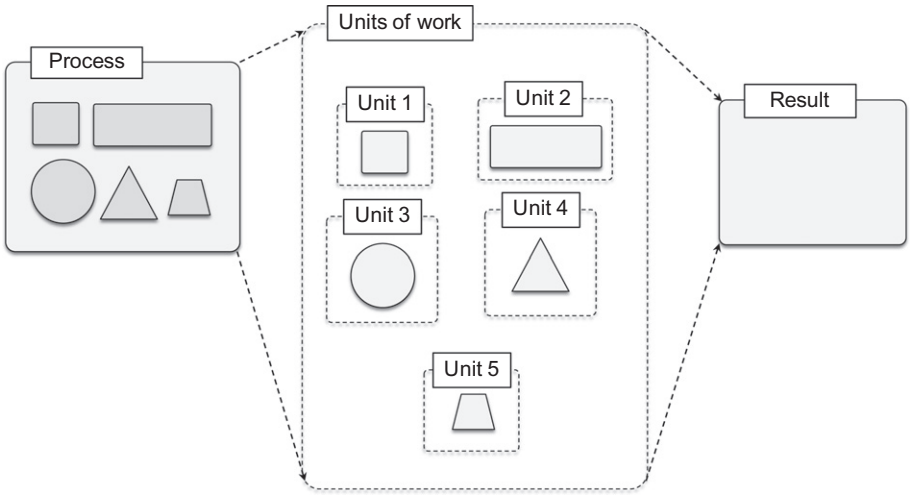

FIGURE 6.5

Functional decomposition.

defined by distinct logic operations. Figure 6.5 provides a pictorial view of how decomposition operates and allows parallelization.

As described by the schematic in Figure 6.5, problems that are subject to functional decomposition can also require a composition phase in which the outcomes of each of the independent units of work are composed together. In the case of domain decomposition, this phase often results in an aggregation process. The way in which results are composed in this case strongly depends on the type of operations that define the problem.

In the following, we show a very simple example of how a mathematical problem can be parallelized using functional decomposition. Suppose, for example, that we need to calculate the value of the following function for a given value of x:

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

It is apparent that, once the value of x has been set, the three different operations can be performed independently of each other. This is an example of functional decomposition because the entire problem can be separated into three distinct operations. A possible implementation of a parallel version of the computation is shown in Listing 6.3.

The program computes the *sine*, *cosine*, and *tangent* functions in three separate threads and then aggregates the results. The implementation provided constitutes an example of the alternative technique discussed in the previous sample program. Instead of using a data structure for keeping track of the worker threads that have been created, a function pointer is passed to each thread so that it can update the final result at the end of the computation. This technique introduces a synchronization problem that is properly handled with the *lock* statement in the method referenced by the function pointer. The *lock* statement creates a critical section that can only be accessed by one thread at time and guarantees that the final result is properly updated.

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

/// <summary>
/// Delegate UpdateResult. Function pointer that is used to update the final result
/// from the slave threads once the computation is completed.
/// </summary>
/// <param name="x">partial value to add.</param>
public delegate void UpdateResult(double x);

/// <summary>
/// Class Sine. Computes the sine of a given value.
/// </summary>
public class Sine
{
   /// <summary>
   /// Input value for which the sine function is computed.
   /// </summary>
   private double x;
   /// <summary>
   /// Gets the input value of the sine function.
   /// </summary>
   public double X { get { return this.x; } }
   /// <summary>
   /// Result value.
   /// </summary>
   private double y;
   /// <summary>
   /// Gets the result value of the sine function.
   /// </summary>
   public double Y { get { return this.y; } }
   /// <summary>
   /// Function pointer used to update the result.
   /// </summary>
   private UpdateResult updater;
   /// <summary>
   /// Creates an instance of the Sine and sets the input to the given angle.
   /// </summary>
   /// <param name="x">Angle in radiants.</param>
   /// <param name="updater">Function pointer used to update the result.</param>
   public Sine(double x, UpdateResultupdater)
   {
      this.x = x;
      this.updater = updater;
   }
   /// <summary>
   /// Executes the sine function.
   /// </summary>
   public void Apply()
   {
      this.y = Math.Sin(this.x);
      if (this.updater != null)
      {
         this.updater(this.y);
```

LISTING 6.3

Mathematical Function.

```
        }
    }
}

///<summary>
/// Class Cosine. Computes the cosine of a given value.
///</summary>
public class Cosine
{
    /// <summary>
    /// Input value for which the cosine function is computed.
    /// </summary>
    private double x;
    /// <summary>
    /// Gets the input value of the cosine function.
    /// </summary>
    public double X { get { return this.x; } }
    /// <summary>
    /// Result value.
    /// </summary>
    private double y;
    /// <summary>
    /// Gets the result value of the cosine function.
    /// </summary>
    public double Y { get { return this.y; } }
    /// <summary>
    /// Function pointer used to update the result.
    /// </summary>
    private UpdateResultupdater;
    /// <summary>
    /// Creates an instance of the Cosine and sets the input to the given angle.
    /// </summary>
    /// <param name="x">Angle in radiants.</param>
    /// <param name="updater">Function pointer used to update the result.</param>
    public Cosine(double x, UpdateResultupdater)
    {
        this.x = x;
        this.updater = updater;
    }
    /// <summary>
    /// Executes the cosine function.
    /// </summary>
    public void Apply()
    {
        this.y = Math.Cos(this.x);
        if (this.updater != null)
        {
            this.updater(this.y);
        }
    }
}

///<summary>
/// Class Tangent. Computes the tangent of a given value.
///</summary>
public class Tangent
{
```

LISTING 6.3

(Continued)

```
    /// <summary>
    /// Input value for which the tangent function is computed.
    /// </summary>
    private double x;
    /// <summary>
    /// Gets the input value of the tangent function.
    /// </summary>
    public double X { get { return this.x; } }
    /// <summary>
    /// Result value.
    /// </summary>
    private double y;
    /// <summary>
    /// Gets the result value of the tangent function.
    /// </summary>
    public double Y { get { return this.y; } }
    /// <summary>
    /// Function pointer used to update the result.
    /// </summary>
    private UpdateResultupdater;
    /// <summary>
    /// Creates an instance of the Tangent and sets the input to the given angle.
    /// </summary>
    /// <param name="x">Angle in radiants.</param>
    /// <param name="updater">Function pointer used to update the result.</param>
    public Tangent(double x, UpdateResultupdater)
    {
       this.x = x;
       this.updater = updater;
    }
    /// <summary>
    /// Executes the cosine function.
    /// </summary>
    public void Apply()
    {
       this.y = Math.Tan(this.x);
       if (this.updater != null)
       {
           this.updater(this.y);
       }
    }
}
/// <summary>
/// Class Program. Computes the function sin(x) + cos(x) + tan(x).
/// </summary>
public class Program
{
    /// <summary>
    /// Variable storing the coputed value for the function.
    /// </summary>
    private static double result;
    /// <summary>
    /// Synchronization instance used to avoid keeping track of the threads.
    /// </summary>
    private static object synchRoot = new object();
    /// <summary>
    /// Read the command line parameters and perform the scalar product.
```

LISTING 6.3

(Continued)

```
    /// </summary>
    /// <param name="args">Array strings containing the command line parameters.</param>
    public static void Main(string[] args)
    {
        // gets a value for x
        double x = 3.4d;

        // creates the function pointer to the update method.
        UpdateResult updater = newUpdateResult(Program.Sum);

        // creates the sine thread.
        Sine sine = newSine(x, updater);
        Thread tSine =new Thread(new ThreadStart(sine.Apply);

        // creates the cosine thread.
        Cosine cosine = newCosine(x, updater);
        Thread tCosine =new Thread(new ThreadStart(cosine.Apply);

        // creates the tangent thread.
        Tangent tangent = newTangent(x, updater);
        Thread tTangent =new Thread(new ThreadStart(tangent.Apply);

        // shuffles the execution order.
        tTangent.Start();
        tSine.Start();
        tCosine.Start();

        // waits for the completion of the threads.
        tCosine.Join();
        tTangent.Join();
        tSine.Join();

