2, S3, SimpleDB, CloudFront, and others. Initially accessible through additional subscription, now is made available for free to all the AWS users. By using CloudWatch, developers can have a detailed breakdown of the usage of the service they are renting on AWS and devise more efficient and cost saving applications.

Amazon FPS is an infrastructure allowing AWS users to leverage Amazon's billing infrastructure to sell goods and services to other AWS users. By using Amazon FPS developers do not have to set up alternative payment methods, but can charge users by using a billing service, which is already familiar to them. The payment models available with FPS include: one-time payments, delayed and periodic payments, required by subscriptions and usage-based services, transactions, and aggregate multiple payments.

### 9.1.5 Summary

Amazon provides a complete set of services for developing, deploying, and managing Cloud computing systems by leveraging the large and distributed AWS infrastructure. Developers can use EC2 to fully control and shape the computing infrastructure hosted in the Cloud or leverage other computing services, such as AWS CloudFormation, Elastic Beanstalk or Elastic MapReduce, if they do not need complete control over the computing stack. Applications hosted in the AWS Cloud, can leverage S3, SimpleDB, or other storage services to manage structured and unstructured data. These services are primarily meant for storage, while other options such as Amazon SQS, SNS, and SES provide solutions for dynamically connecting applications both from inside and outside the AWS Cloud. Network connectivity to AWS applications are addressed by Amazon VPC and Amazon Direct Connect.

## 9.2 GOOGLE APPENGINE

Google AppEngine is a Platfom-as-a-Service implementation providing services for developing and hosting scalable Web applications. AppEngine is essentially a distributed and scalable runtime environment that leverages Google's distributed infrastructure to scale out applications facing a large amount of requests by allocating more computing resources to them and balancing the load among them. The runtime is completed by a collection of services allowing developers to design and implement applications that naturally scale on AppEngine. Developers can develop applications in Java, Python, and Go, a new programming language developed by Google to simplify the development of Web applications. Application's usage of Google resources and services is constantly metered by AppEngine that bills users when their applications trespass free quotas.

### 9.2.1 Architecture and Core Concepts

AppEngine is a platform for developing scalable applications accessible through the Web. The platform is can be logically divided into four major components: infrastructure, the runtime environment, the underlying storage, and the set of scalable services that can be used to develop applications.

#### *1. Infrastructure*

AppEngine hosts Web applications and its primary function is to serve users requests efficiently. In order to do so, AppEngine's infrastructure takes advantage of the many servers available within Google datacenters and for each HTTP request locates the servers hosting the application processing the request, evaluates their load, and if necessary allocates more resources (i.e., servers) otherwise redirects the request to an existing server. The particular design of applications, which does not expect any state information to be implicitly maintained between requests to the same application, simplifies the work of the infrastructure that can redirect each of the requests to any of the server hosting the target application or even allocate a new one.

The infrastructure is also responsible for monitoring the application performance and for collecting statistics on top of which the billing is calculated.

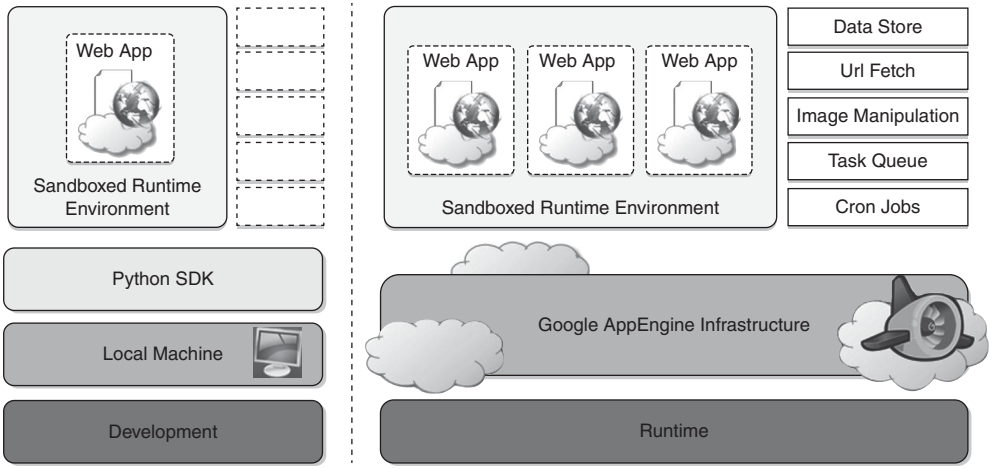


**Fig. 9.2.** Google AppEngine Platform Architecture.

## 2. *Runtime Environment*

The runtime environment represents the execution context of applications hosted on AppEngine. With reference to the AppEngine infrastructure code, which is always active and running, the runtime comes into existence when the request handler starts to execute and terminates once the handler has completed.

**(a) Sandboxing.** One of the major responsibilities of the runtime environment is to provide application environment with an isolated and protected context where they can execute without constitute a threat to the server and without being influenced by other applications. In other words, it provides applications with a *sandbox*.

Currently, AppEngine supports applications that are developed only with managed or interpreted languages, which by design require a runtime translating their code into executable instructions. Therefore, sandboxing is achieved by means of modified runtimes for applications that disable some of the common features normally available with their default implementations. If an application tries to perform any operation considered potentially harmful, an exception is thrown and the execution is interrupted. Some of the operations not allowed in the sandbox are: writing to the server's file system; accessing computer through network besides using *Mail*, *UrlFetch*, and *XMPP*; executing code outside the scope of a request, a queued task, and a cron job; and processing a request for more than 30 seconds.

**(b) Supported Runtimes.** Currently, it is possible to develop AppEngine applications by using three different languages and related technologies: *Java*, *Python*, and *Go*.

AppEngine currently supports Java 6 and developers can use the common tools for Web applications development in Java, such as the *Java Server Pages (JSP)* and the applications interact with the environment by using the *Java Servlet* standard. Also, access to AppEngine services is provided by means of Java libraries that expose specific interfaces of provider specific implementations of given abstraction layer. Developers can create applications with the AppEngine Java SDK that allows developing application with either Java 5 or Java 6 and by using any Java library that does not exceed the restrictions imposed by the sandbox.

Support for Python is provided by an optimized interpreter supporting Python 2.5.2. As happens for Java, the runtime environment supports the Python standard library but some of the modules that implement potentially harmful operations have been removed and attempts to import such modules or call specific methods generate exceptions. Regarding application development, AppEngine comes with a rich set of libraries connecting applications to AppEngine services. Also, developers can use a specific Python Web application framework, called *Webapp*, simplifying Web application development.

The Go runtime environment allows applications developed with the Go programming language to be hosted and executed in AppEngine. The current release of Go supported by AppEngine is r58.1. The SDK includes the compiler and the standard libraries for developing applications in Go and interfacing it with AppEngine services. As happens for the Python environment some of the functionalities have been removed or generate a run-time exception. Also, developers can include third party libraries into their applications as long as they are implemented in pure Go.

## 3. *Storage*

AppEngine provides different types of storage, which operate differently according to the volatility of the data they are designed for. There are three different level of storage: in memory-cache, storage for semi-structured data, and long-term storage for static data. In this section, we describe *DataStore* and the use of static file servers and we cover *MemCache* in the application services section.

**(a) Static File Servers.** Web applications are composed by dynamic and static data. Dynamic data is a result of the logic of the application and the interaction with the user. Static data often is mostly constituted by the components that define the graphical layout of the application (css files, plain html files, javascript files, images, icons, and sound files) or data files. These files can be hosted on static file servers, since they are not frequently modified. Such servers are optimized for serving static content and users can specify how dynamic content should be served when uploading their applications to AppEngine.

**(b) DataStore.** DataStore is a service allowing developers to store semi-structured data. The service is designed to scale and optimized to quickly access data. DataStore can be considered as a large object database where to store objects that can be retrieved by a specified key. Both the type of the key and the structure of the object can vary.

With respect to the traditional Web applications backed by a relational database, DataStore imposes less constraint on the regularity of the data but, at the same time, does not implement some of the features of the relational model (such as reference constraints and join operations). These design decisions originated from a careful analysis of data usage patterns for Web applications and have been taken in order to obtain a more scalable and efficient data store. The underlying infrastructure of *DataStore* is based on *Bigtable* [93], which is a redundant, distributed, and semi-structured data store that organizes data in the form of tables (See Section 8.2.1).

DataStore provides high-level abstractions that simplify the interaction with Bigtable. Developers define their data in terms of *entity* and *properties*, and these are persisted and maintained by the service into tables in *Bigtable*. An entity constitutes the level of granularity for the storage and it identifies a collection of properties, which define the data it stores. Properties are defined according to one of the several primitive types supported by the service. Each entity has associated a key, which is either provided by the user of created automatically by AppEngine. An entity is associated with a *named kind* that is used by AppEngine to optimize its retrieval from Bigtable. Although entities and properties seem to be similar to rows and tables in SQL, there are a few differences that have to be taken account. Entity of the same kind might not have the same properties, and properties of the same name might contain values of different types. Moreover, properties can store different version of the same values. Finally, keys are immutable elements and once created, they cannot be changed anymore.

DataStore also provides facilities for creating indexes on data and to update data within the context of a transaction. Indexes are used to support and speed-up queries. A query can return zero or more objects of the same kind or simply the corresponding keys. It is possible to query the data store either by specifying the key or conditions on the values of the properties. Returned result sets can be sorted either by key value or properties value. Even though the queries are quite similar to SQL queries, their implementation is substantially different. DataStore has been designed to be extremely fast in returning result sets. In order to do so, it needs to know in advance all the possible queries that can be done for a given kind because it stores for each of them a separate index. The indexes are provided by the user while uploading the application to AppEngine and can be automatically defined by the development server: when the developer tests the application, the server monitors all the different types of query made against the simulated data store and creates an index for them. The structure of the indexes is saved into a configuration file and can be further changed by the developer before uploading the application. The use of pre-computed indexes makes the query execution time independent from the size of the stored data, but only influenced by the size of the result set.

The implementation of transaction is limited in order to keep the store scalable and fast. AppEngine ensures that the update of a single entity is performed automatically. Multiple operations on the same entity can be performed within the context of a transaction. It is also possible to update multiple entities automatically.This is only possible if these entities belong to the same *entity group.* The entity group to which an entity belongs is specified at the time of entity creation and cannot be changed later. With regards to concurrency, AppEngine uses an *optimistic concurrency control*: if one user tries to update an entity that is being updated, the control returns and the operation fails. Retrieving an entity never incurs into exceptions.

## 4. Application Services

Applications hosted on AppEngine take the most from the services made available through the runtime environment. These services simplify most of the common operations that are performed in Web applications: access to data, account management, integration of external resources, messaging and communication, image manipulation, and asynchronous computation.

**(a) UrlFetch.** Web 2.0 has introduced the concept of composite Web applications. Different resources are put together and organized as meshes within a single Web page. Meshes are fragment of HTML generated in different ways: they can be directly obtained from a remote server, they rendered from an XML document retrieved from a Web services or rendered by the browser as the result of an embedded and remote component. A common characteristic of all these examples is the fact that the resource is not local to the server, and often not even in the same administrative domain. It is therefore fundamental for Web applications to be able to retrieve remote resources.

The sandbox environment does not allow applications to open arbitrary connections through sockets, but provides developers with the capability of retrieving a remote resource through HTTP/HTTPS by means of the *UrlFetch* service. Applications can make synchronous and asynchronous Web requests and integrate the resources obtained in this way into the normal request handling cycle of the application. One of the interesting features of *UrlFetch* is the ability to set a deadline for a request, so that they can be completed (or aborted) within a given time. Moreover, the ability of performing such requests asynchronously allows the applications to continue with their logic while the resource is retrieved in background. *UrlFetch* is not only used to integrate meshes into a Web page, but also to leverage remote Web services in accordance with the SOA reference model for distributed applications.

**(b) MemCache.** AppEngine provides developers with access to a fast and reliable storage, which is *DataStore*. Despite this, the main objective of the service is to serve as a scalable and long-term storage, where data is persisted to disk redundantly in order to ensure reliability and availability of

data against failures. This design poses a limit on how much fast the store can be when compared to other solutions, especially for the case of objects that are frequently accessed, let's say at each Web request.

AppEngine provides caching services by means of *MemCache*. This is a distributed in-memory cache that is optimized for fast access and provides developer with a volatile store for the objects that are frequently accessed. The caching algorithm implemented by *MemCache* will automatically remove the objects that are accessed rarely. The use of *MemCache* can significantly reduce the access time to data: developers can structure their applications so that each object is first looked up into *MemCache* and if there is a miss, it will be retrieved from *DataStore* and put into the cache for future look-ups.