        // the result is available, dumps it to console.
        Console.WriteLine("f({0}): {1}", x, Program.result);
    }
    /// <summary>
    /// Callback that is executed once the computation in the thread is completed
    /// and adds the partial value passed as a parameter to the result.
    /// </summary>
    /// <param name="partial">Partial value to add.</param>
    private static voidSum(double partial)
    {
       lock(Program.synchRoot)
       {
          Program.result += partial;
       }
    }
}
```

LISTING 6.3

(Continued)

6.2.3.3 Computation vs. communication

In designing parallel and in general distributed applications, it is very important to carefully evaluate the communication patterns among the components that have been identified during problem decomposition. The two decomposition methods presented in this section and the corresponding sample applications are based on the assumption that the computations are *independent*. This means that:

- The input values required by one computation do not depend on the output values generated by another computation.
- The different units of work generated as a result of the decomposition do not need to interact (i.e., exchange data) with each other.

These two assumptions strongly simplify the implementation and allow achieving a high degree of parallelism and a high throughput. Having all the worker threads independent from each other gives the maximum freedom to the operating system (or the virtual runtime environment) scheduler in scheduling all the threads. The need to exchange data among different threads introduces dependencies among them and ultimately can result in introducing performance bottlenecks. For example, we did not introduce any queuing technique for threads; but queuing threads might potentially constitute a problem for the execution of the application if data need to be exchanged with some threads that are still in the queue. A more common disadvantage is the fact that while a thread exchanges data with another one, it uses some kind of synchronization strategy that might lead to blocking the execution of other threads. The more data that need to be exchanged, the more they block threads for synchronization, thus ultimately impacting the overall throughput.

As a general rule of thumb, it is important to minimize the amount of data that needs to be exchanged while implementing parallel and distributed applications. The lack of communication among different threads constitutes the condition leading to the highest throughput.

6.3 Multithreading with Aneka

As applications become increasingly complex, there is greater demand for computational power that can be delivered by a single multicore machine. Often this demand cannot be addressed with the computing capacity of a single machine. It is then necessary to leverage distributed infrastructures such as clouds. Decomposition techniques can be applied to partition a given application into several units of work that, rather than being executed as threads on a single node, can be submitted for execution by leveraging clouds.

Even though a distributed facility can dramatically increase the degree of parallelism of applications, its use comes with a cost in term of application design and performance. For example, since the different units of work are not executing within the same process space but on different nodes, both the code and the data need to be moved to a different execution context; the same happens for results that need to be collected remotely and brought back to the master process. Moreover, if there is any communication among the different workers, it is necessary to redesign the communication model eventually by leveraging the APIs, if any, provided by the middleware. In other words, the transition from a single-process multithreaded execution to a distributed execution is not transparent, and application redesign and reimplementation are often required.

The amount of effort required to convert an application often depends on the facilities offered by the middleware managing the distributed infrastructure. Aneka, as middleware for managing clusters, grids, and clouds, provides developers with advanced capabilities for implementing distributed applications. In particular, it takes traditional thread programming a step further. It lets you write multithreaded applications the traditional way, with the added twist that each of these threads can now be executed outside the parent process and on a separate machine. In reality, these "threads" are independent processes executing on different nodes and do not share memory or other resources, but they allow you to write applications using the same thread constructs for concurrency and synchronization as with traditional threads. Aneka threads, as they are called, let you easily port existing multithreaded compute-intensive applications to distributed versions that can run faster by utilizing multiple machines simultaneously, with minimum conversion effort.

6.3.1 Introducing the thread programming model

Aneka offers the capability of implementing multithreaded applications over the cloud by means of the *Thread Programming Model*. This model introduces the abstraction of distributed thread, also called *Aneka thread*, which mimics the behavior of local threads but executes over a distributed infrastructure. The Thread Programming Model has been designed to transparently port high-throughput multithreaded parallel applications over a distributed infrastructure and provides the best advantage in the case of embarrassingly parallel applications.

As described in Section 5.4.1, each application designed for Aneka is represented by a local object that interfaces to the middleware. According to the various programming models supported by the framework, such an interface exposes different capabilities, which are tailored to efficiently support the design and the implementation of applications by following a specific programming style. In the case of the Thread Programming Model, the application is designed as a collection of threads, the collective execution of which represents the application run. Threads are created and controlled by the application developer, while Aneka is in charge of scheduling their execution once they have been started. Threads are transparently moved and remotely executed while developers control them from local objects that act like proxies of the remote threads. This approach makes the transition from local multithreaded applications to distributed applications quite easy and seamless.

The Thread Programming Model exhibits APIs that mimic the ones exposed by .NET base class libraries for threading. In this way developers do not have to completely rewrite applications in order to leverage Aneka; the process of porting local multithreaded applications is as simple as replacing the *System.Threading.Thread* class and introducing the *AnekaApplication* class. There are three major elements that constitute the object model of applications based on the Thread Programming Model:

- *Application.* This class represents the interface to the Aneka middleware and constitutes a local view of a distributed application. In the Thread Programming Model the single units of work are created by the programmer. Therefore, the specific class used will be *Aneka.Entity.AnekaApplication* $< T,M >$, with T and M properly selected.
- *Threads.* Threads represent the main abstractions of the model and constitute the building blocks of the distributed application. Aneka provides the *Aneka.Threading.AnekaThread* class, which represents a distributed thread. This class exposes a subset of the methods exposed by the

System.Threading.Thread class, which has been reduced to those operations and properties that make sense or can be efficiently implemented in a distributed context.

- *Thread Manager.* This is an internal component that is used to keep track of the execution of distributed threads and provide feedback to the application. Aneka provides a specific version of the manager for this model, which is implemented in the *Aneka.Threading.ThreadManager* class.

As a result, porting local multithreaded applications to Aneka involves defining an instance of the *AnekaApplication* < *AnekaThread, ThreadManager* > class and replacing any occurrence of *System.Threading.Thread* with *Aneka.Threading.AnekaThread*. Developers can start creating threads, control their life cycles, and coordinate their execution similarly to local threads.

Aneka applications expose additional other properties, such as events that notify the completion of threads, their failure, the completion of the entire application, and thread state transitions. These operations are also available for the Thread Programming Model and constitute additional features that can be leveraged while porting local multithreaded applications, where this support needs to be explicitly programmed. Also, the *AnekaApplication* class provides support for files, which are automatically and transparently moved in the distributed environment.

6.3.2 Aneka thread vs. common threads

To efficiently run on a distributed infrastructure, Aneka threads have certain limitations compared to local threads. These limitations relate to the communication and synchronization strategies that are normally used in multithreaded applications.

6.3.2.1 Interface compatibility

The *Aneka.Threading.AnekaThread* class exposes almost the same interface as the *System.Threading.Thread* class with the exception of a few operations that are not supported. Table 6.1 compares the operations that are exposed by the two classes. The reference namespace that defines all the types referring to the support for threading is *Aneka.Threading* rather than *System.Threading*.

The basic control operations for local threads such as *Start* and *Abort* have a direct mapping, whereas operations that involve the temporary interruption of the thread execution have not been supported. The reasons for such a design decision are twofold. First, the use of the *Suspend/Resume* operations is generally a deprecated practice, even for local threads, since *Suspend* abruptly interrupts the execution state of the thread. Second, thread suspension in a distributed environment leads to an ineffective use of the infrastructure, where resources are shared among different tenants and applications. This is also the reason that the *Sleep* operation is not supported. Therefore, there is no need to support the *Interrupt* operation, which forcibly resumes the thread from a waiting or a sleeping state. To support synchronization among threads, a corresponding implementation of the *Join* operation has been provided.

Besides the basic thread control operations, the most relevant properties have been implemented, such as name, unique identifier, and state. Whereas the name can be freely assigned, the identifier is generated by Aneka, and it represents a globally unique identifier (GUID) in its string form rather than an integer. Properties such as *IsBackground*, *Priority*, and *IsThreadPoolThread* have been provided for interface compatibility but actually do not have any effect on thread scheduling and always expose the values reported in the table. Other properties concerning the state of the

Table 6.1 Thread API Comparison

.Net Threading API	Aneka Threading API
System.Threading	*Aneka.Threading*
Thread	*AnekaThread*
Thread.ManagedThreadId (int)	*AnekaThread.Id (string)*
Thread.Name	*AnekaThread.Name*
Thread.ThreadState (ThreadState)	*AnekaThread.State*
Thread.IsAlive	*AnekaThread.IsAlive*
Thread.IsRunning	*AnekaThread.IsRunning*
Thread.IsBackground	*AnekaThread.IsBackground[false]*
Thread.Priority	*AnekaThread.Priority[ThreadPriority.Normal]*
Thread.IsThreadPoolThread	*AnekaThread.IsThreadPoolThread [false]*
Thread.Start	*AnekaThread.Start*
Thread.Abort	*AnekaThread.Abort*
Thread.Sleep	[Not provided]
Thread.Interrupt	[Not provided]
Thread.Suspend	[Not provided]
Thread.Resume	[Not provided]
Thread.Join	*AnekaThread.Join*

thread, such as *IsAlive* and *IsRunning*, exhibit the expected behavior, whereas a slightly different behavior has been implemented for the *ThreadState* property that is mapped to the *State* property. The remaining methods of the *System.Threading.Thread* class (.NET 2.0) are not supported.

Finally, it is important to note differences in thread creation. Local threads implicitly belong to the hosting process and their range of action is limited by the process boundaries. To create local threads it is only necessary to provide a pointer to a method to execute in the form of the *ThreadStart* or *ParameterizedThreadStart* delegates. Aneka threads live in the context of a distributed application, and multiple distributed applications can be managed within a single process; for this reason, thread creation also requires the specification of the reference to the application to which the thread belongs.

Interface compatibility between Aneka threading APIs and the base class library allow quick porting of most of the local multithreaded applications to Aneka by simply replacing the class names and modifying the thread constructors.

6.3.2.2 Thread life cycle

Since Aneka threads live and execute in a distributed environment, their life cycle is necessarily different from the life cycle of local threads. For this reason, it is not possible to directly map the state values of a local thread to those exposed by Aneka threads. Figure 6.6 provides a comparative view of the two life cycles.

The white balloons in the figure indicate states that do not have a corresponding mapping on the other life cycle; the shaded balloons indicate the common states. Moreover, in local threads most of the state transitions are controlled by the developer, who actually triggers the state transition by invoking methods on the thread instance, whereas in Aneka threads, many of the state transitions are controlled

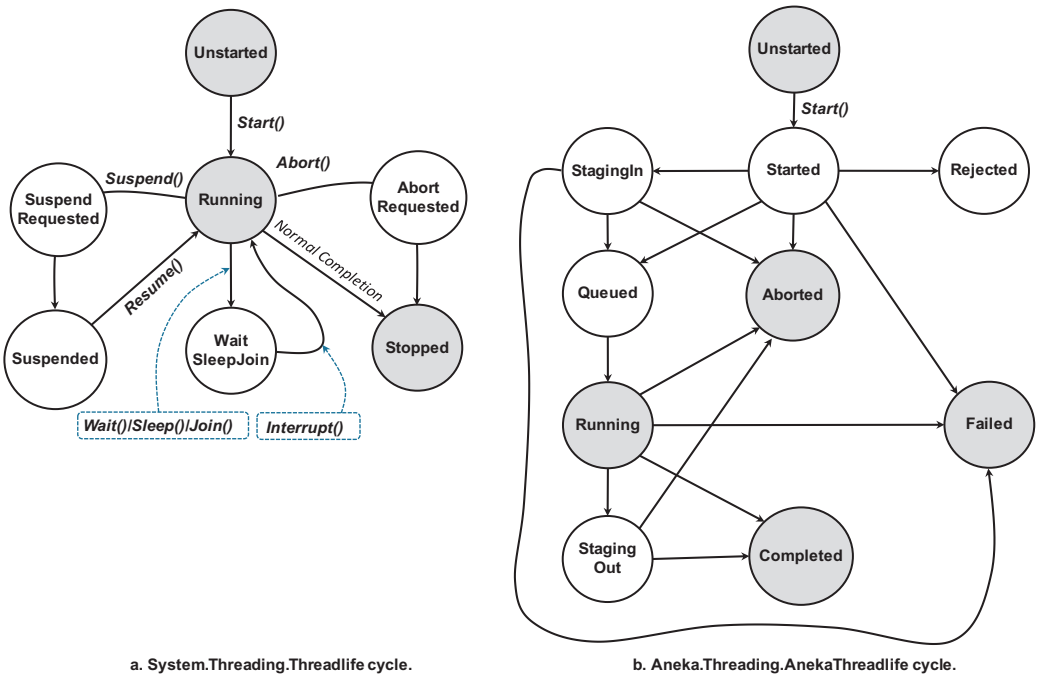

FIGURE 6.6

Thread life-cycle comparison.

by the middleware. As depicted in Figure 6.6, Aneka threads exhibit more states than local threads because Aneka threads support file staging and they are scheduled by the middleware, which can queue them for a considerable amount of time. As Aneka supports the reservation of nodes for execution of thread related to a specific application, an explicit state indicating execution failure due to missing reservation credential has been introduced. This occurs when a thread is sent to an execution node in a time window where only nodes with specific reservation credentials can be executed.

An Aneka thread is initially found in the *Unstarted* state. Once the *Start()* method is called, the thread transits to the *Started* state, from which it is possible to move to the *StagingIn* state if there are files to upload for its execution or directly to the *Queued* state. If there is any error while uploading files, the thread fails and it ends its execution with the *Failed* state, which can also be reached for any exception that occurred while invoking *Start()*.

Another outcome might be the *Rejected* state that occurs if the thread is started with an invalid reservation token. This is a final state and implies execution failure due to lack of rights. Once the thread is in the queue, if there is a free node where to execute it, the middleware moves all the object data and depending files to the remote node and starts its execution, thus changing the state into *Running*. If the thread generates an exception or does not produce the expected output files, the execution is considered failed and the final state of the thread is set to *Failed*. If the execution is successful, the final state is set to *Completed*. If there are output files to retrieve, the thread state is set to *StagingOut* while files are collected and sent to their final destination, and then it transits

to *Completed*. At any point, if the developer stops the execution of the application or directly calls the *Abort()* method, the thread is aborted and its final state is set to *Aborted*.

In most cases, the normal state transition will resemble the one occurring for local threads: *Unstarted → [Started] → [Queued] → Running → Completed/Aborted/Failed.*

6.3.2.3 Thread synchronization

The .NET base class libraries provide advanced facilities to support thread synchronization by the means of monitors, semaphores, reader-writer locks, and basic synchronization constructs at the language level. Aneka provides minimal support for thread synchronization that is limited to the implementation of the *join* operation for thread abstraction. Most of the constructs and classes that are provided by the .NET framework are used to provide controlled access to shared data from different threads in order to preserve their integrity. This requirement is less stringent in a distributed environment, where there is no shared memory among the thread instances and therefore it is not necessary. Moreover, the reason for porting a local multithread application to Aneka threads implicitly involves the need for a distributed facility in which to execute a large number of threads, which might not be executing all at the same time. Providing coordination facilities that introduce a locking strategy in such an environment might lead to distributed deadlocks that are hard to detect. Therefore, by design Aneka threads do not feature any synchronization facility that goes beyond the simple *join* operation between executing threads.

6.3.2.4 Thread priorities

The *System.Threading.Thread* class supports thread priorities, where the scheduling priority can be one selected from one of the values of the *ThreadPriority* enumeration: *Highest*, *AboveNormal*, *Normal*, *BelowNormal*, or *Lowest*. However, operating systems are not required to honor the priority of a thread, and the current version of Aneka does not support thread priorities. For interface compatibility purposes the *Aneka.Threading.Thread* class exhibits a *Priority* property whose type is *ThreadPriority*, but its value is always set to *Normal*, and changes to it do not produce any effect on thread scheduling by the Aneka middleware.

6.3.2.5 Type serialization

Aneka threads execute in a distributed environment in which the object code in the form of libraries and live instances information are moved over the network. This condition imposes some limitations that are mostly concerned with the serialization of types in the .NET framework.

Local threads execute all within the same address space and share memory; therefore, they do not need objects to be copied or transferred into a different address space. Aneka threads are distributed and execute on remote computing nodes, and this implies that the object code related to the method to be executed within a thread needs to be transferred over the network. Since delegates can point to instance methods, the state of the enclosing instance needs to be transferred and reconstructed on the remote execution environment. This is a particular feature at the class level and goes by the term *type serialization*.

A .NET type is considered *serializable* if it is possible to convert an instance of the type into a binary array containing all the information required to revert it to its original form or into a possibly different execution context. This property is generally given for several types defined in the .NET framework by simply tagging the class definition with the *Serializable* attribute. If the class

exposes a specific set of characteristics, the framework will automatically provide facilities to serialize and deserialize instances of that type. Alternatively, custom serialization can be implemented for any user-defined type.

Aneka threads execute methods defined in serializable types, since it is necessary to move the enclosing instance to remote execution method. In most cases, providing serialization is as easy as tagging the class definition with the *Serializable* attribute; in other cases it might be necessary to implement the *ISerializable* interface and provide appropriate constructors for the type. This is not a strong limitation, since there are very few cases in which types cannot be defined as serializable. For example, local threads, network connections, and streams are not serializable since they directly access local resources that cannot be implicitly moved onto a different node.

6.4 Programming applications with Aneka threads

To show how it is possible to quickly port multithreaded application to Aneka threads, we provide a distributed implementation of the previously discussed examples for local threads.

6.4.1 Aneka threads application model

The *Thread Programming Model* is a programming model in which the programmer creates the units of work as Aneka threads. Therefore, it is necessary to utilize the *AnekaApplication* $< W,M >$ class, which is the application reference class for all the programming models falling into this category. The Aneka APIs make strong use of generics and characterize the support given to different programming models through template specialization. Hence, to develop distributed applications with Aneka threads, it is necessary to specialize the template type as follows:

AnekaApplication $<$ *AnekaThread, ThreadManager* $>$

This will be the class type for all the distributed applications that use the Thread Programming Model. These two types are defined in the *Aneka.Threading* namespace noted in the *Aneka.Threading.dll* library of the Aneka SDK.

Another important component of the application model is the *Configuration* class, which is defined in the *Aneka.Entity* namespace (*Aneka.dll*). This class contains a set of properties that allow the application class to configure its interaction with the middleware, such as the address of the Aneka index service, which constitutes the main entry point of Aneka Clouds; the user credentials required to authenticate the application with the middleware; some additional tuning parameters; and an extended set of properties that might be used to convey additional information to the middleware. The code excerpt presented in Listing 6.4 demonstrates how to create a simple application instance and configure it to connect to an Aneka Cloud whose index service is local.

Once the application has been created, it is possible to create threads by specifying the reference to the application and the method to execute in each thread, and the management of the application execution is mostly concerned with controlling the execution of each thread instance. Listing 6.5 provides a very simple example of how to create Aneka threads.

The rest of the operations relate to the common management of thread instances, similar to local multithreaded applications discussed earlier.

```
// namespaces containing types of common use
using System;
using System.Collections.Generic;
// common Aneka namespaces.
using Aneka;
using Aneka.Util;
using Aneka.Entity;
// Aneka Thread Programming Model user classes
using Aneka.Threading;

// ……

///<summary>
///Creates an instance of the Aneka Application configured to use the
/// Thread Programming Model.
/// </summary>
/// <returns>Application instance.</returns>
private AnekaApplication<AnekaThread,ThreadManager> CreateApplication();
{
   Configuration conf =new Configuration();
   // this is the common address and port of a local installation
   // of the Aneka Cloud.
   conf.SchedulerUri = newUri("tcp://localhost:9090/Aneka");
   conf.Credentials =newUserCredentials("Administrator", string.Empty);
   // we will not need support for file transfer, hence we optimize the
   // application in order to not require any file transfer service.
   conf.UseFileTransfer = false;
   // we do not need any other configuration setting

   // we create the application instance and configure it.
   AnekaApplication<AnekaThread,ThreadManager> app =
                  new AnekaApplication<AnekaThread,ThreadManager>(conf);
   return app;
}
```

LISTING 6.4

Application Creation and Configuration.

6.4.2 Domain decomposition: matrix multiplication

To port to Aneka threads the multithreaded matrix multiplication, we need to apply the considerations made in the previous section. Hence, we start reviewing the code by first making the proper changes to the *ScalarProduct* class. Listing 6.6 shows the modified version of *ScalarProduct*.

The class has been tagged with the *Serializable* attribute and extended with the methods required to implement custom serialization. Supporting custom serialization implies the following:

- Including the *System.Runtime.Serialization* namespace.
- Implementing the *ISerializable* interface. This interface has only one method that is void *GetObjectData(SerializationInfo, StreamingContext)*, and it is called when the runtime needs to serialize the instance.
- Providing a constructor with the following signature: *ScalarProduct(SerializationInfo, StreamingContext)*. This constructor is invoked when the instance is deserialized.

```
// ..... continues from the previous listing
///<summary>
///Thread worker method (implementation skipped).
///</summary>
private void WorkerMethod()
{
   // .....
}
///<summary>
///Creates a collection of threads that are executed in the context of the
/// the given application.
///</summary>
/// <param name="app">>Application instance.</param>
private void CreateThreads(AnekaApplication<AnekaThread,ThreadManager> app);
{
    // creates a delegate to the method to execute inside the threads.
    ThreadStart worker = newThreadStart(this.WorkerMethod);
    // iterates over a loop and creates ten threads.
    for(int i=0; i<10; i++)
    {
        AnekaThread thread = new AnekaThread(worker, app);
        thread.Start();
    }
}
```

LISTING 6.5

Thread Creation and Execution.

The *SerializationInfo* class has a central role of providing a repository where all the properties defining the serialized format of a class can be stored and referenced by name. With minimum changes to the class layout, it would be possible to rely on the default serialization provided by the framework. To leverage such capability, it is necessary that all the properties defining the state of an instance are accessible through both *get* and *set* methods. In that case, it would be possible to simply tag the class as serialization, since all the fields constituting the state of the instance are also serializable. It can be noted that, apart from serialization, there is no need to make any change to the way the class operates.

The second step is to change the *MatrixProduct* class to leverage Aneka threads. We need to first create a properly configured application and then substitute the occurrences of the *System.Threading.Thread* class with *Aneka.Threading.Thread* (see Listing 6.7).

As shown in Listing 6.7, the changes that need to be applied to the logic of the program are minimal, and most of the modifications are related to exception management and the proper use of Aneka logging facilities. The *MatrixProduct* class integrates the method discussed in the previous section for application creation and setup and introduces a *try...catch...finally* block to handle exceptions that occurred while the application was executing. The rest of the code, except for renaming the occurrences of the *Thread* class, is unchanged.

There is only one important change to note: Once the Aneka thread instance is completed, the updated reference to the object containing the remotely executed method is exposed by the

```
using System.Runtime.Serialization;

///<summary>
/// Class ScalarProduct. Computes the scalar product between the row and the column
/// arrays. The class uses custom serialization. In order to do so it implements the
/// the ISerializable interface.
///</summary>
[Serializable]
public class ScalarProduct : ISerializable
{
   /// <summary>
   /// Scalar product.
   /// </summary>
   private double result;
   /// <summary>
   /// Gets the resulting scalar product.
   /// </summary>
   public double Result{get { returnthis.result; }}

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

   /// <summary>
   /// Creates an instance of the ScalarProduct class and configures it with the given
   /// row and column arrays.
   /// </summary>
   /// <param name="row">Array with the elements of the row to be multiplied.</param>
   /// <param name="column">Array with the elements of the column to be multiplied.
   /// </param>
   public ScalarProduct(double[] row, double[] column)
   {
      this.row = row;
      this.colum = colum;
   }
   /// <summary>
   /// Deserialization constructor used by the .NET runtime to recreate instances of
   /// of types implementing custom serialization.
   /// </summary>
   /// <param name="info">Bag containing the serialized data instance.</param>
   /// <param name="context">Serialization context (not used).</param>
   public ScalarProduct(SerializationInfo info, StreamingContext context)
   {
      this.result = info.GetDouble("result");
      this.row = info.GetValue("row", typeof(double[])) as double[];
      this.column = info.GetValue("column", typeof(double[])) as double[];
   }
```

LISTING 6.6

ScalarProduct Class (Modified Version).

```
        /// <summary>
        /// Executes the scalar product between the row and the colum.
        /// </summary>
        /// <param name="row">Array with the elements of the row to be multiplied.</param>
        /// <param name="column">Array with the elements of the column to be multiplied.
        /// </param>
        public void Multiply()
        {
            this.result = 0;
            for(int i=0; i<this.row.Length; i++)
            {
                this.result += this.row[i] * this.column[i];
            }
        }
        /// <summary>
        /// Serialization method used by the .NET runtime to serialize instances of
        /// of types implementing custom serialization.
        /// </summary>
        /// <param name="info">Bag containing the serialized data instance.</param>
        /// <param name="context">Serialization context (not used).</param>
        public ScalarProduct(SerializationInfo info, StreamingContext context)
        {
            this.result = info.GetDouble("result");
            this.row = info.GetValue("row", typeof(double[])) as double[];
            this.column = info.GetValue("column", typeof(double[])) as double[];
        }
        /// <summary>
        /// Executes the scalar product between the row and the colum.
        /// </summary>
        /// <param name="row">Array with the elements of the row to be multiplied.</param>
        /// <param name="column">Array with the elements of the column to be multiplied.
        /// </param>
        public void Multiply()
        {
            this.result = 0;
            for(int i=0; i<this.row.Length; i++)
            {
                this.result += this.row[i] * this.column[i];
            }
        }
        /// <summary>
        /// Serialization method used by the .NET runtime to serialize instances of
        /// of types implementing custom serialization.
        /// </summary>
        /// <param name="info">Bag containing the serialized data instance.</param>
        /// <param name="context">Serialization context (not used).</param>
        public void GetObjectData(SerializationInfo info, StreamingContext context)
        {
            info.AddValue("result",this.result);
            info.AddValue("row", this.row,typeof(double[]));
            info.AddValue("column", this.column,typeof(double[]));
        }
    }
```

LISTING 6.6

(Continued)

```
using System;
// we do not anymore need the reference to the threading namespace.
// using System.Threading;
using System.Collections.Generic;

// reference to the Aneka namespaces of interest.
// common Aneka namespaces.
using Aneka;
using Aneka.Util;
using Aneka.Entity;
// Aneka Thread Programming Model user classes
using Aneka.Threading;

/// <summary>
/// Class MatrixProduct. Performs the matrix product of two matrices.
/// </summary>
public class MatrixProduct
{
   /// <summary>
   /// First and second matrix of the produt.
   /// </summary>
   private static double[,]a, b;
   /// <summary>
   /// Result matrix.
   /// </summary>
   private static double[,] c;
   ///<summary>
   /// Dictionary mapping the thread instances to the corresponding ScalarProduct
   ///instances that are run inside.The occurrence of the Thread class has been
   ///substituted with AnekaThread.
   ///</summary>
   private static IDictionary<AnekaThread, ScalarProduct> workers.
   /// <summary>
   /// Reference to the distributed application the threads belong to.
   /// </summary>
   private static AnekaApplication<AnekaThread, ThreadManager> app;

   /// <summary>
   /// Read the command line parameters and perform the scalar product.
   /// </summary>
   /// <param name="args">Array strings containing the command line parameters.</param>
   public static void Main(string[] args)
   {
      try
      {
         // activates the logging facility.
         Logger.Start();

         // creates the Aneka application instance.
         MatrixProduct.app =Program.CreateApplication();

         // reads the input matrices a and b.
         MatrixProduct.ReadMatrices();
         // executes the parallel matrix product.
         MatrixProduct.ExecuteProudct();
         // waits for all the threads to complete and
         // composes the final matrix.
```

LISTING 6.7

MatrixProduct Class (Modified Version).