**(c) Mail and Instant Messaging.** Communication is another important aspect of Web applications. It is common the use of e-mails for following up with users about operations performed by the application. E-mail can also be used to trigger activities into Web applications. In order to facilitate the implementation of such tasks, AppEngine provides developers with the ability of sending and receiving mails through *Mail*. The service allows sending email on behalf of the application of specific user accounts. It is also possible to include several types of different attachments and target multiple recipients. Mail operates asynchronously and in case of failed delivery, the sending address is notified through an email detailing the error.

AppEngine provides also another way to communicate with the external world: the XMPP protocol. Any chat service supporting XMPP, such as Google Talk can send and receive chat messages to and from the Web application, which is identified by its own address. Even though the chat is communication media mostly used for human interactions, XMPP can be conveniently used to connect the Web application with chat bots or to implement a small administrative console.

**(d) Account Management.** Web applications often keep various data that customize the interaction that they have with users. These data normally goes under the name of user profile and are attached to an account. AppEngine simplifies account management by allowing developers to leverage Google accounts management, by means of *Google Accounts*. The integration with the service also allows Web applications to offload the implementation of authentication capabilities by offloading such a task to the Google's authentication system.

By using *Google Accounts,* Web applications can conveniently store profile settings into the form of key-value pair, attach them to given Google account, and quickly retrieve them once the user authenticates. With respect to a custom solution, the use of *Google Accounts* requires users to be have a Google account, but does not require any further implementation. The use of *Google Accounts* is particularly advantageous when developing Web applications within a corporate environment using *Google Apps*. In this case, the applications can be easily integrated with all the other services (and profile settings) included in *Google Apps*.

**(e) Image Manipulation.** Web applications render pages with graphics. Often it is required to perform simple operations such as add watermarks or apply simple filters. AppEngine allows applications to perform image resizing, rotation, mirroring, and enhancement by means of *Image Manipulation*, a service that is also used in other Google products. *Image Manipulation* is mostly designed for lightweight image processing and optimized for speed.

## *5. Compute Services*

Web applications are mostly designed to interface the applications with users by means of a ubiquitous channel, i.e., the Web. Most of the interaction is performed synchronously: users navigate the Web pages and an instantaneous feedback is received as a response their actions. This feedback is often the result of some computation happening on the Web application, which implements the intended logic to serve the user request. Sometimes, this approach is not applicable, for example in case of long computations or when some operations need to be triggered at a given point in time. A good design

for these scenarios provides the user with an immediate feedback and a notification once the required operation is completed. AppEngine, include services that simplify the execution of computations that are off-bandwidth, or, in other words, that cannot be contained within the time frame of the Web request handling. These are *Task Queues* and *Cron Jobs*.

**(a) Task Queues.** *Task Queues* allow applications to submit a task for later execution. This service is particularly useful for long computations that cannot complete within the maximum response time of a request handler. The service allows users to have up to 10 queues that can execute tasks at a configurable rate.

In fact, a task is defined by a Web request to a given URL and the queue invokes the request handler by passing the payload as part of the Web request to the handler. It is the responsibility of the request handler to perform the "task execution", which is seen from the queue as a simple Web request. The queue is designed to re-execute the task in case of failure in order to avoid that transient failures prevent the task from a successful completion.

**(b) Cron Jobs.** Sometimes the length of computation might not be the primary reason why an operation is not performed within the scope of the Web request. It might be possible that the required operation needs to be performed at a specific time of the day, which does not coincide with the time of the Web request. In this case, it is possible to schedule the required operation at the desired time, by using the *Cron Jobs* service. The service operates similarly to Task Queues, but invokes the request handler specified in the task at a given time and does not re-execute the task in case of failure. This behavior can be useful to implement maintenance operations or to send periodic notifications.

## 9.2.2 Application Life Cycle

AppEngine provides support for almost all the phases characterizing the life cycle of an application: test and development, deployment, and monitoring. The SDKs released by Google provide developers with most of the functionalities required by these tasks. Currently there are two SDKs available for development: the Java SDK and the Python SDK.

### *1. Application Development and Testing*

Developers can start building their Web applications on a local development server. This is a self-contained environment that helps developers in tuning applications without uploading them to AppEngine. The development server simulates the AppEngine runtime environment by providing a mock implementation of *DataStore*, *MemCache*, *UrlFetch*, and the other services leveraged by Web applications. Besides hosting Web applications, the development server also features a complete set of monitoring features that are helpful to profile the behavior of applications, especially for what regards the access to the *DataStore* service and the queries performed against it. This is a particularly important feature that will be of relevance while deploying the application to AppEngine. As discussed in the previous section, AppEngine builds indexes for each of the queries performed by a given application in order to speed up the access to the relevant data. This capability is enabled by a priori knowledge about all the possible queries made by the application; such knowledge is made available to AppEngine by the developer while uploading the application. The development server analyses application behavior while it is running and traces all the queries made during the test and development, thus providing the required information about the indexes to be built.

**(a) Java SDK.** The Java SDK provides developers with facility for building applications with the Java 5 and the Java 6 runtime environments. Alternatively, it is possible to develop applications within the Eclipse development environment by using the Google AppEngine plug-in, which integrates the features of the SDK within the powerful Eclipse environment. By using the Eclipse software installer, it is possible to download and install into Eclipse the Java SDK, the Google Web Toolkit, and the Google

AppEngine plugin. These three components allow developers to program powerful and rich Java applications for AppEngine.

The SDK supports the development of applications by using the *servlet* abstraction, which is the common development model for applications. Together with servlets, many other features are available to build applications. Moreover, developers using Eclipse and the Google AppEngine plugin can develop Web applications by using the *Eclipse Web Platform*, which provides a set of tools and components that simplify Web development.

The plugin allows developing, testing, and deploying applications on AppEngine. Other tasks, such as retrieving the log of applications, are available by means of command line tools that are part of the SDK.

**(b) Python SDK.** The Python SDK allows developing Web applications for AppEngine with Python 2.5. It provides a standalone tool, called *GoogleAppEngineLauncher*, for managing Web application locally and deploying them to AppEngine. The tool provides a convenient user interface that lists all the available Web applications, controls their execution, and integrates with the default code editor for editing applications files. Also the launcher provides access to some important services for application monitoring and analysis such as the logs, the SDK console, and the dashboard. The logs console captures all the information that is logged by the application while running. The console SDK provides developers with a Web interface where they can see the application profile in terms of resource utilized. This feature is particularly useful since it allows previewing the behavior of applications once they are deployed on AppEngine and it can be used to tune applications with regards to the services made available through the runtime.

The python implementation of the SDK also comes with an integrated Web application framework called *Webapp* that includes a set of models, components, and tools that simplify the development of Web applications and enforce a set of coherent practices. This is not the only Web framework that can be used to develop Web applications. There exists dozens of Python Web frameworks available that can be used; because of the restrictions enforced by the sandboxed environment not all of them can no be used seamlessly. The *Webapp* framework has been re-implemented and made available in the Python SDK so that it can be used with AppEngine. Another Web framework that is known to work well is *Django*[57].

The SDK is completed by a set of command-line tools that allows developers to perform all the operations available through the launcher and more from the command shell.

## 2. Application Deployment and Management

Once the application has been developed and tested, it can be deployed on AppEngine with a simple click or command line tool. Before performing such task, it is necessary to create an application identifier, which will be used to locate the application from the Web browser by typing the address: **http://<application-id>.appspot.com**. Alternatively, it is also possible to map the application with a registered DNS domain name; this is particularly useful for commercial development where users want to make the application available through a more appropriate name.

An application identifier is mandatory, since it allows unique identification of the application while interacting with AppEngine. It is used to upload and to update applications by developers and it constitutes part of the email through which applications are accessible via XMPP. Besides being unique, it also needs to be compliant with the rules that are enforced for domain names. It is possible to register an application identifier by logging into AppEngine and selecting the “create application” option. It is also possible to provide an application title, which is descriptive of the application. Differently from the application identifier, the application title can be changed over time.

Once an application identifier has been created, it is possible to deploy an application on AppEngine. This task can be done either by using the respective development environment (*GoogleAppEngineLauncher*

[57] ***http://www.djangoproject.com***

*and Google AppEngine* Plugin) or the command line tools. Once the application is uploaded, nothing else needs to be done in order to make it available. AppEngine will take care of everything. Developers can then manage the application by using the administrative console. This is the primary tool used for application monitoring and provides users with insights on the usage of resources (CPU, and bandwidth), services and other useful counters. It is also possible to manage multiple versions of a single application and select the one available for the release and manage its billing related issues.

## 9.2.3 Cost Model

AppEngine provides a free service with limited quotas that get reset every 24 hours. Once the application has been tested and tuned for AppEngine, it is possible if needed to set up a billing account and obtain more allowance, charged on a pay-per-use basis. This allows developers to identify the appropriate daily budget that they want to allocate for a given application.

An application is measured against *billable quotas*, *fixed quotas*, and *per-minute quotas*. Google AppEngine uses these quotas in order to ensure that users do not spend more than the allocated budget and that applications run without being influenced by each other from a performance point of view. Billable quotas identify the daily quotas that are set by the application administrator and are defined by the daily budget allocated for the application. AppEngine will ensure that the application does not exceed these quotas. Free quotas are part of the billable quota and identify the portion of the quota for which users are not charged. Fixed quotas are internal quotas set by AppEngine that identify the infrastructure boundaries and define operations that applicants can carry out on the infrastructure (services and run-time). These quotas are generally bigger than billable quotas and are set by AppEngine in order to avoid impact of the applications on each other's performance, or that overloading the infrastructure. The costing model also includes per-minute quotas, which are defined in order to avoid that applications consume all their credit in a very limited period of time, monopolize a resource, and create service interruption for other applications.

Once an application reaches the quota for a given resource, the resource is depleted and will not be available to the application until the quota get replenished. Once a resource is depleted, subsequent request to that resource will generate an error or an exception. Resources such as CPU time and incoming or outgoing bandwidth will return a HTTP 403 error page to the users, all the other resources and services will generate an exception that can be trapped in code in order to provide a more useful feedback to users.

Resources and services quotas are organized into free default quotas and billing enabled default quotas. For these two categories, a daily limit and a maximum rate are defined. A detailed explanation of how quotas work, their limit, and the amount that is charged to the user can be found on the AppEngine Website at the following Internet address: *http://code.google.com/appengine/docs/quotas.html.*

## 9.2.4 Observations

AppEngine is a framework for developing a scalable Web application that leverages Google's infrastructure. The core components of the service are a scalable and sandboxed runtime environment for executing applications and a collection of services that implement most of the common features required for Web development and help developers in building applications that are easy to scale. One of the characteristic elements of AppEngine is the use of simple interfaces allowing applications to perform specific operations that are optimized and designed to scale. By building on top of these blocks, developers can build applications and let AppEngine scale them out when it is needed.

With respect to the traditional approach to Web development, the implementation of rich and powerful applications requires a change of perspective and more effort. Developers have to get familiar with the capabilities of AppEngine and implement the required features in a way that is conformant with AppEngine application model.

# 9.3 MICROSOFT AZURE

*Microsoft Windows Azure* is Cloud operating system built on top of Microsoft data centers' infrastructure and provides developers with a collection of services for building application with the Cloud technology. Services range from compute, storage, and networking to application connectivity, access control, and business intelligence. Any application that is build on the Microsoft technology can be scaled by using the Azure platform, which integrates the scalability features into the common Microsoft technologies such as Microsoft Windows Server 2008, SQL Server, and ASP.NET.

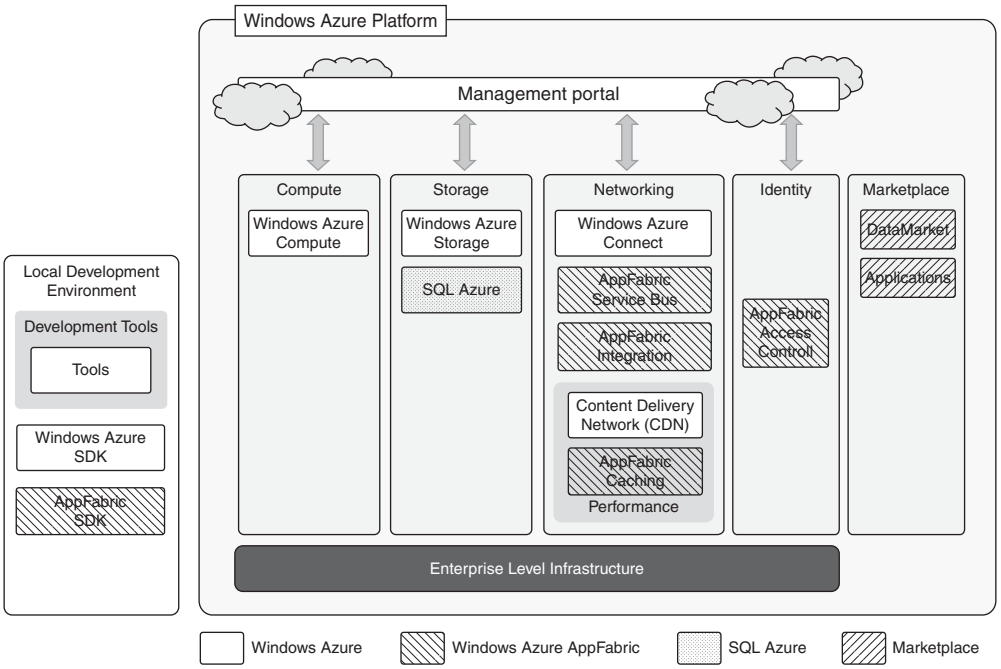


**Fig. 9.3.** Microsoft Windows Azure Platform Architecture.

Fig. 9.3 provides an overview of services provided by Azure. These services can be all managed and controlled through the *Windows Azure Management Portal*, which acts as administrative console for all the services of the Azure platform. In this section, we present the core features of the major services available with Azure.