```
            MatrixProduct.ComposeResult();
        }
        catch(Exception ex)
        {
            IOUtil.DumpErrorReport(ex, "Matrix Multiplication - Error executing " +
                                       "the application");
        }
        finally
        {
            try
            {
                // checks whether the application instance has been created
                // stops it.
                if (MatrixProduct.app != null)
                {
                    MatrixProduct.app.Stop();
                }
            }
            catch(Exception ex)
            {
                IOUtil.DumpErrorReport(ex, "Matrix Multiplication - Error stopping " +
                                           "the application");
            }
            // stops the logging thread.
            Logger.Stop();
        }
    }

    /// <summary>
    /// Executes the parallel matrix product by decomposing the problem in
    /// independent scalar product between rows and colums.
    /// </summary>
    private static void ExecuteThreads()
    {
        // we replace the Thread class with AnekaThread.
        MatrixProduct.workers = new Dictionary<AnekaThread, ScalarProduct>();
        int rows = MatrixProduct.a.Length;
        // in .NET matrices are arrays of arrays and the number of columns is
        // is represented by the length of the second array.
        int columns = MatrixProduct.b[0].Length;

        for(int i=0; i<rows; i++)
        for(int j=0; j<columns; j++)
        {
            double[] row = MatrixProduct.a[i];
            // beacause matrices are stored as arrays of arrays in order to
            // to get the columns we need to traverse the array and copy the
            // the data to another array.
            double[] column = new double[common];
            for(int k=0; k<common; k++)
            {
                column[j] = MatrixProduct.b[j][i];
            }
            // creates a ScalarProduct instance with the previous rows and
            // columns and starts a thread executing the Multiply method.
            ScalarProduct scalar = newScalarProduct(row, column);
            // we change the System.Threading.Thread class with the corresponding
```

LISTING 6.7

(Continued)

```
            // Aneka.Threading.AnekaThread class and reference the application instance.
            AnekaThread worker = newAnekaThread(newThreadStart(scalar.Multiply), app);
            worker.Name =string.Format("{0}.{1}",row,column);
            worker.Start();
            // adds the thread to the dictionary so that it can be
            // further retrieved.
            MatrixProduct.workers.Add(worker, scalar);
        }
    }
    /// <summary>
    /// Waits for the completion of all the threads and composes the final
    /// result matrix.
    /// </summary>
    private static void ComposeResult()
    {
        MatrixProduct.c = new double[rows,columns];
        // we replace the Thread class with AnekaThread.
        foreach(KeyValuePair<AnekaThread,ScalarProduct>pair in MatrixProduct.workers)
        {
            AnekaThread worker = pair.Key;
            // we have saved the coordinates of each scalar product in the name
            // of the thread now we get them back by parsing the name.
            string[] indices = string.Split(worker.Name, new char[] {'.'});
            int i = int.Parse(indices[0]);
            int j = int.Parse(indices[1]);
            // we wait for the thread to complete
            worker.Join();
            // instead of using the local value of the ScalarProduct instance
            // we use the one that has is stored in the Target property.
            // MatrixProduct.c[i,j] = pair.Value.Result;
            MatrixProduct.c[i,j] = ((ScalarProduct) worker.Target).Result;
        }
        MatrixProduct.PrintMatrix(MatrixProduct.c);
    }
    /// <summary>
    /// Reads the matrices.
    /// </summary>
    private static void ReadMatrices()
    {
        // code for reading the matrices a and b
    }
    /// <summary>
    /// Prints the given matrix.
    /// </summary>
    /// <param name="matrix">Matrix to print.</param>
    private static voidPrintMatrices(double[,] matrix)
    {
        // code for printing the matrix.
    }
    /// <summary>
    /// Creates an instance of the Aneka Application configured to use the
    /// Thread Programming Model.
    /// </summary>
    /// <returns>Application instance.</returns>
    private AnekaApplication<AnekaThread,ThreadManager> CreateApplication();
    {
```

LISTING 6.7

(Continued)

```
            Configuration conf =new Configuration();
            // this is the common address and port of a local installation
            // of the Aneka Cloud.
            conf.SchedulerUri = newUri("tcp://localhost:9090/Aneka");
            conf.Credentials =newUserCredentials("Administrator", string.Empty);
            // we will not need support for file transfer, hence we optimize the
            // application in order to not require any file transfer service.
            conf.UseFileTransfer = false;
            // we do not need any other configuration setting

            // we create the application instance and configure it.
            AnekaApplication<AnekaThread,ThreadManager> app =
                        new AnekaApplication<AnekaThread,ThreadManager>(conf);
            return app;
        }
    }
```

LISTING 6.7

(Continued)

AnekaThread.Target property and not the local variable referencing the object that was initially used to create the delegate.

6.4.3 Functional decomposition: *Sine*, *Cosine*, and *Tangent*

The modifications required to port this sample to Aneka threads are basically the same as those discussed in the previous example. There is only one significant difference in this case: Each of the threads has a reference to a delegate that is used to update the global sum at the end of the computation. Since we are operating in a distributed environment, the instance on which the object will operate is not shared among the threads, but each thread instance has its own local copy. This prevents the global sum from being updated in the master thread and requires a change in the update strategy utilized.

This example also illustrates how to modify the classes *Sine*, *Cosine*, and *Tangent* so that they can leverage the default serialization capabilities of the framework (see Listing 6.8).

This example demonstrated how to change the logic of the application in case the worker methods executed in the threads have a reference to a local object that is updated as a consequence of the execution. To allow the execution of such applications with Aneka threads, it is necessary to extrapolate the update logic from the worker method of the threads and perform it into the master thread.

SUMMARY

This chapter provided a brief overview of multithreaded programming and the technologies used for multiprocessing on a single machine. We introduced the basics of multicore technology, which is the latest technological advancement for achieving parallelism on a single computer, and discussed how such parallelism can be leveraged to speed up applications by using multithreaded programming. A thread defines a single control flow within a process, which is the logical unit for

```
using System;
// we do not anymore need the reference to the threading namespace.
// using System.Threading;
using System.Collections.Generic;

// reference to the Aneka namespaces of interest.
// common Aneka namespaces.
using Aneka;
using Aneka.Util;
using Aneka.Entity;
// Aneka Thread Programming Model user classes
using Aneka.Threading;


// this is not needed anymore.
// /// <summary>
// /// Delegate UpdateResult. Function pointer that is used to update the final result
// /// from the slave threads once the computation is completed.
// /// </summary>
// /// <param name="x">partial value to add.</param>
// public delegate void UpdateResult(double x);

/// <summary>
/// Class Sine. Computes the sine of a given value.
/// </summary>
[Serializable]
public class Sine
{
   /// <summary>
   /// Input value for which the sine function is computed.
   /// </summary>
   private double x;
   /// <summary>
   /// Gets or sets the input value of the sine function.
   /// </summary>
   public double X { get { return this.x; } set { this.x = value; } }
   /// <summary>
   /// Result value.
   /// </summary>
   private double y;
   /// <summary>
   /// Gets or sets the result value of the sine function.
   /// </summary>
   public double Y { get { return this.y; } set { this.y = value; } }
   // we can't use this anymore.
   // /// <summary>
   // /// Function pointer used to update the result.
   // /// </summary>
   // private UpdateResult updater;

   // we need a default constructor, which is automatically provided by the compiler
   // if we do not specify any constructor.
   // /// <summary>
   // /// Creates an instance of the Sine and sets the input to the given angle.
   // /// </summary>
   // /// <param name="x">Angle in radiants.</param>
   // /// <param name="updater">Function pointer used to update the result.</param>
```

LISTING 6.8

Mathematical Function (Modified Version).

```
    // public Sine(double x, UpdateResult updater)
    // {
    //    this.x = x;
    //     this.updater = updater;
    // }
    /// <summary>
    /// Executes the sine function.
    /// </summary>
    public void Apply()
    {
        this.y = Math.Sin(this.x);
        // we cannot use this anymore because there is no
        // shared memory space.
        // if (this.updater != null)
        // {
        //     this.updater(this.y);
        // }
    }
}

///<summary>
/// Class Cosine. Computes the cosine of a given value. The same changes have been
/// applied by removing the code not needed anymore rather than commenting it out.
///</summary>
[Serializable]
public class Cosine
{
    /// <summary>
    /// Input value for which the cosine function is computed.
    /// </summary>
    private double x;
    /// <summary>
    /// Gets or sets the input value of the cosine function.
    /// </summary>
    public double X { get { return this.x; } set { this.x = value; } }
    /// <summary>
    /// Result value.
    /// </summary>
    private double y;
    /// <summary>
    /// Gets or sets the result value of the cosine function.
    /// </summary>
    public double Y { get { return this.y; } set { this.y = value; } }
    /// <summary>
    /// Executes the cosine function.
    /// </summary>
    public voidApply()
    {
        this.y = Math.Cos(this.x);
    }
}

///<summary>
/// Class Cosine. Computes the cosine of a given value.  The same changes have been
/// applied by removing the code not needed anymore  rather than commenting it out.
///</summary>
[Serializable]
```

LISTING 6.8

(Continued)

```
public class Tangent
{
   /// <summary>
   /// Input value for which the tangent function is computed.
   /// </summary>
   private double x;
   /// <summary>
   /// Gets or sets the input value of the tangent function.
   /// </summary>
   public double X { get { return this.x; } set { this.x = value; } }
   /// <summary>
   /// Result value.
   /// </summary>
   private double y;
   /// <summary>
   /// Gets or sets the result value of the tangent function.
   /// </summary>
   public double Y { get { return this.y; } set { this.y = value; } }
   /// <summary>
   /// Executes the tangent function.
   /// </summary>
   public voidApply()
   {
      this.y = Math.Cos(this.x);
   }
}

///<summary>
/// Class Cosine. Computes the cosine of a given value. The same changes have been
/// applied by removing the code not needed anymore  rather than commenting it out.
///</summary>
[Serializable]
public class Tangent
{
   /// <summary>
   /// Input value for which the tangent function is computed.
   /// </summary>
   private double x;
   /// <summary>
   /// Gets or sets the input value of the tangent function.
   /// </summary>
   public double X { get { return this.x; } set { this.x = value; } }
   /// <summary>
   /// Result value.
   /// </summary>
   private double y;
   /// <summary>
   /// Gets or sets the result value of the tangent function.
   /// </summary>
   public double Y { get { return this.y; } set { this.y = value; } }
   /// <summary>
   /// Executes the tangent function.
   /// </summary>
   public voidApply()
   {
      this.y = Math.Tan(this.x);
   }
```

LISTING 6.8

(Continued)

```
}

/// <summary>
/// Class Program. Computes the function sin(x) + cos(x) + tan(x).
/// </summary>
public class Program
{
   /// <summary>
   /// Variable storing the coputed value for the function.
   /// </summary>
   private static double result;

   // we do not need synchronization anymore, because the update of the global
   // sum is done sequentially.
   // /// <summary>
   // /// Synchronization instance used to avoid keeping track of the threads.
   // /// </summary>
   // private static object synchRoot = new object();

   /// <summary>
   /// Reference to the distributed application the threads belong to .
   /// </summary>
   private static AnekaApplication<AnekaThread, ThreadManager> app;

   /// <summary>
   /// Read the command line parameters and perform the scalar product.
   /// </summary>
   /// <param name="args">Array strings containing the command line parameters. </param>
   public static voidMain(string[] args)
   {
      try
      {
         // activates the logging facility.
         Logger.Start();
         // creates the Aneka application instance.
            app = Program.CreateApplication();

         // gets a value for x
            double x = 3.4d;

            // creates the function pointer to the update method.
         UpdateResult updater = newUpdateResult(Program.Sum);

         // creates the sine thread.
            Sine sine = newSine(x, updater);
         AnekaThread tSine =newAnekaThread(new ThreadStart(sine.Apply), app);

         // creates the cosine thread.
         Cosine cosine = newCosine(x, updater);
         AnekaThread tCosine =newAnekaThread(new ThreadStart(cosine.Apply), app);

         // creates the tangent thread.
         Tangent tangent = newTangent(x, updater);
         AnekaThread tTangent =newAnekaThread(new ThreadStart(tangent.Apply), app);

         // shuffles the execution order.
            tTangent.Start();
```

LISTING 6.8

(Continued)

```
            tSine.Start();
            tCosine.Start();

            // waits for the completion of the  threads.
            tCosine.Join();
            tTangent.Join();
            tSine.Join();

            // once we have joined all the threads the values have been collected back
            // and we use the Target property in order to obtain the object with the
            // updated values.
            sine = (Sine) tSine.Target;
            cosine = (Cosine) tSine.Target;
            tangent = (Tangent) tSine.Target;

            Program.result = sine.Target.Y + cosine.Y + tangent.Y;

            // the result is available, dumps it to  console.
            Console.WriteLine("f({0}): {1}", x, Program.result);
        }
        catch(Exception ex)
        {
            IOUtil.DumpErrorReport(ex,  "Math Functions - Error executing " +
                                        "the application");
        }
        finally
        {
            try
            {
                // checks whether the application instance has been created
                // stops it.
                if (app != null)
                {
                    app.Stop();
                }
            }
            catch(Exception ex)
            {
                IOUtil.DumpErrorReport(ex,  "Math Functions - Error stopping " +
                                            "the application");
            }
            // stops the logging thread.
            Logger.Stop();
        }
    }
    // we do not need this anymore.
    // /// <summary>
    // /// Callback that is executed once the computation in the thread is completed
    // /// and adds the partial value passed as a parameter to the result.
    // /// </summary>
    // /// <param name="partial">Partial value to add.</param>
    // private static void Sum(double partial)
    // {
    //     lock(Program.synchRoot)
    //     {
    //         Program.result += partial;
    //     }
```

LISTING 6.8

(Continued)

```
        // }
        /// <summary>
        /// Creates an instance of the Aneka Application configured to use the
        /// Thread Programming Model.
        /// </summary>
        /// <returns>Application instance.</returns>
        private AnekaApplication<AnekaThread,ThreadManager> CreateApplication();
        {
            Configuration conf =new Configuration();
            // this is the common address and port of a local installation
            // of the Aneka Cloud.
            conf.SchedulerUri = newUri("tcp://localhost:9090/Aneka");
            conf.Credentials =newUserCredentials("Administrator", string.Empty);
            // we will not need support for file transfer, hence we optimize the
            // application in order to not require any file transfer service.
            conf.UseFileTransfer = false;
            // we do not need any other configuration setting