## 9.3.1 Azure Core Concepts

The Windows Azure platform is composed by a foundation layer and a set of developer services, which can be used for building scalable applications. These services cover compute, storage, networking, and identity management, which are tied together by middleware called *AppFabric*. This scalable computing environment is hosted within Microsoft data centers and accessible through the Windows Azure Management Portal. Alternatively, developers can recreate a Windows Azure environment (with limited capabilities) on their own machine for development and testing purposes. In this section, we provide an overview of the Azure middleware and the services it offers.

### *1. Compute Services*

Compute services constitute the core components of Microsoft Windows Azure, and they are delivering by means of the abstraction of *roles*. A role is runtime environment that is customized for a specific

compute task. Roles are managed by the Azure operating system and instantiated on demand in order to address surges in application demand. Currently, there are three different roles: *Web role*, *Worker role*, and *Virtual Machine (VM) role*.

**(a) Web Role.** The Web role is designed to implement scalable Web applications. Web roles represent the units of deployment of Web applications within the Azure infrastructure. They are hosted within the IIS 7 Web Server, which is a component of the infrastructure that supports Azure. When Azure detects peak loads in the request made to a given application, it instantiates multiple Web roles for that application and distributes the load among them by means of a load balancer.

Since version 3.5, the .NET technology natively supports Web roles: developers can directly develop their applications in Visual Studio, test them locally, and upload to Azure. It is possible to develop ASP.NET (*ASP.NET Web Role* and *ASP.NET MVC 2 Web Role*) and WCF (*WCF Service Web Role*) applications. Since IIS 7 also supports the PHP runtime environment by means of the FastCGI module, Web roles can be used to run and scale PHP Web applications on Azure (*CGI Web Role*). Other Web technologies, which are not integrated with IIS, can still be hosted on Azure (i.e., Java Server Pages on Apache Tomcat) but there is no advantage in using a Web role, with respect to a worker role.

**(b) Worker Role.** Worker roles are designed to host general compute services on Azure. They can be either used to quickly provide compute power or to host services that do not communicate with the external world through HTTP. A common practice for Worker Roles is to use them to provide background processing for Web applications developed within Web Roles.

Developing a worker role is like developing a service. Differently from a web role whose computation is triggered by the interaction with an HTTP client (i.e., a browser), a worker role runs continuously, since when an instance of it is created until it is shut down. The Azure SDK provides developers with convenient APIs and libraries that allow connecting the role with the service provided by the runtime and easily controlling is startup as well as being notified with the changes in the hosting environment. As happens for Web Roles, the .NET technology provides complete support for worker roles, but any technology that runs on a Windows Server stack can be used to implement its core logic. For example, worker roles can be used to host Tomcat and serve JSP-based applications.

**(c) Virtual Machine Role.** The *Virtual Machine Role* allows developers to fully control the computing stack of their compute service, by defining a custom image of the Windows Server 2008 R2 operating system and all the service stack required by their applications. The Virtual Machine Role is based on the Windows Hyper-V virtualization technology (see Section 3.6.3), which is natively integrated in the Windows server technology at the base of Azure. Developers can image a windows server installation complete of all the required applications and components, save into a Virtual Hard Disk (VHD) file and upload it to Windows Azure to create compute instances on demand. Different types of instances are available and Table 9.7 provides an overview of the options offered during 2011–2012

**Table 9.7.** Windows Azure Compute Instances Characteristics.

| *Compute Instance Type* | *CPU* | *Memory* | *Instance Storage* | *I/O Performance* | *Hourly Cost (USD)* |
|---|---|---|---|---|---|
| *Extra Small* | 1.0 GHz | 768 MB | 20 GB | Low | $0.04 |
| *Small* | 1.6 GHz | 1.75 GB | 225 GB | Moderate | $0.12 |
| *Medium* | 2 x 1.6 GHz | 3.5 GB | 490 GB | High | $0.24 |
| *Large* | 4 x 1.6 GHz | 7 GB | 1000 GB | High | $0.48 |
| *Extra Large* | 8 x 1.6 GHz | 14 GB | 2040 GB | High | $0.96 |

With respect to the worker role and web role, the VM role provides a finer control on the compute service and resource that are deployed on the Azure Cloud, at the same time it also requires more administrative effort for services configuration, installation, and management.

## 2. Storage Services

Compute resources are equipped with local storage in the form of a directory on the local file system that can be used to temporarily store information useful for the current execution cycle of a role. If the role is restarted and activated on a different physical machine, this information gets lost.

Windows Azure provides for different types of storage solutions, which complement compute services with a more durable and redundant option if compared to local storage. Differently from local storage, these services can be accessed by multiple clients at the same time and from everywhere, thus becoming a general solution for storage.

**(a) Blobs.** Azure allows storing large amount of data in the form of Binary Large Objects (BLOBs) by means of the *blobs* service. This service is optimal to store large text of binary files. Two types of blobs are available:

*Block Blobs.* Block blobs are composed by blocks and they are optimized for sequential access and therefore they are appropriate for media streaming. At the time of writing (during 2011-2012), blocks are of 4MB and a single block blob can reach 200 GB of dimension.

*Page Blobs.* Page blobs are made of pages that are identified by an offset from the beginning of the blob. A page blob can be split in multiple pages or constituted by a single page. This type of blob is optimized for random access and can be used to host data different from streaming. At the time of writing, the maximum dimension of a page blob is 1 TB.

Blobs storage provides users with the ability to describe the data by adding metadata. It is also possible to take snapshots of a blob for backup purposes. Moreover, in order to optimize its distribution, blobs storage can leverage the Windows Azure CDN, so that blobs are kept close to those requesting them and can served more efficiently.

**(b) Azure Drive.** Page blobs can be used to store an entire file system in the form of a single *Virtual Hard Drive (VHD)* file. This can then be mounted as a part of the NTFS file system by Azure compute resources, thus providing persisting and durable storage. A page blob mounted as part of an NTFS tree is called an *Azure Drive*.

**(c) Tables.** Tables constitute a semi-structured storage solution, allowing users to store information in the form of entities having a collection of properties. Entities are stored as rows into the table and identified by a key, which also constitutes the unique index built for the table. Users can insert, update, delete, and select a subset of the rows stored in the table. Unlike SQL tables, there is no schema enforcing constraints on the properties of entities and there is no facility for representing relationships among entities. Because of this, tables are more similar to spreadsheets rather than SQL tables.

The service is designed to handle large amounts of data and query returning huge result sets. Two main features provide support in this sense: partial result sets and table partitions. A partial result set is returned together with a continuation token allowing the client to resume the query for large result sets. Table partitions allow tables to be divided among several servers for load balancing purposes. A partition is identified by a key, which is represented by three of the columns of the table.

At the time of writing, a table can contain up to 100 TB of data and rows can have up to 255 properties, with a maximum of 1 MB for each row. The maximum dimension of row keys and partition keys is 1 KB.

**(c) Queues.** Queue storage allows applications to communicate by exchanging messages through durable queues, thus preventing messages from getting lost or remaining unprocessed. Applications enter messages into a queue and other applications can read them in a FIFO style.

In order to ensure that messages get processed, when an application reads a message, the message is marked as invisible, thus being not available to other clients. Once the application has completed processing the message, it needs to explicitly delete it from the queue. This two-phase process ensures that messages get processed before they are removed from the queue, and that client failures do not prevent messages from being processed. At the same time, this is also a reason why the queue does not enforce a strict FIFO model: messages that are read by applications that crash during the processing are made available again after a time out, during which other messages can be read by other clients. An alternative to reading a message is peeking, which allow retrieving the message but letting it be visible in the queue. Messages that are peeked are not considered processed.

All the services described are geo-replicated three times in order to ensure their availability in case of major disasters. Geo-replication involves the copy of data into a different data center, which is hundred thousands of miles away from the original datacenter.

## *3. Core Infrastructure: AppFabric*

AppFabric is a comprehensive middleware for developing, deploying, and managing applications on the Cloud or for integrating existing applications with Cloud services. AppFabric implements an optimized infrastructure supporting scaling out and high availability; sandboxing and multi-tenancy; state management; and dynamic address resolution and routing. On top of this infrastructure, the middleware offers a collection of services that simplify many of the common tasks in a distributed application such as communication, authentication and authorization, and data access. These services are available through language agnostic interfaces, thus allowing developers to build heterogeneous applications.

**(a) Access Control.** AppFabric provides the capability of encoding access control to resources within Web applications and services into a set of rules, which are expressed outside the application code base. These rules give a great degree of flexibility in terms of the ability of securing components of the application and defining access control policies for users and groups.

Access control services also integrate several authentication providers into a single coherent identity management framework. Applications can leverage Active Directory, Windows Live, Google, Facebook and other services to authenticate users. This feature also allows easily building hybrid systems, which part existing in the private premises and part are deployed in the Cloud.

**(b) Service Bus.** Service Bus constitutes the messaging and connectivity infrastructure provided with AppFabric for building distributed and disconnected in the Azure Cloud and between the private premises and the Azure Cloud. Service Bus allows applications to interact with different protocols and patterns over a reliable communication channel that guarantees delivery.

The service is designed to allow transparent network traversal and to simplify the development of loosely coupled applications, without renouncing to security and reliability, and letting the developers focus on the logic of the interaction, rather than the details of its implementation. Service bus allows services to be available by simple URLs, which are untied to their deployment location. It is possible to support publish-subscribe models, full duplex communications point to point as well as in a peer-to-peer environment, unicast and multi-cast message delivery in one-way communications, and asynchronous messaging to decouple applications components.

Applications in order to leverage these features need to be connected to the bus, which provides these services. A connection is the element that is priced by Azure on a pay-as-you-go basis for service bus. Users are billed on connection/month basis and they can buy in advance “connection packs”, which have a discounted price, if they can estimate in advance their needs.

**(c) Azure Cache.** Windows Azure provides a set of durable storage solutions that allow applications to persist their data. These solutions are based on disk storage, which might constitute a bottleneck for applications that need to gracefully scale along the clients requests and dataset size dimensions.

*Azure Cache* is a service that allows developers to quickly access data persisted on the Windows Azure storage or in SQL Azure. The service implements a distributed in-memory cache whose size can be dynamically adjusted by applications according to their needs. It is possible to store any .NET managed object as well as many common data formats (table, rows, XML, and binary data) and control its access to it by applications. Azure Cache is delivered as a service and easily integrates with applications, since there is no need to implement or deploy specific components. This is particularly true for ASP.NET applications, which already integrate providers for session state and page output caching based on Azure Cache.

The service is priced according to the size of cache allocated by applications per month, despite their effective use of the cache. At the time of writing, several cache sizes are available, ranging from 128 MB ($45/month) to 4 GB ($325/month).

## *4. Other Services*

Compute, storage, and middleware services constitute the core components of the Windows Azure platform. Beside these, other services and components simplify the development and the integration of applications with the Azure Cloud. An important area for these services is applications connectivity, including virtual networking and content delivery.

**(a) Windows Azure Virtual Network.** Networking services for applications are offered under the name of *Windows Azure Virtual Network,* which includes: *Windows Azure Connect* and *Windows Azure Traffic Manager*.

Windows Azure Connect allows easily setting up IP-based network connectivity among machines hosted on the private premises and the roles deployed on the Azure Cloud. This service is particularly useful in case of VM roles, where machines hosted in the Azure Cloud become part of the private network of the enterprise and can be managed with the same tools used in the private premises.

Windows Azure Traffic Manager provides load balancing features for services listening the HTTP or HTTPS ports and hosted on multiple roles. It allows developers to choose from three different load-balancing strategies: Performance, Round-Robin, and Failover.

Currently, the two services are still in beta phase and available for free only by invitation.

**(b) Windows Azure Content Delivery Network (CDN).** *Windows Azure CDN* is the content delivery network solution that improves the content delivery capabilities of *Windows Azure Storage* and several other Microsoft services such as *Microsoft Windows Update* and *Bing* maps. The services allows to serve Web objects (images, static HTML, css, and scripts) as well as streaming content by using a network of 24 locations distributed across the world.

# 9.3.2 SQL Azure

SQL Azure is a relational database service hosted on Windows Azure and built on the SQL Server technologies. The service extends the capabilities of SQL Server to the Cloud and provides developers with a scalable, highly available, and fault-tolerant relational database. SQL Azure is accessible from either the Windows Azure Cloud or any other location that has access to the Azure Cloud. It is fully compatible with the interface exposed by SQL server, so that applications built for SQL server can transparently migrate to SQL Azure. Moreover, the service is fully manageable by using REST API, allowing developers to control databases deployed in the Azure Cloud as well as the firewall rules set up for their accessibility.

Fig. 9.4 shows the architecture of SQL Azure. Access to SQL Azure is based on the Tabular Data Stream (TDS) protocol, which is the communication protocol underlying all the different interfaces used by applications to connect to a SQL Server based installation such as ODBC and ADO.NET. On the SQL Azure side, access to data is mediated by the service layer, which provides provisioning, billing, and connection routing services. These services are logically part of server instances, which are managed by SQL Azure Fabric. This is the distributed database middleware that constitutes the infrastructure of SQL Azure and that is deployed on Microsoft data centers.

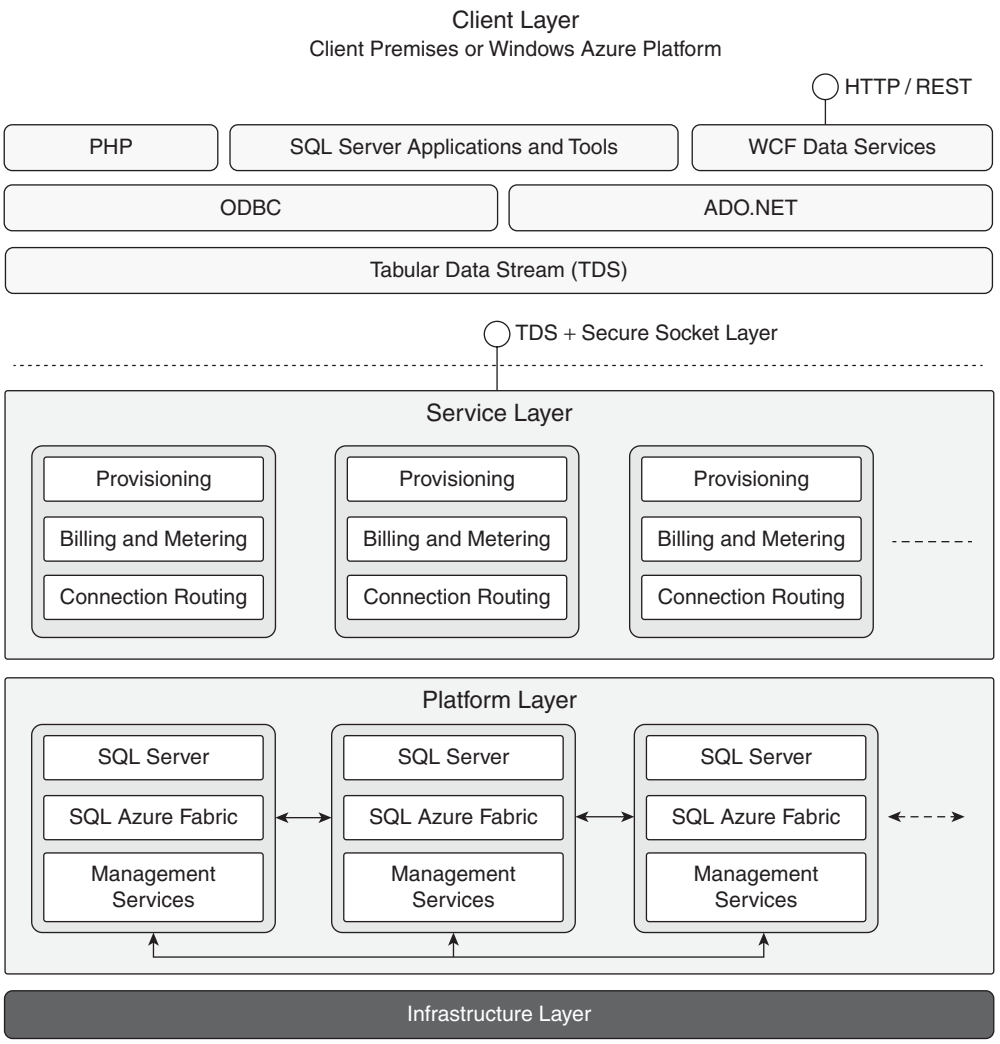


**Fig. 9.4.** SQL Azure Architecture.

Developers have to sign up for a Windows Azure account in order to use SQL Azure. Once the account is activated, they can either use the Windows Azure Management Portal or the REST APIs to create servers, logins, and configure access to servers. SQL Azure servers are abstractions that closely resemble physical SQL Servers: they have a fully qualified domain name under *database.windows.net* (i.e., *server-name.*

*database.windows.net*) domain name. This simplifies the management tasks and the interaction with SQL Azure from client applications. SQL Azure ensures that multiple copies of each server are maintained within the Azure Cloud and that these copies are kept synchronized when client applications insert, update, and delete data on them.

At the time of writing, the SQL Azure service is billed according on the space usage and the type of edition. Currently, two different editions are available: Web Edition and Business Edition. The former is suited for small Web applications and supports databases with a maximum size of 1 or 5 GB. The latter is suited for independent software vendors, line of business applications, and enterprise applications and supports databases with a maximum size from 10 GB to 50 GB in increments of 10 GB. Moreover, a bandwidth fee applies for any data transfer trespassing the Windows Azure Cloud or the region where the database is located. A monthly fee per user/database is also charged and it is based on the peak size reached by the database during the month.

## 9.3.3 Windows Azure Platform Appliance

The Windows Azure platform can also be deployed as an appliance on third-party data centers and constitute the Cloud infrastructure governing the physical servers of the data center. Windows Azure Platform Appliance includes *Windows Azure*, *SQL Azure*, and Microsoft-specified configuration of network, storage, and server hardware. The appliance is a solution that targets governments and service providers, which want to (or need to) their own Cloud computing infrastructure.

As introduced before, Azure already provides a development environment that allows building applications for Azure in their own premises. The local development environment is not intended to be a production middleware but it is designed for development and testing the functionalities of applications that will be eventually deployed on Azure. Azure appliance is instead a full-featured implementation of Windows Azure that includes hardware and system setups. Its goal is to replicate Azure on a third- party infrastructure and make available its services beyond the boundaries of the Microsoft's Cloud. The appliance addresses two major scenarios: institutions that have very large computing needs (such as government agencies) and those that cannot afford to transfer their data outside their premises; service providers who want to provide Cloud computing services but do not have platform to turn their data centers into a Cloud.

## 9.3.4 Summary

Windows Azure is the solution provided by Microsoft for developing Cloud computing applications. Azure is an implementation of the Platform-as-a-Service layer and provides developer with a collection of services and scalable middleware hosted on Microsoft data centers that address compute, storage, networking, and identity management needs of distributed applications. The services offered by Azure can be either used individually or all together for building both applications that integrate Cloud features and elastic computing systems completely hosted in the Cloud.

The core components of the platform are constituted by compute services, storage services, and the middleware. Compute services are based on the abstraction of roles, which identify a sandboxed environment where developers can build their distributed and scalable components. These roles are useful for Web applications, backend processing, and virtual computing. Storage services include solutions for static and dynamic content, which is organized in the form of tables, with fewer constraints than those imposed by the relational model. These and other services are implemented and made available through *AppFabric*, which constitute the distributed and scalable middleware of Azure.

SQL Azure is another important element of the Windows Azure and provides support for relational data in the Cloud. SQL Azure is an extension of the capabilities of SQL Server adapted for the Cloud environment and designed for dynamic scaling.

The platform is mostly based on the .NET technology and the Windows systems even though other technologies and systems can be supported. For this reason, Azure constitutes the solution of choice for migrating applications that are already based on the .NET technology to the Cloud.

## 9.4 OBSERVATIONS

In this chapter, we have introduced some of the Cloud platforms that are widely used in industry for building real commercial applications: Amazon Web Services, Google AppEngine, and Microsoft Windows Azure.

Amazon Web Services provides solutions for building infrastructure in the Cloud. Amazon EC2 and Amazon S3 represent its core value offering. The former allows developer to create virtual servers and customize at will their computing stack. The latter is a storage solution allowing users to store documents of any size. These core services are then complemented by a wide collection of services covering networking, data management, content distribution, computing middleware, and communication, which make AWS a complete solution for developing entire Cloud computing systems on top of the Amazon infrastructure.

Google AppEngine is a distributed and scalable platform for building Web applications in the Cloud. AppEngine is a scalable runtime that offers developers a collection of services for simplifying the development of Web applications. These services are designed with scalability in mind and constitute functional blocks that can be reused to define applications. Developers can build their applications either in Java or Python, first locally by using the AppEngine SDK. Once the applications have been completed and fully tested, they can deploy the application on AppEngine.

Windows Azure is the Cloud operating system that is deployed on Microsoft data centers for building dynamically scalable applications. The core components of Azure are represented by compute services expressed in term of roles, storage services, and the AppFabric, ie., the middleware that ties together all these services and constitutes the infrastructure of Azure. A role is a sandboxed runtime environment specialized for a specific development scenario: Web applications, background processing, and virtual computing. Developers define their Azure applications in terms of roles and then deploy these roles on Azure. Storage services represent a natural complement to roles. Besides, storage for static data and semi-structured data, Windows Azure also provides storage for relational data by means of the SQL Azure service.

AppEngine and Windows Azure are Platform-as-a-Service solutions, AWS extends its services across all the three layers of the Cloud computing reference model, although it well known for its Infrastructure-as-a-Service offering represented by EC2 and S3.

## Review Questions

1. What is AWS? What types of services does it provide?
2. Describe Amazon EC2 and its basic features.
3. What is a bucket? What type of storage does it provide?
4. What is the difference between Amazon SimpleDB and Amazon RDS?
5. What type of problems addresses Amazon Virtual Private Cloud?
6. Introduce and present the services provided by AWS to support connectivity among applications.
7. What is the Amazon CloudWatch?
8. What type of service is AppEngine?
9. Describe the core components of AppEngine.

10. What are the development technologies currently supported by AppEngine?
11. What is DataStore? What type of data can be stored in it?
12. Discuss the compute services offered by AppEngine.
13. What is Windows Azure?
14. Describe the architecture of Windows Azure.
15. What is a role? What type of roles can be used?
16. What is AppFabric and which services does it provide?
17. Discuss the storage services provided by Windows Azure.
18. What is SQL Azure?
19. Illustrate the architecture of SQL Azure.
20. What is Windows Azure Platform Appliance? For which kind of scenarios it has been designed?

# 10 Cloud Applications

Cloud computing has gained huge popularity in industry due to its ability to host applications whose services can be delivered to consumers rapidly at minimal cost. This chapter discusses various application case studies detailing their architecture and how they leveraged various Cloud technologies. Applications from a range of domains—from scientific to engineering, gaming, to social networking—are considered.

## 10.1 SCIENTIFIC APPLICATIONS

Scientific applications are a sector that is increasingly using Cloud computing systems and technologies. The immediate benefit seen by researchers and academics is the potentially infinite availability of computing resources and storage at sustainable prices if compared to a complete in-house deployment. Cloud computing systems meet the needs of different types of applications in the scientific domain: High Performance Computing (HPC) applications, High Throughput Computing (HTC) applications, and data-intensive applications. The opportunity for using Cloud resources is even more appealing since minimal changes need to be done to existing applications in order to leverage Cloud resources.

The most relevant option is *Infrastructure-as-a-Service* solutions, which offer the optimal environment for running bag-of-tasks applications and workflows. Virtual machine instances are opportunely customized to host the required software stack for running such applications and coordinated together by distributed computing middleware capable of interacting with Cloud-based infrastructures. *Platform-as-a-Service* solutions have been also considered. They allow scientists to explore new programming models for tackling computationally challenging problems. Applications have been redesigned and implemented on top of Cloud programming application models and platforms to leverage their unique capabilities. For instance, MapReduce programming model provides scientists with a very simple and effective model for building applications that need to process large datasets. Therefore, it has been widely used to develop data-intensive scientific applications. Problems that require a higher degree of flexibility in terms of structuring of their computation model can leverage platforms such as Aneka, which supports MapReduce and other programming models. We now discuss some interesting case studies in which Aneka has been used.