            // we create the application instance and configure it.
            AnekaApplication<AnekaThread,ThreadManager> app =
                                new AnekaApplication<AnekaThread,ThreadManager>(conf);
            return app;
         }
    }
```

LISTING 6.8

(Continued)

representing a running program in modern operating systems. Currently, all the most popular operating systems support multithreading, irrespective of whether the underlying hardware explicitly supports real parallelism or not. Real parallelism is supported by the use of multiple processors or cores at the same time, if they are available; otherwise, multithreading is obtained by interleaving the execution of multiple threads on the same processing unit.

To support multithreaded programming, programming languages define the abstraction of process and thread in their class libraries. A popular standard for operations on threads and thread synchronization is POSIX, which is supported by all the Linux/UNIX operating systems and is available as an additional library for the Windows operating systems family. A common implementation of POSIX is given in C/C++ as a library of functions. New-generation languages such as Java and C# (.NET) provide a set of abstractions for thread management and synchronization that is compliant and that most closely follows the object-oriented design that characterizes these languages. These implementations are portable over any operating system that provides an implementation for the runtime environment required by these languages.

Multithreaded programming is a practice that allows achieving parallelism within the boundaries of a single machine. Applications requiring a high degree of parallelism cannot be supported by normal multithreaded programming and must rely on distributed infrastructures such as clusters, grids, or, most recently, clouds. The use of these facilities imposes application redesign and the use of specific APIs, which might require significant changes to the existing applications. To address this issue, Aneka provides the Thread Programming Model, which extends the philosophy behind multithreaded programming beyond the boundaries of a single node and allows leveraging

heterogeneous distributed infrastructure for execution. To minimize application reconversion, the Thread Programming Model mimics the API of the *System.Threading* namespace, with some limitations that are imposed by the fact that threads are executed on a distributed infrastructure. High-throughput applications can be easily ported to Aneka threads with minimal or no changes at all to their logic. Examples of such features and the basic steps of converting a local multithreaded application to Aneka threads were given in the chapter by discussing simple applications demonstrating the methodology of domain and functional decomposition for parallel problems.

As a framework for distributed programming, Aneka provides many built-in features that are not generally of use while architecting an application in terms of concurrent threads. These are, for example, event notification and support for file transfer. These capabilities are available as core features of the Aneka application model but have not been demonstrated in the case of the Thread Programming Model, which is concerned with providing support for partitioning the execution of algorithms to speed up execution. However, they are indeed of great use in the case of "bag of tasks" applications, discussed in the next chapter.

Review questions

1. What is throughput computing and what does it aim to achieve?
2. What is multiprocessing? Describe the different techniques for implementing multiprocessing.
3. What is multicore technology and how does it relate to multiprocessing?
4. Briefly describe the architecture of a multicore system.
5. What is multitasking?
6. What is multithreading and how does it relate to multitasking?
7. Describe the relationship between a process and a thread.
8. Does parallelism of applications depend on parallel hardware architectures?
9. Describe the principal characteristics of a thread from a programming point of view and the uses of threads for parallelizing application execution.
10. What is POSIX?
11. Describe the support given for programming with threads in new-generation languages such as Java or C#.
12. What do the terms *logical thread* and *physical thread* refer to?
13. What are the common operations implemented for a thread?
14. Describe the two major techniques used to define a parallel implementation of computer algorithms.
15. What is an *embarrassingly parallel* problem?
16. Describe how to implement a parallel matrix scalar product by using domain decomposition.
17. How does communication impact design and the implementation of parallel or distributed algorithms?
18. Which kind of support does Aneka provide for multithreading?
19. Describe the major differences between Aneka threads and local threads.
20. What are the limitations of the Thread Programming Model?
21. Design a parallel implementation for the tabulation of the Gaussian function by using simple threads and then convert it to Aneka threads.

CHAPTER

High-Throughput Computing
Task Programming

7

Task computing is a wide area of distributed system programming encompassing several different models of architecting distributed applications, which, eventually, are based on the same fundamental abstraction: the *task*. A task generally represents a program, which might require input files and produce output files as a result of its execution. Applications are then constituted of a collection of tasks. These are submitted for execution and their output data are collected at the end of their execution. The way tasks are generated, the order in which they are executed, or whether they need to exchange data differentiate the application models that fall under the umbrella of task programming.

This chapter characterizes the abstraction of a task and provides a brief overview of the distributed application models that are based on the task abstraction. The Aneka Task Programming Model is taken as a reference implementation to illustrate the execution of *bag-of-tasks (BoT)* applications on a distributed infrastructure.

7.1 Task computing

Organizing an application in terms of tasks is the most intuitive and common practice for developing parallel and distributed computing applications. A task identifies one or more operations that produce a distinct output and that can be isolated as a single logical unit. In practice, a task is represented as a distinct unit of code, or a *program*, that can be separated and executed in a remote runtime environment. Programs are the most common option for representing tasks, especially in the field of scientific computing, which has leveraged distributed computing for its computational needs.

Multithreaded programming is mainly concerned with providing a support for parallelism within a single machine. Task computing provides distribution by harnessing the compute power of several computing nodes. Hence, the presence of a distributed infrastructure is explicit in this model. Historically, the infrastructures that have been leveraged to execute tasks are clusters, supercomputers, and computing grids. Now clouds have emerged as an attractive solution to obtain a huge computing power on demand for the execution of distributed applications. To achieve it, suitable middleware is needed. A reference scenario for task computing is depicted in Figure 7.1.

The *middleware* is a software layer that enables the coordinated use of multiple resources, which are drawn from a datacenter or geographically distributed networked computers. A user

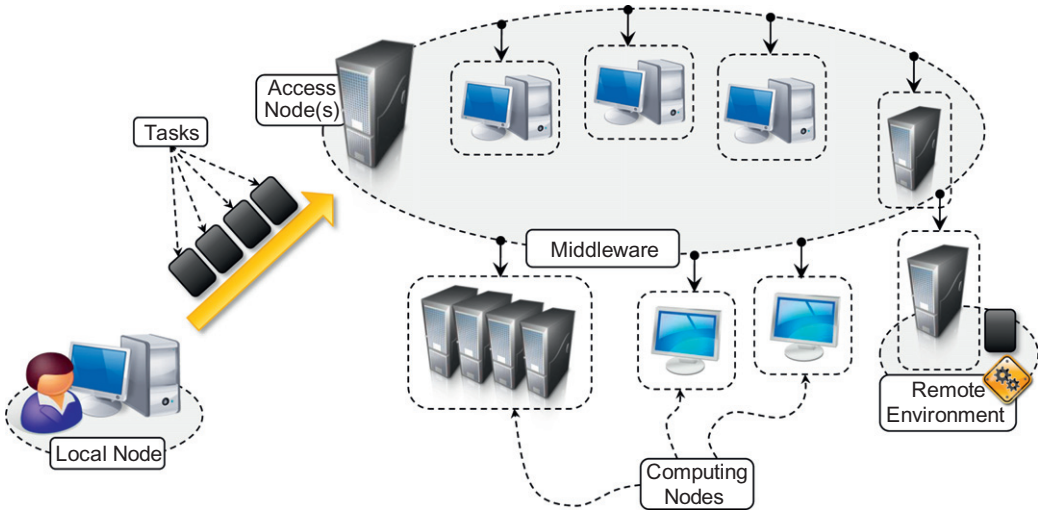

FIGURE 7.1

Task computing scenario.

submits the collection of tasks to the access point(s) of the middleware, which will take care of scheduling and monitoring the execution of tasks. Each computing resource provides an appropriate runtime environment, which may vary from implementation to implementation (a simple shell, a sandboxed environment, or a virtual machine). Task submission is normally done using the APIs provided by the middleware, whether a Web or programming language interface. Appropriate APIs are also provided to monitor task status and collect their results upon completion.

Because task abstraction is general, there exist different models of distributed applications falling under the umbrella of task computing. Despite this variety, it is possible to identify a set of common operations that the middleware needs to support the creation and execution of task-based applications. These operations are:

- Coordinating and scheduling tasks for execution on a set of remote nodes
- Moving programs to remote nodes and managing their dependencies
- Creating an environment for execution of tasks on the remote nodes
- Monitoring each task's execution and informing the user about its status
- Access to the output produced by the task

Models for task computing may differ in the way tasks are scheduled, which in turn depends on whether tasks are interrelated or they need to communicate among themselves.

7.1.1 Characterizing a task

A *task* is a general abstraction that identifies a program or a combination of programs that constitute a computing unit of a distributed application with a tangible output. A task represents a component of an application that can be logically isolated and executed separately. Distributed

applications are composed of tasks, the collective execution and interrelations of which define the nature of the applications. A task can be represented by different elements:

- A shell script composing together the execution of several applications
- A single program
- A unit of code (a Java/C++/.NET class) that executes within the context of a specific runtime environment

A task is generally characterized by input files, executable code (programs, shell scripts, etc.), and output files. In many cases the common runtime environment in which tasks execute is represented by the operating system or an equivalent sandboxed environment. A task may also need specific software appliances on the remote execution nodes in addition to the library dependencies that can be transferred to the node.

Some distributed applications may have additional constraints. For example, distributed computing frameworks that present the abstraction of tasks at programming level, by means of a class to inherit or an interface to implement, might require additional constraints (i.e., compliance to the inheritance rules) but also a richer set of features that can be exploited by developers. Based on the specific model of application, tasks might have dependencies.

7.1.2 Computing categories

According to the specific nature of the problem, a variety of categories for task computing have been proposed over time. These categories do not enforce any specific application model but provide an overall view of the characteristics of the problems. They implicitly impose requirements on the infrastructure and the middleware. Applications falling into this category are *high-performance computing (HPC)*, *high-throughput computing (HTC)*, and *many-task computing (MTC)*.

7.1.2.1 High-performance computing

High-performance computing (HPC) is the use of distributed computing facilities for solving problems that need large computing power. Historically, supercomputers and clusters are specifically designed to support HPC applications that are developed to solve "Grand Challenge" problems in science and engineering. The general profile of HPC applications is constituted by a large collection of compute-intensive tasks that need to be processed in a short period of time. It is common to have parallel and tightly coupled tasks, which require low-latency interconnection networks to minimize the data exchange time. The metrics to evaluate HPC systems are *floating-point operations per second (FLOPS)*, now tera-FLOPS or even peta-FLOPS, which identify the number of floating-point operations per second that a computing system can perform.

7.1.2.2 High-throughput computing

High-throughput computing (HTC) is the use of distributed computing facilities for applications requiring large computing power over a long period of time. HTC systems need to be robust and to reliably operate over a long time scale. Traditionally, computing grids composed of heterogeneous resources (clusters, workstations, and volunteer desktop machines) have been used to support HTC. The general profile of HTC applications is that they are made up of a large number of tasks of which the execution can last for a considerable amount of time (i.e., weeks or months). Classical