### 10.1.1 Healthcare: ECG Analysis in the Cloud

Healthcare is a domain where computer technology has found several and diverse applications: from supporting the business functions to assisting scientists in developing solutions to cure diseases. An

important application is the use of Cloud technologies for supporting doctors in providing more effective diagnostic processes. In particular, we discuss electrocardiogram (ECG) data analysis on the Cloud [160].

The capillary development of Internet connectivity and its accessibility from any device at any time has made Cloud technologies an attractive option for developing health-monitoring systems. Electrocardiogram (ECG) data analysis and monitoring constitutes a case study that naturally fits in this scenario. ECG is the electrical manifestation of the contractile activity of the heart's myocardium. This activity produces a specific waveform that is repeated overtime and that represents the heartbeat. The analysis of the shape of the waveform is used to identify arrhythmias, and it is the most common way for detecting heart diseases. Cloud computing technologies allow the remote monitoring of a patient's heartbeat data, its analysis in minimum time, and the notification of first-aid personnel and doctors should this data reveal potentially dangerous conditions. This way a patient at risk can be constantly monitored without going to hospital for ECG analysis. At the same time, doctors and first-aid personnel can instantly be notified with cases that require their attention.

An illustration of the infrastructure and model for supporting remote ECG monitoring is shown in Fig. 10.1. Wearable computing devices equipped with ECG sensors constantly monitor the patient's hearth-beat. Such information is transmitted to the patient's mobile device that will eventually forward it to the Cloud-hosted Web service for analysis. The Web service forms the front-end of a platform that is entirely hosted in the Cloud and that leverages the three layers of the Cloud computing stack: *SaaS*, *PaaS*, and *IaaS*. The Web service constitutes the SaaS application that will store ECG data into the Amazon S3 service and issues a processing request to the scalable Cloud platform. The runtime

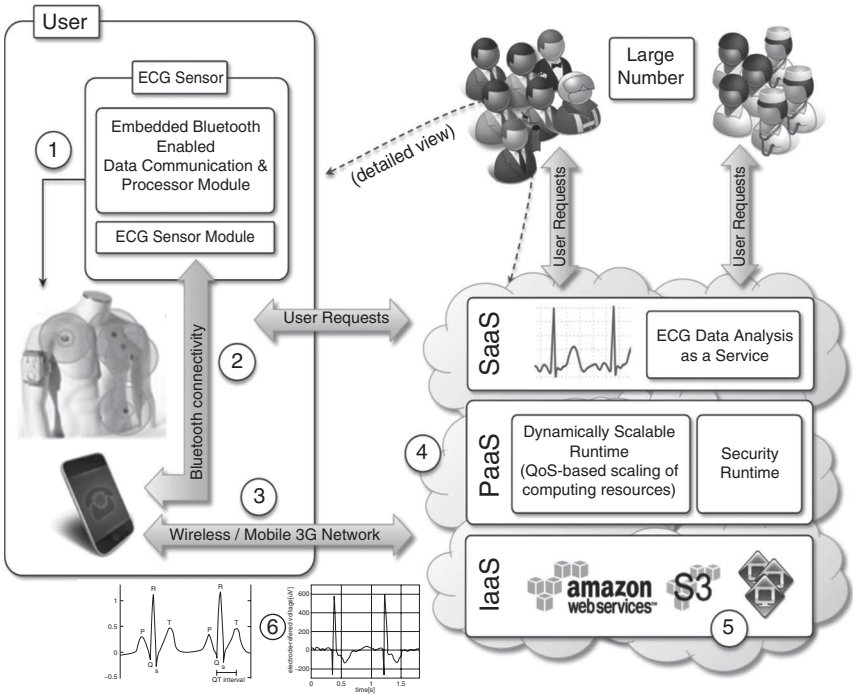


**Fig. 10.1.** Online Health-Monitoring System Hosted in the Cloud.

platform is composed by a dynamically sizable number of instances running the workflow engine and Aneka. The number of workflow engine instances is controlled according to the number of requests in

the queue of each instance, while Aneka controls the number EC2 instances used to execute the single tasks defined by the workflow engine for a single ECG processing job. Each of these jobs consists of a set of operations involving the extraction of the waveform from the heart-beat data and the comparison of the waveform with a reference waveform to detect anomalies. In case anomalies are found, doctors and first-aid personnel can be notified to act on a specific patient.

Even though remote ECG monitoring does not necessarily require Cloud technologies, Cloud computing introduces opportunities that would be otherwise hardly achievable. The first advantage is the elasticity of the Cloud infrastructure that can grow and shrink according to the requests served. As a result, doctors and hospitals do not have to invest in large computing infrastructures designed after capacity planning, thus making a more effective use of budgets. The second advantage is ubiquity. Cloud computing technologies have now become easily accessible, and promise to deliver systems with minimum or no downtime. Computing systems hosted in the Cloud are accessible from any Internet device through simple interfaces (such as SOAP and REST based Web services). This makes not only these systems ubiquitous but they can also be easily integrated with other systems maintained in the hospital's premises. Lastly, cost savings constitute another reason. Cloud services are priced on a pay-per-use basis and with volume prices in case of large numbers of service requests. These two models provide a set of flexible options that can be used to price the service, thus actually charging costs based on effective use rather than capital costs.

## 10.1.2 Biology: Protein-Structure Prediction

Applications in biology often require high computing capabilities and often operate on large datasets that cause extensive I/O operations. Because of these requirements, they have often made extensive use of supercomputing and cluster computing infrastructures. Similar capabilities can be leveraged ondemand by using Cloud computing technologies in a more dynamic fashion, thus opening new opportunities for bioinformatics applications.

Protein structure prediction is a computationally intensive task fundamental for different types of research in the life sciences. Among these is the design of new drugs for the treatment of diseases. The geometrical structure of a protein cannot be directly inferred from the sequence of genes that compose its structure, but it is the result of complex computations aimed at identifying the structure that minimizes the required energy. This task requires the investigation of a space with a massive number of states, and consequently creating a large number of computations for each of these states. The computational power required for protein structure prediction can now be acquired on demand, without owning a cluster or doing all the bureaucracy for getting access to parallel and distributed computing facilities. Cloud computing grants the access to such capacity on a pay-per-use basis.

A project that investigates the use of Cloud technologies for protein structure prediction is *Jeeva* [161]—an integrated Web portal that enables scientists to offload the prediction task to a computing Cloud based on Aneka. The prediction task uses machine learning techniques (support vector machines) for determining the secondary structure of proteins. These techniques translate the problem into a pattern recognition problem where a sequence has to be classified into one of the three possible classes (E, H, and C). A popular implementation, based on support vector machines, divides the pattern recognition problem into three phases: *initialization*, *classification*, and a *final phase*. Even though these three phases have to be executed in sequence, it is possible to take advantage of parallel execution in the classification phase where multiple classifiers are executed concurrently. This creates the opportunity of sensibly reducing the computational time of the prediction. The prediction algorithm is then translated into a task graph that is submitted to Aneka. Once completed, the middleware makes the results available for visualization through the portal.

The advantage of using Cloud technologies (i.e., Aneka as a scalable Cloud middleware) versus conventional grid infrastructures is the capability of leveraging a scalable computing infrastructure that can be grown and shrunk on demand. This concept is distinctive of Cloud technologies and constitutes a strategic advantage when applications are offered and delivered as a service.

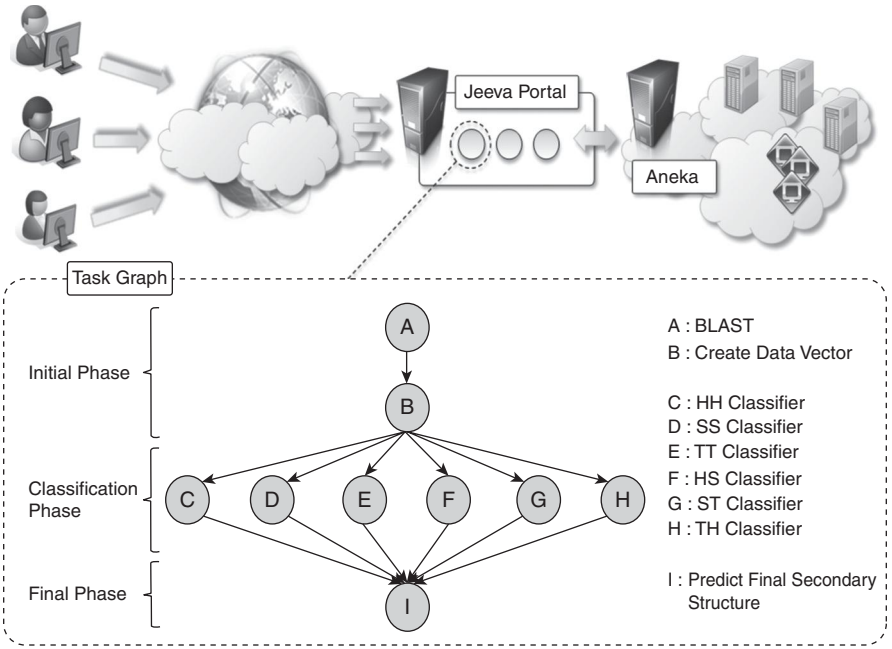


**Fig. 10.2.** Architecture and Overview of the Jeeva Portal.

## 10.1.3 Biology: Gene-Expression Data Analysis for Cancer Diagnosis

Gene expression profiling is the measurement of the expression levels of thousands of genes at once. It is used to understand the biological processes that are triggered by the treatment at a cellular level. Together with protein structure prediction, this activity is a fundamental component of drug design, since it allows scientists to identify the effects of a specific treatment.

Another important application of gene expression profiling is cancer diagnosis and treatment. Cancer is a disease characterized by uncontrolled cell growth and proliferation. This behavior occurs because of mutation of the genes that regulate the cell growth. This means that all the cancerous cells contain mutated genes. In this context, gene expression profiling is utilized to provide a more accurate classification of tumors. The classification of gene-expression data samples into distinct classes is a challenging task. The dimensionality of typical gene expression data sets ranges from several thousands to over ten thousands genes. However, only small sample sizes are typically available for analysis.

This problem is often approached with learning classifiers, which generate a population of condition-action rule that guide the classification process. Among these, the *eXtended Classifier System (XCS)* has been successfully utilized for classifying large datasets in the bioinformatics and computer science domains. However, the effectiveness of XCS when confronted with high-dimensional data sets (such as microarray gene expression data sets) has not been explored in detail. A variation of such algorithm, CoXCS [162], has proven to be effective in these conditions. CoXCS divides the entire search space into subdomains, and employs the standard XCS algorithm in each of these subdomains. Such a process is computationally intensive but can be easily parallelized as the classification problems on the subdomains can be solved concurrently. Cloud-CoXCS is a Cloud-based implementation of CoXCS that leverages Aneka to solve the classification problem in parallel and compose their outcomes. The algorithm is controlled by strategies, which define the way in which the outcomes are composed together and whether the process needs to be iterated.

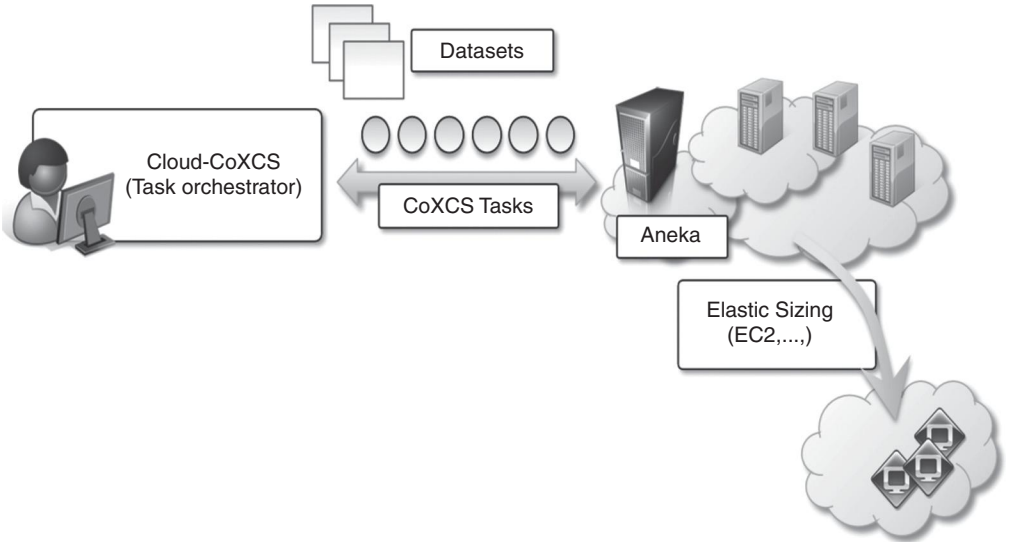


**Fig. 10.3** Cloud-CoXCS: An Environment for Microarray Data Processing on the Cloud.

Because of the dynamic nature of XCS, the number of required compute resources to execute it can vary over time. Therefore, the use of a scalable middleware such as Aneka offers a distinctive advantage.

## 10.1.4 Geoscience: Satellite Image Processing

Geoscience applications collect, produce, and analyse massive amounts of geospatial and non-spatial data. As the technology progresses and our planet becomes more instrumented (i.e., through the deployment of sensors and satellites for monitoring), the volume of data that need to be processed increases significantly. In particular, the Geographic Information System (GIS) is a major element of geoscience applications. GIS applications capture, store, manipulate, analyze, manage, and present all types of geographically referenced data. This type of information is now becoming increasingly relevant to a wide variety of application domains: from advanced farming to civil security and also natural resources management. As a result, a considerable amount of geo-referenced data is ingested into computer systems for further processing and analysis. Cloud computing is an attractive option for executing these demanding tasks and extracting meaningful information for supporting decision makers.

Satellite remote sensing generates hundreds of gigabytes of raw images that need to be further processed to become the basis of several different GIS products. This process requires both I/O and compute intensive tasks. Large size images need to be moved from the ground station's local storage to compute facilities where several transformations and corrections are applied. Cloud computing provides the appropriate infrastructure to support such application scenario. A Cloud-based implementation of such a workflow has been developed by the Department of Space, Government of India [163]. The system shown in Fig. 10.4 integrates several technologies across the entire computing stack. A SaaS application provides a collection of services for such as geocode generation and data visualization. At the PaaS level, Aneka controls the import of data into the virtualized infrastructure and the execution of image processing tasks that produce the desired outcome from raw satellite images. The platform leverages a Xen private Cloud and the Aneka technology to dynamically provision the required resources (i.e, grow or shrink) on demand.

The project demonstrates how Cloud computing technologies can be effectively employed to offload local computing facilities from excessive workloads and leverage more elastic computing infrastructures.

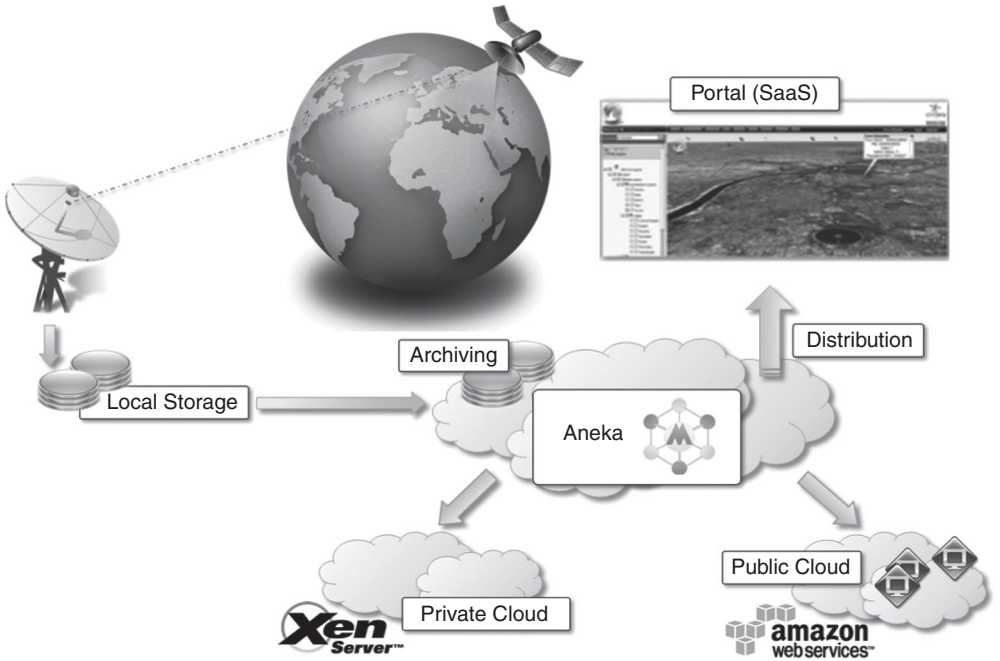


**Fig. 10.4.** A Cloud Environment for Satellite Data Processing.

# 10.2 BUSINESS AND CONSUMER APPLICATIONS

The business and consumer sector is the one that probably benefits the most from Cloud computing technologies. On the one hand, the opportunity of transforming capital cost into operational costs makes Clouds an attractive option for all enterprises that are IT centric. On the other hand, the sense of ubiquity that Cloud offers for accessing data and services makes it interesting for end users as well. Moreover, the elastic nature of Cloud technologies does not require huge upfront investments, thus allowing new ideas to be quickly translated into products and services that can comfortably grow with the demand. The combination of all these elements has made Cloud computing the preferred technology for a wide range of applications: from CRM and ERP systems to productivity and social networking applications.

## 10.2.1 CRM and ERP

*Customer Relationship Management (CRM)* and *Enterprise Resource Planning (ERP)* applications are market segments that are flourishing in the Cloud, with CRM applications being more mature than ERP implementations. Cloud CRM applications constitute a great opportunity for small enterprises and start-ups to have a fully functional CRM software without large upfront costs and by paying subscriptions. Moreover, customer relationship management is not an activity that requires specific needs and it can be easily moved to the Cloud. Such a characteristic, together with the possibility of having access to your business and customer data from everywhere and any device, has fostered the spread of Cloud CRM applications. ERP solutions on the Cloud are less mature and have to compete with well-established in-house solutions. ERP systems integrate several aspects of an enterprise: finance and accounting, human resources, manufacturing, supply chain management, project management, and customer relationship management. Their goal is to provide a uniform view and access to all operations that need to be performed to sustain a complex organization. Because of the organizations that they

target, the transition to a Cloud-based model is more difficult: the cost advantage over a long term might not be clear and the switch to the Cloud could not be easy, if organizations already have large ERP installations. For this reason, Cloud ERP solutions are less popular.

## 1. SalesForce.com

Salesforce.com is probably the most popular and developed CRM solution available today. As of today, more than 100 thousands customers have chosen Safesforce.com to implement their CRM solutions. The application provides customizable CRM solutions that can be integrated with additional features developed by third parties. Salesforce.com is based on the *Force.com* Cloud development platform. This represents the scalable and high-performance middleware executing all the operations of all Salesforce.com applications.

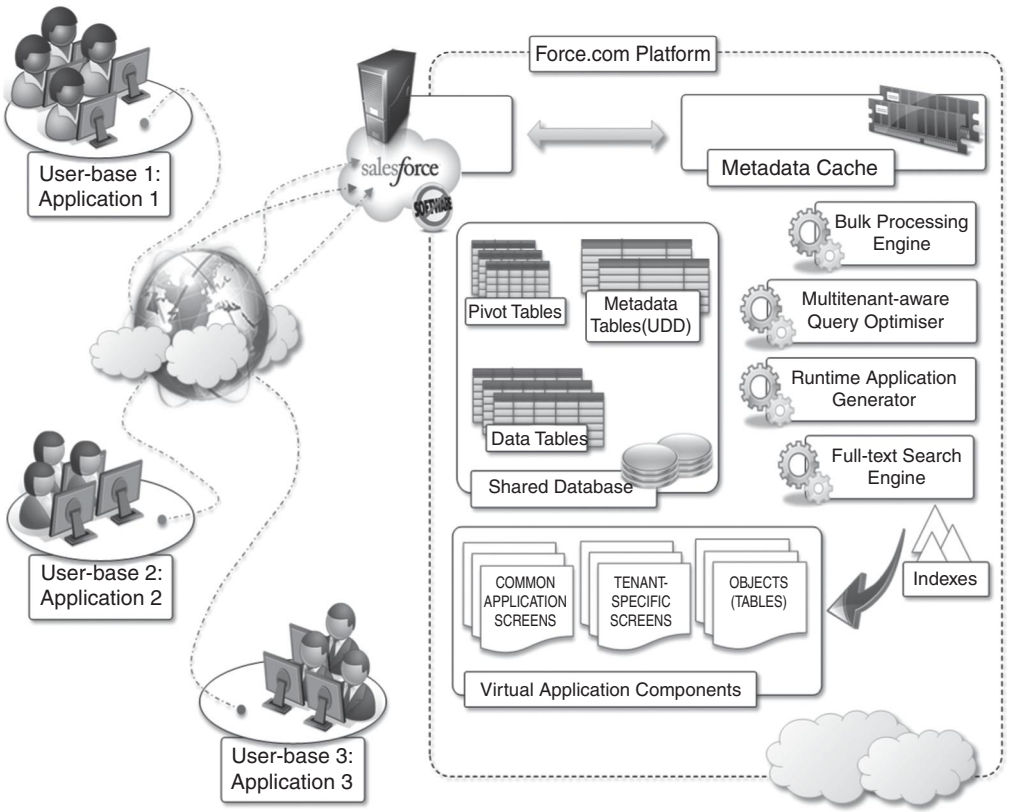


**Fig. 10.5.** Salesforce.com and Force.com Architecture.

The architecture of the Force.com platform is shown in Fig. 10.5. Initially designed to support scalable CRM applications, the platform has evolved to support the entire life cycle of a wider range of Cloud applications by implementing a flexible and scalable infrastructure. At the core of the platform resides its metadata architecture that provides the system with flexibility and scalability. Rather than being built on top of specific components and tables, application core logic and business rules are saved as metadata into the Force.com store. Both application structure and application data are stored into the store. A runtime engine executes application logic by retrieving its metadata and then performing the operations on the data. Although running in isolated containers, different applications logically share the same database structure and the runtime engine executes all of them uniformly. A full-text search engine

supports the runtime engine. This allows application users to have an effective user experience despite the large amount of data that needs to be crawled. The search engine maintains its indexing data in a separate store and gets constantly updated by background processes triggered by user interaction.

Users can customize their application by leveraging the "native" Force.com application framework or by using programmatic APIs in the most popular programming languages. The application framework allows users to visually define either the data or the core structure of a Force.com application, while the programmatic APIs provide them with a more conventional way for developing applications that relies on Web services to interact with the platform. Customization of application processes and logic can also be implemented by developing scripts in APEX. This is a Java-like language providing object-oriented and procedural capabilities for defining either scripts executed on demand or triggers. APEX also offers the capability of expressing searches and queries to have complete access to the data managed by the Force.com platform.

### *2. Microsoft Dynamics CRM*

Microsoft Dynamics CRM is the solution implemented by Microsoft for customer relationship management. Dynamics CRM is available either for installation on the enterprise's premises or as an online solution priced with a monthly per user subscription.

The system is completely hosted in Microsoft's data center across the world and offers to customers a 99.9% SLA, with bonus credits in case the system does not fulfill the agreement. Each CRM instance is deployed on a separate database, and the application provides users with facilities for marketing, sales, and advanced customer relationship management. Dynamics CRM Online features can be accessed either through a Web browser interface, or programmatically by means of SOAP and RESTful Web services. This allows Dynamics CRM to be easily integrated with both other Microsoft products and line of business applications. Dynamics CRM can be extended by developing plug-ins that allow implementing specific behaviors triggered on the occurrence of given events. Dynamics CRM can also leverage the capability of Windows Azure for the development and integration of new features.

### *3. NetSuite*

NetSuite provides a collection of applications that help customers manage every aspect of the business enterprise. Its offering is divided in three major products: *NetSuite Global ERP*, *NetSuite Global CRM+*, and *NetSuite Global Ecommerce*. Moreover, an all-in-one solution integrates all the three products together: *NetSuite One World*.

The services delivered by the company are powered by two large datacenters on the opposite coasts (east and west coasts) of the United States connected by redundant links. This allows NetSuite to guarantee 99.5% of uptime to its customers. Besides the pre-packaged solutions, NetSuite also provides an infrastructure and a development environment for implementing customized applications. The *NetSuite Business Operating System (NS-BOS)* is a complete stack of technologies for building Software-as-a-Service business applications that leverage the capabilities of NetSuite products. On top of the SaaS infrastructure, the NetSuite Business Suite components offer accounting, ERP, CRM, and e-commerce capabilities. An online development environment, *SuiteFlex*, allows integrating such capabilities into new Web applications, which are then packaged for distribution by *SuiteBundler*. The entire infrastructure is hosted in the NetSuite datacenters, which provide the warranties about the application uptime and availability.

## 10.2.2 Productivity

Productivity applications replicate in the Cloud some of the most common tasks that we perform on our desktop: from document storage, to office automation, and complete desktop environment hosted in the Cloud.

# 1. DropBox and iCloud

One of the core features of Cloud computing is to be available anywhere, at anytime, and from any Internet connected device. Therefore, document storage constitutes a natural application for such technology. Online storage solutions are precedent to Cloud computing, but they have never become popular. With the development of Cloud technologies, they have turn into Software-as-a-Service applications and become more usable as well as more advanced and accessible.