examples of such applications are scientific simulations or statistical analyses. It is quite common to have independent tasks that can be scheduled in distributed resources because they do not need to communicate. HTC systems measure their performance in terms of jobs completed per month.

7.1.2.3 Many-task computing

The *many-task computing (MTC)* [61] model started receiving attention recently and covers a wide variety of applications. It aims to bridge the gap between HPC and HTC. MTC is similar to HTC, but it concentrates on the use of many computing resources over a short period of time to accomplish many computational tasks. In brief, MTC denotes high-performance computations comprising multiple distinct activities coupled via file system operations. What characterizes MTC is the heterogeneity of tasks that might be of considerably different nature: Tasks may be small or large, single-processor or multiprocessor, compute-intensive or data-intensive, static or dynamic, homogeneous or heterogeneous. The general profile of MTC applications includes loosely coupled applications that are generally communication-intensive but not naturally expressed using the message-passing interface commonly found in HPC, drawing attention to the many computations that are heterogeneous but not embarrassingly parallel. Given the large number of tasks commonly composing MTC applications, any distributed facility with a large availability of computing elements is able to support MTC. Such facilities include supercomputers, large clusters, and emerging cloud infrastructures.

7.1.3 Frameworks for task computing

There are several frameworks that can be used to support the execution of task-based applications on distributed computing resources, including clouds. Some popular software systems that support the task-computing framework are *Condor* [5], *Globus Toolkit* [12], *Sun Grid Engine (SGE)* [13], *BOINC* [14], *Nimrod/G* [164], and *Aneka*.

Architecture of all these systems is similar to the general reference architecture depicted in Figure 7.1. They consist of two main components: a scheduling node (one or more) and worker nodes. The organization of the system components may vary. For example, multiple scheduling nodes can be organized in hierarchical structures. This configuration is quite common in the middleware for computing grids, which harness a variety of distributed resources from one or more organizations or sites. Each of these sites may have their own scheduling engine, especially if the system contributes to the grid but also serves local users.

A classic example is the cluster setup where the system might feature an installation of Condor or SGE for batch job submission; these services are generally used locally to the site, but the cluster can be integrated into a larger grid where meta-schedulers such as *GRAM (Globus Resource Allocation Manager)*[1] can dispatch a collection of jobs to the cluster. Other options include the presence of gateway nodes that do not have any scheduling capabilities and simply constitute the access point to the system. These nodes have indexing services that allow users to identify the available resources in the system, its current status, and the available schedulers. For worker nodes, they generally provide a sandboxed environment where tasks are executed on behalf of a specific

[1]Globus Resource Allocation Manager, or GRAM, is a software component of the Globus Toolkit that is in charge of locating, submitting, monitoring, and canceling jobs in grid computing systems.

user or within a given security context, limiting the operations that can be performed by programs such as file system access. File staging is also a fundamental feature supported by these systems. Clusters are normally equipped with shared file systems and parallel I/O facilities. Grids provide users with various staging facilities, such as credential access to remote worker nodes or automated staging services that transparently move files from user local machine to remote nodes.

Condor is probably the most widely used and long-lived middleware for managing clusters, idle workstations, and a collection of clusters. Condor-G is a version of Condor that supports integration with grid computing resources, such as those managed by Globus. Condor supports common features of batch-queuing systems along with the capability to checkpoint jobs and manage overload nodes. It provides a powerful job resource-matching mechanism, which schedules jobs only on resources that have the appropriate runtime environment. Condor can handle both serial and parallel jobs on a wide variety of resources. It is used by hundreds of organizations in industry, government, and academia to manage infrastructures ranging from a handful to well over thousands of workstations.

Sun Grid Engine (SGE), now Oracle Grid Engine, is middleware for workload and distributed resource management. Initially developed to support the execution of jobs on clusters, SGE integrated additional capabilities and now is able to manage heterogeneous resources and constitutes middleware for grid computing. It supports the execution of parallel, serial, interactive, and parametric jobs and features advanced scheduling capabilities such as budget-based and group-based scheduling, scheduling applications that have deadlines, custom policies, and advance reservation.

The Globus Toolkit is a collection of technologies that enable grid computing. It provides a comprehensive set of tools for sharing computing power, databases, and other services across corporate, institutional, and geographic boundaries without sacrificing local autonomy. The toolkit features software services, libraries, and tools for resource monitoring, discovery, and management as well as security and file management. The Globus Toolkit addresses core issues of grid computing: the management of a distributed environment composed of heterogeneous resources spanning different organizations, with all this condition implies in terms of security and interoperation. To provide valid support for grid computing in such scenarios, the toolkit defines a collection of interfaces and protocol for interoperation that enable different systems to integrate with each other and expose resources outside their boundaries.

Nimrod/G [164] is a tool for automated modeling and execution of parameter sweep applications (parameter studies) over global computational grids. It provides a simple declarative parametric modeling language for expressing parametric experiments. A domain expert can easily create a plan for a parametric experiment and use the Nimrod/G system to deploy jobs on distributed resources for execution. It has been used for a very wide range of applications over the years, ranging from quantum chemistry to policy and environmental impact. Moreover, it uses novel resource management and scheduling algorithms based on economic principles. Specifically, it supports deadline- and budget-constrained scheduling of applications on distributed grid resources to minimize the execution cost and at the same deliver results in a timely manner.

Berkeley Open Infrastructure for Network Computing (BOINC) is framework for volunteer and grid computing. It allows us to turn desktop machines into volunteer computing nodes that are leveraged to run jobs when such machines become inactive. BOINC is composed of two main components: the *BOINC server* and the *BOINC client*. The former is the central node that keeps track of

all the available resources and scheduling jobs; the latter is the software component that is deployed on desktop machines and that creates the BOINC execution environment for job submission. Given the volatility of BOINC clients, BOINC supports job checkpointing and duplication. Even if mostly focused on volunteer computing, BOINC systems can be easily set up to provide more stable support for job execution by creating computing grids with dedicated machines. To leverage BOINC, it is necessary to create an application project. When installing BOINC clients, users can decide the application project to which they want to donate the CPU cycles of their computer. Currently several projects, ranging from medicine to astronomy and cryptography, are running on the BOINC infrastructure.

7.2 Task-based application models

There are several models based on the concept of the task as the fundamental unit for composing distributed applications. What makes these models different from one another is the way in which tasks are generated, the relationships they have with each other, and the presence of dependencies or other conditions—for example, a specific set of services in the runtime environment—that have to be met. In this section, we quickly review the most common and popular models based on the concept of the task.

7.2.1 Embarrassingly parallel applications

Embarrassingly parallel applications constitute the most simple and intuitive category of distributed applications. As we discussed in Chapter 6, embarrassingly parallel applications constitute a collection of tasks that are independent from each other and that can be executed in any order. The tasks might be of the same type or of different types, and they do not need to communicate among themselves.

This category of applications is supported by the majority of the frameworks for distributed computing. Since tasks do not need to communicate, there is a lot of freedom regarding the way they are scheduled. Tasks can be executed in any order, and there is no specific requirement for tasks to be executed at the same time. Therefore, scheduling these applications is simplified and mostly concerned with the optimal mapping of tasks to available resources. Frameworks and tools supporting embarrassingly parallel applications are the Globus Toolkit, BOINC, and Aneka.

There are several problems that can be modeled as embarrassingly parallel. These include image and video rendering, evolutionary optimization, and model forecasting. In image and video rendering the task is represented by the rendering of a pixel (more likely a portion of the image) or a frame, respectively. For evolutionary optimization metaheuristics, a task is identified by a single run of the algorithm with a given parameter set. The same applies to model forecasting applications. In general, scientific applications constitute a considerable source of embarrassingly parallel applications, even though they mostly fall into the more specific category of parameter sweep applications.

7.2.2 Parameter sweep applications

Parameter sweep applications are a specific class of embarrassingly parallel applications for which the tasks are identical in their nature and differ only by the specific parameters used to execute them. Parameter sweep applications are identified by a template task and a set of parameters. The *template task* defines the operations that will be performed on the remote node for the execution of tasks. The template task is parametric, and the parameter set identifies the combination of variables whose assignments specialize the template task into a specific instance. The combination of parameters, together with their range of admissible values, identifies the multidimensional domain of the application, and each point in this domain identifies a task instance.

Any distributed computing framework that provides support for embarrassingly parallel applications can also support the execution of parameter sweep applications, since the tasks composing the application can be executed independently of each other. The only difference is that the tasks that will be executed are generated by iterating over all the possible and admissible combinations of parameters. This operation can be performed by frameworks natively or tools that are part of the distributed computing middleware. For example, Nimrod/G is natively designed to support the execution of parameter sweep applications, and Aneka provides client-based tools for visually composing a template task, defining parameters, and iterating over all the possible combinations of such parameters.

A plethora of applications fall into this category. Mostly they come from the scientific computing domain: evolutionary optimization algorithms, weather-forecasting models, computational fluid dynamics applications, Monte Carlo methods, and many others. For example, in the case of evolutionary algorithms it is possible to identify the domain of the applications as a combination of the relevant parameters of the algorithm. For genetic algorithms these might be the number of individuals of the population used by the optimizer and the number of generations for which to run the optimizer. The following example in pseudo-code demonstrates how to use parameter sweeping for the execution of a generic evolutionary algorithm which can be configured with different population sizes and generations.