Perhaps the most popular solution for online document storage is *Dropbox*. This is an online application that allows you to synchronise any file across any platform and any device in a seamless manner (see Fig. 10.6). Dropbox provides users with a free amount of storage that is accessible through the abstraction of a folder. Users can either access their Dropbox folder through a browser or by downloading and installing a Dropbox client, which provides access to the online storage by means of a special folder. All the modifications into this folder are silently synched so that changes are notified to all the local instances of the Dropbox folder across all the devices. The key advantage of Dropbox is its availability on different platforms (Windows, Mac, Linux, and mobile), and the capability to work seamlessly and transparently across all of them.

Another interesting application in this area is *iCloud*. iCloud is a Cloud-based document sharing application provided by Apple to synchronise IOS-based devices in a completely transparent manner. Different from Dropbox, which provides synchronization through the abstraction of a local folder, iCloud has been designed to be completely transparent once it has been set up: documents, photos, and videos are automatically synched as changes are made without any explicit operation. This allows to efficiently automate common operations without any human intervention: taking a picture with an iPhone and having it automatically available in iPhoto on your Mac at home; editing a document in the iMac at home and having the changes updated in the iPad. Unfortunately, this capability is limited only to IOS devices, and currently there are no plans to provide iCloud with a Web-based interface that would make user's content accessible even from unsupported platforms.

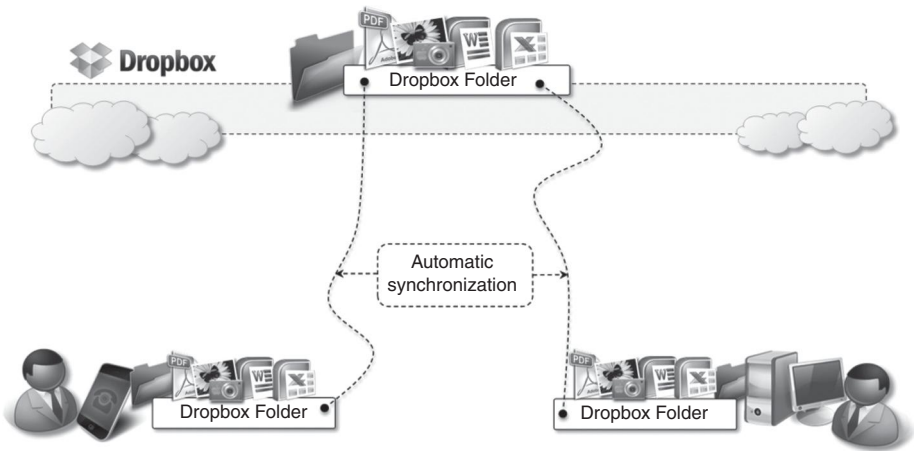


**Fig. 10.6.** Dropbox Usage Scenario.

There are other solutions for online document sharing that are popular and that we did not cover such as *Windows Live*, *Amazon Cloud Drive*, and *CloudMe*. These solutions offer more or less the same capabilities as those discussed above with different levels of integration between platforms and devices.

## 2. Google Docs

*Google Docs* is a Software-as-a-Service (SaaS) application that delivers the basic office automation capabilities with support for collaborative editing over the Web. The application is executed on top of Google distributed computing infrastructure that allows the system to dynamically scale according to the number of users using the service.

Google Docs allows creating and editing text documents, spreadsheets, presentations, forms, and drawings. It aims to substitute desktop products such as Microsoft Office and OpenOffice, and provide similar interface and functionality as a Cloud service. It supports collaborative editing over the Web for most of the applications included in the suite. This eliminates tedious mailing and synchronization tasks when documents need to be edited by multiple users. By being stored in the Google infrastructure, these documents are always available from anywhere and any device that is connected to the Internet. Moreover, the suite allows users to work off-line in case the Internet connectivity is not available. The support of various formats such as those that are produced by the most popular desktop office solutions allows a user to easily import and move documents in and out of Google Docs, thus eliminating barriers for the use of this application.

Google Docs is a good example of what Cloud computing can deliver to end users: ubiquitous access to resources, elasticity, absence of installation and maintenance costs, and delivery of core functionalities as a service.

## 3. Cloud Desktops: EyeOS and XIOS/3

Asynchronous Javascript and XML (AJAX) technologies have considerably augmented the capabilities that can be implemented in Web applications. This is a fundamental aspect for Cloud computing that delivers a considerable amount of its services through the Web browser. Together with the opportunity of leveraging large-scale storage and computation, this technology has made possible the replication of complex desktop environments in the Cloud and made them available through the Web browser. These applications—called *Cloud desktops*—are rapidly gaining popularity.

EyeOS[58] is one of the most popular Web desktop solutions based on Cloud technologies. It replicates the functionalities of classic desktop environment and comes with pre-installed applications for the most common file and document management (see Fig. 10.7). Single users can access the EyeOS desktop environment from anywhere and through any Internet connected device, while organisations can create a private EyeOS Cloud into their premises to virtualize the desktop environment of their employees and centralize their management.

The architecture of EyeOS is quite simple: on the server side, the EyeOS application maintains the information about user profiles and their data, and the client side constitutes the access point for users and administrators to interact with the system. EyeOS stores the data about users and applications onto the server file system. Once the user has logged in, by providing his/her credentials, the desktop environment is rendered in client's browser by downloading all the Javascript libraries required to build the user interface and implement the core functionalities of EyeOS. Each application loaded in the environment communicates with the server by using AJAX, and this communication model is used to access user's data as well as to perform application's operations: editing documents, visualizing images, copying and saving files, sending emails, and chatting.

EyeOS also provides API for developing new applications and integrating new capabilities into the system. EyeOS applications are server side components that are defined at least by two files (stored in the *eyeos*/*apps*/*appname* directory): *appname.php* and *appname.js*. The first file defines and implements all the operations that the application exposes while the Javascript file contains the code that needs to be loaded in the browser in order to provide user interaction with the application.

*Xcerion XML Internet OS/3 (XIOS/3)* is another example of a Web desktop environment. The service is delivered as part of the CloudMe application, which is a solution for Cloud document storage. The key

[58] ***http://www.eyeos.org***

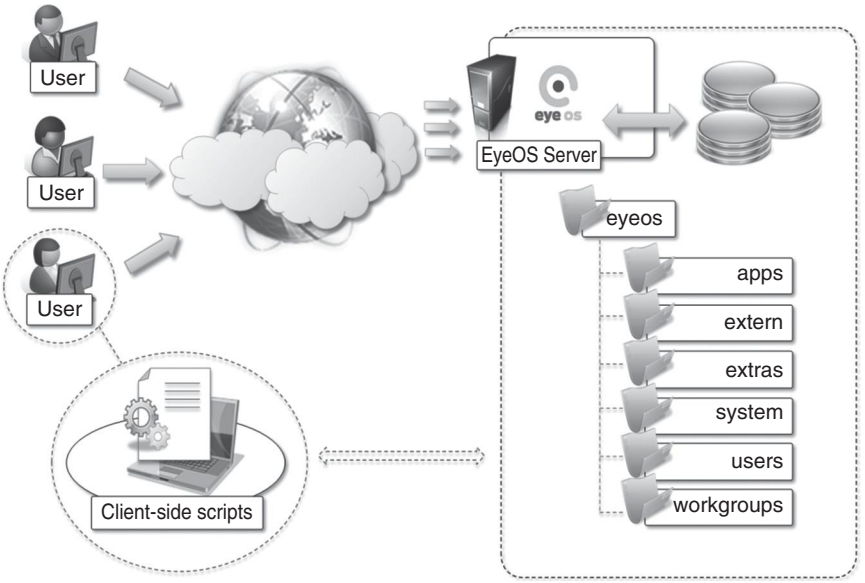


**Fig. 10.7.** EyeOS Architecture.

differentiator of XIOS/3 is its strong leverage on XML, used to implement many of the tasks of the OS: rendering user interfaces; defining application business logics; structuring file system organization; and even application development. The architecture of the OS concentrates most of the functionalities on the client side, while implementing server-based functionalities by means of XML Web services. The client side renders the user interface, orchestrates processes, and provides data binding capabilities on XML data that is exchanged with Web services. The server is responsible for implementing core functions such as transaction management for documents edited in a collaborative mode, and core logic of installed application into the environment. XIOS/3 also provides an environment for developing application (XIDE), which allows users to quickly develop complex applications by visual tools for the user interface and XML documents for business logic.

XIOS/3 is released as an open source software and implements a market place where third parties can easily deploy applications that can be installed on top of the virtual desktop environment. It is possible to develop any type of application and feed it with data accessible through XML Web services: developers have to define the user interface, bind UI components to service calls and operations, and provide the logic on how to process the data. XIDE will package this information into a proper set of XML documents and the rest will be performed by XML virtual machine implemented in XIOS.

XIOS/3 is an advanced Web desktop environment that focuses on the integration of services into the environment by means of XML-based services and simplifies collaboration with peers.

## 10.2.3 Social Networking

Social networking applications have considerably grown in the last years to become the most active sites on the Web. In order to sustain their traffic and to serve millions of users seamlessly, services like *Twitter* or *Facebook*, have leveraged Cloud computing technologies. The possibility of continuously adding capacity while systems are running is the most attractive feature for social networks, which constantly increase their user base.

### *Facebook*

Facebook is probably the most evident and interesting environment in social networking. It has become one of the largest Web sites in the world with more than 800 million users. In order to sustain this incredible growth, it has been fundamental to be capable of continuously adding capacity, developing new scalable technologies and software systems while keeping a high performance for a smooth user experience.

At the time of writing, the social network is backed by two data centers that have been built and optimized to reduce costs and impact on the environment. On top of this highly efficient infrastructure built and designed out of inexpensive hardware, a completely customized stack of open source technologies opportunely modified and refined constitutes the backend of the largest social network. Taken all together, these technologies constitute a powerful platform for developing Cloud applications. This platform primarily supports Facebook itself and offers APIs to integrate third-party applications with Facebook's core infrastructure to deliver additional services such as social games and quizzes created by others.

The reference stack serving Facebook is based on *Linux*, *Apache*, *MySQL*, and *PHP(*LAMP*)*. This collection of technologies is accompanied by a collection of other services developed in-house. These services are developed in a variety of languages and implement specific functionalities such as search, new feeds, notifications, and others. While serving page requests, the *social graph* of the user is composed. The social graph identifies collection of interlinked information that is of relevance for a given user. Most of the user data is served by querying a distributed cluster of MySQL instances, which mostly contain key-value pairs. This data is then cached for faster retrieval. The rest of the relevant information is then composed together by using the services mentioned before. These services are located closer to the data and developed in languages that provide a better performance than PHP.

The development of services is facilitated by a set of tools internally developed. One of the core elements is *Thrift*. This is a collection of abstractions (and language bindings) that allow cross-language development. Thrift allows services developed in different languages to communicate and exchange data. Bindings for Thrift in different languages take care of data serialization and deserialization, communication, and client and server boilerplate code. This simplifies the work of the developers that can quickly prototype services and leverage existing one. Other relevant services and tools are *Scribe*, which aggregates streaming log feeds, and applications for alerting and monitoring.

## 10.2.4 Media Applications

Media applications are a niche that has taken a considerable advantage from leveraging Cloud computing technologies. In particular, video processing operations, such as encoding, transcoding, composition, and rendering, are good candidates for a Cloud-based environment. These are computationally intensive tasks that can be easily offloaded to Cloud computing infrastructures.

### *1. Animoto*

Animoto[59] is perhaps the most popular example of media applications on the Cloud. The Website provides users with a very straightforward interface for quickly creating videos out of images, music, and video fragments submitted by users. Users select a specific theme for the video, upload the photos and videos and order them in the sequence they want to appear, select the song for the music, and render the video. The process is executed in the background and the user is notified via e-mail once the video is rendered.

The core value of Animoto is the ability to quickly create videos with stunning effects without the user intervention. A proprietary AI engine that selects the animation and transition effects according to pictures and music drives the rendering operation. Users only have to define the storyboard by organizing pictures and videos into the desired sequence. If not, the video can be rendered again and

[59] *http://www.animoto.com/*

the engine will select a different composition, thus producing a different outcome every time. The service allows creating 30 seconds videos for free. By paying a monthly or a yearly subscription, it is possible to produce videos of any length and to choose among a wider range of templates.

The infrastructure supporting Animoto is complex and is composed by different systems that all need to scale (see Fig. 10.8). The core function is implemented on top of the Amazon Web Services infrastructure. In particular, it uses Amazon EC2 for the Web front end and the worker nodes, Amazon S3 for the storage of pictures, music, and videos, and Amazon SQS for connecting all the components. The auto-scaling capabilities of the system are managed by Rightscale, which monitors the load and controls the creation of new worker instances as well as they reclaim. Front-end nodes collect the components required to make the video and store them into S3. Once the storyboard of the video is completed, a video rendering request is entered into a SQS queue. Worker nodes pick up rendering requests and perform the rendering. When the process is completed, another message is entered into a different SQS queue and another request is served. This last queue is cleared routinely and users get notified about the completion. The life of EC2 instances is controlled by Rightscale, which constantly monitors the load and the performance of the system, and decides whether it is necessary to grow or shrink.

The architecture of the system has proven to be very scalable and reliable by using up to 4000 servers on EC2 in peak times without dropping requests, but simply causing acceptable temporary delays for the rendering process.

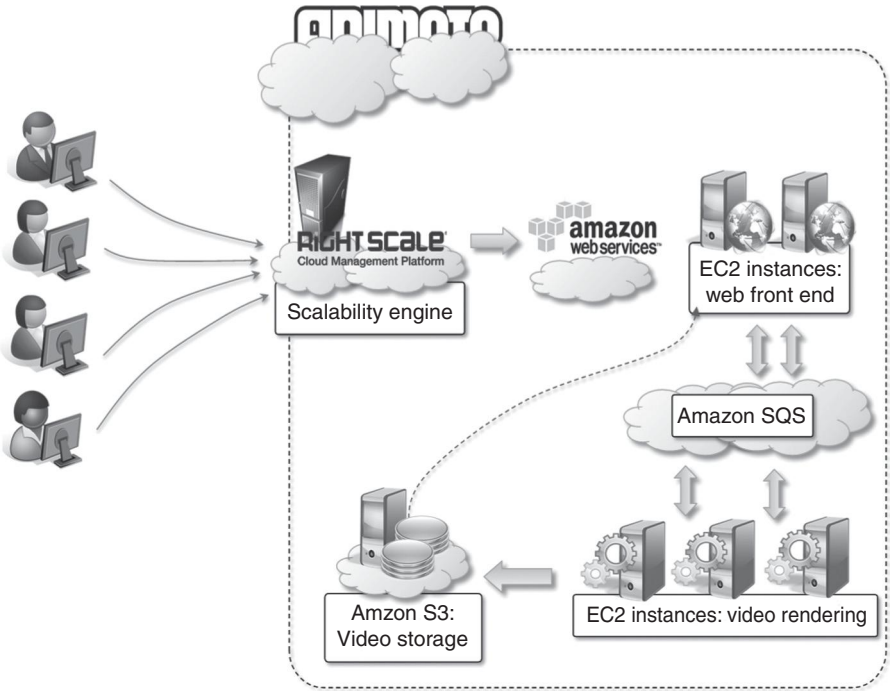


**Fig. 10.8.** Animoto Reference Architecture.

## 2. *Maya Rendering with Aneka*

Interesting applications of media processing are found in the engineering disciplines and the movie production industry. Operations such as rendering of models are now an integral part of the design workflow, which has become computationally demanding. The visualization of mechanical models is

not only used at the end of the design process, but it is iteratively used to improve the design. It is then fundamental to perform such task as fast as possible. Cloud computing provides engineers with the necessary computing power to make this happen.

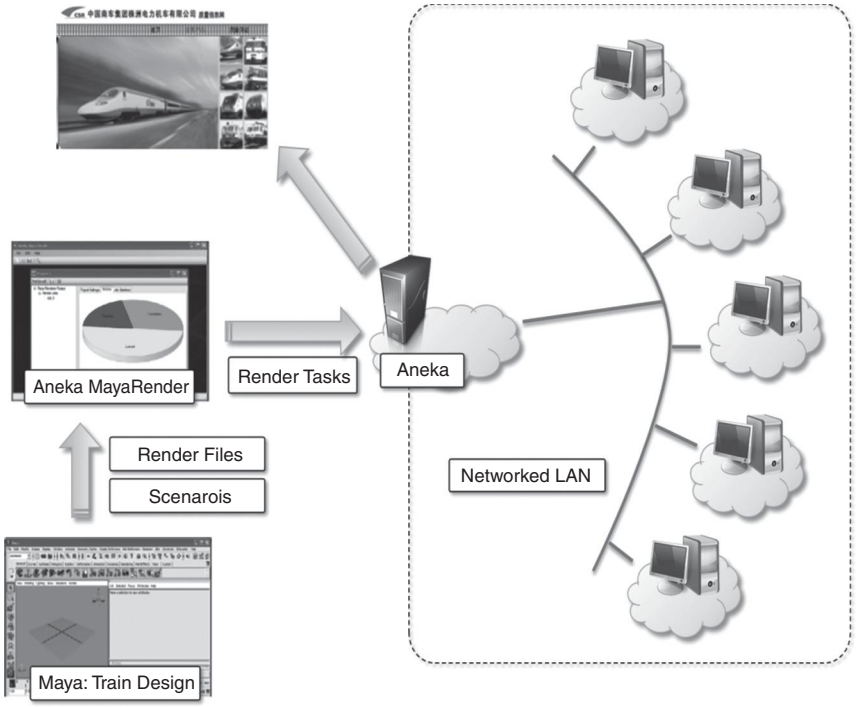


**Fig. 10.9.** 3D Rendering on Private Clouds.

A private Cloud solution for rendering train designs has been implemented by the engineering department of GoFront group—a division of China Southern Railway (see Fig.10.9). The department is responsible for designing models of high-speed electric locomotives, metro cars, urban transportation vehicles, and motor trains. The design process of prototypes requires high-quality 3D images. The analysis of these images can help engineers to identify problems and correct their design. It is critical for the department to reduce the time spent in these iterations, and 3D rendering tasks take considerable amount of time, especially in case of huge number of frames. This goal has been achieved by leveraging Cloud computing technologies, which turned the network of desktops of the departments into a desktop Cloud managed by Aneka. The implemented system includes a specialized client interface that can be used by GoFront engineers to enter all the details of the rendering process (the number of frames, the number of cameras, and other parameters). The application is used to submit the rendering tasks to the Aneka Cloud that distributes the load across all the available machines. Every rendering task triggers the execution of the local Maya batch renderer, and collects the result of the execution. The renders are then retrieved and put all together for visualisation.

By turning the local network into a private Cloud whose resources can be used off-peak (i.e., at night when desktops are not utilized), it has been possible for GoFront to sensibly reduce the time spent in the rendering process from days to hours.

### 3. *Video Encoding on the Cloud: Encoding.com*

Video encoding and transcoding are operations that can take a great benefit from using Cloud technologies: they are computationally intensive and potentially require considerable amount of storage. Moreover, with the continuous improvement of mobile devices as well as the diffusion of Internet, requests for video content have significantly increased. The variety of devices with video playback capabilities has led to an explosion of video formats through which a video can be delivered. Software and hardware for video encoding and transcoding often have prohibitive costs, or are not flexible enough to support conversion from any format to any format. Cloud technologies present an opportunity for turning these tedious and often demanding tasks into services that can be easily integrated into different workflows or made available to everyone according to their needs.

*Encoding.com* is software solution that offers video transcoding services on demand and leverages Cloud technology to provide both the horse-power required for video conversion and the storage for staging videos. The service integrates both with Amazon Web Services technologies (*EC2*, *S3*, and *CloudFront*) and Rackspace (*Cloud Servers*, *Cloud Files*, and *Limelight CDN* access). Users can access the services through a variety of interfaces: Encoding.com Website, Web service XML APIs, desktop applications, and watched folders. In order to use the service, users have to specify the location of the video to transcode, the destination format, and the target location of the video. Encoding.com also offers other video editing operations such the insertion of thumbnails, watermarks, or logos. Moreover, it also extends its capabilities to audio and image conversion.

The service provides different pricing options: monthly fee, pay-as-you-go (by batches), and special prices for high volumes. Encoding.com has up to now more than 2000 customers and has already processed more than 10 million videos.

## 10.2.5 Multiplayer Online Gaming

Online multiplayer gaming attracts millions of gamers around the world that share a common experience by playing together on a virtual environment that extends beyond the boundaries of a normal LAN. Online games support hundreds of players in the same session and this is made possible by the specific architecture used to forward interactions that is based on game log processing. Players update the game server hosting the game session, and the server integrates all the updates into a log that is made available to all the players through a TCP port. The client software used for the game connects to the log port and by reading the log updates, the local user can interface with the actions of other players.

Game log processing is also utilized to build statistics on players and rank them. These features constitute the additional value of online gaming portals that attract more and more gamers. The processing of game logs is a potentially compute intensive operation that strongly depends on the number of players online and the number of games monitored. Moreover, gaming portals are Web applications, and therefore, might suffer from the spiky behavior of users that can randomly generate large amount of volatile workloads that do not justify capacity planning.

The use of Cloud computing technologies can provide the required elasticity for seamlessly processing these workloads, and scale as required when the number of users increases. A prototypal implementation of Cloud-based game log processing has been implemented by Titan Inc. (now Xfire)—a company based in California that extended its gaming portal to offload game log processing to the Cloud by using Aneka. The prototype (shown in Fig. 10.10) has utilized a private Cloud deployment that has allowed Titan Inc. to process concurrently multiple logs and sustain a larger number of users.

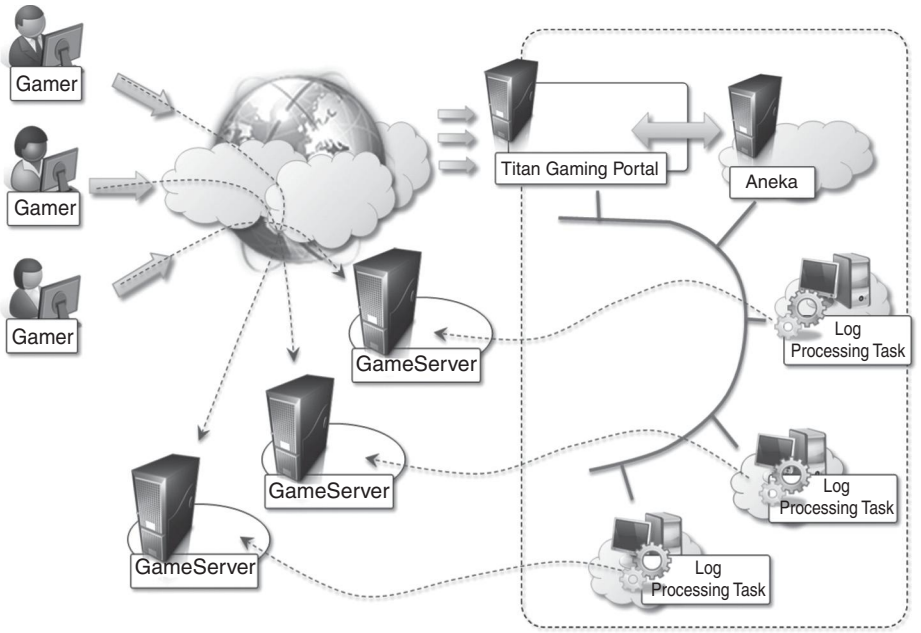


**Fig.10.10.** Scalable Processing of Logs for Network Games.

# Summary

In this chapter, we presented a brief overview of applications developed for the Cloud or that leverage Cloud technologies in some form. Different application domains can take advantage from Cloud computing: from scientific application to business and consumer applications.

Scientific applications take great benefit from the elastic scalability of Cloud environments that also provide the required degree of customization allowing the deployment and execution of scientific experiments. Business and consumer applications can leverage several other characteristics. CRM and ERP applications in the Cloud can reduce or even eliminate maintenance costs due to hardware management, system administration, and software upgrades. Moreover, they can also become ubiquitous and accessible from any device and anywhere. Productivity applications, such as office automation products, can make your document not only accessible but also modifiable from anywhere. This eliminates, for instance, the need of copying documents between devices. Media applications such as video encoding can offload lengthy and compute intensive encoding tasks onto the Cloud. Social networks can leverage the capability of continuously adding capacity without major service disruptions and by maintaining expected performance levels.

All these new opportunities have not only transformed the way in which we use these applications on a daily basis, but also introduced new challenges for developers who have to rethink their design to better benefit from elastic scalability, on demand resource provisioning, and ubiquity. These are key features of Cloud technology that make it an attractive solution in several domains.

# Review Questions

1. What are the types of applications that can benefit from Cloud computing?
2. What are the fundamental advantages brought by Cloud technology to scientific applications?
3. Describe how Cloud computing technology can be applied to support remote ECG monitoring.
4. Describe an application of Cloud computing technology in biology.
5. What are the advantages brought by Cloud computing in geoscience? Explain with an example.
6. Describe some examples of CRM and ERP implementations based on Cloud computing technologies.
7. What is Salesforce.com?
8. What are Dropbox and iCloud? Which kind of problems do they solve by using Cloud technologies?
9. Describe the key features of Google Apps.
10. What are Web desktops? What is their relation with Cloud computing?
11. What is the most important advantage of Cloud technologies for social networking applications?
12. Provide some examples of media applications that use Cloud technologies.
13. Describe an application of Cloud technologies for online gaming.