```
individuals = {100, 200, 300, 500, 1000}
generations = {50, 100, 200, 400}
foreach indiv in individuals do
foreach generation in generations do

 task = generate_task(indiv, generation)
 submit_task(task)
```

The algorithm sketched in the example defines a bidimensional domain composed of discrete variables and then iterated over each combination of individuals and generations to generate all the tasks composing the application. In this case 20 tasks are generated. The function *generate_task* is specific to the application and creates the task instance by substituting the values of *indiv* and *generation* to the corresponding variables in the template definition. The function *submit_task* is specific to the middleware used and performs the actual task submission.

A template task is in general a composition of operations concerning the execution of legacy applications with the appropriate parameters and set of file system operations for moving data. Therefore, frameworks that natively support the execution of parameter sweep applications often

provide a set of useful commands for manipulating or operating on files. Also, the template task is often expressed as single file that composes together the commands provided. The commonly available commands are:

- *Execute*. Executes a program on the remote node.
- *Copy*. Copies a file to/from the remote node.
- *Substitute*. Substitutes the parameter values with their placeholders inside a file.
- *Delete*. Deletes a file.

All these commands can operate with parameters that are substituted with their actual values for each task instance.

Figures 7.2 and 7.3 provide examples of two possible task templates, the former as defined according to the notation used by Nimrod/G, and the latter as required by Aneka.

The template file has two sections: a header for the definition of the parameters, and a task definition section that includes shell commands mixed with Nimrod/G commands. The prefix *node:* identifies the remote location where the task is executed. Parameters are identified with the *${...}* notation. The example shown remotely executes the *echo* command and copies to the local user directory the output of the command by saving it into a file named according to the values of the parameters *x* and *y*.

The Aneka Parameter Sweep file defines the template task for executing the BLAST application. The file is an XML document containing several sections, the most important of which are *sharedFiles*, *parameters*, and *task*. *parameters* contains the definition of the parameters that will customize the template task. Two different types of parameters are defined: a single value and a range parameter. The *sharedFiles* section contains the files that are required to execute the task; *task* specifies the operations that characterize the template task. The task has a collection of input and output files for which local and remote paths are defined, as well as a collection of commands. In the case presented, a simple *execute* command is shown. With respect to the previous example there is no need to explicitly move the files to the remote destination, but this operation is automatically performed by Aneka.

7.2.3 MPI applications

Message Passing Interface (MPI) is a specification for developing parallel programs that communicate by exchanging messages. Compared to earlier models, MPI introduces the constraint of communication that involves MPI tasks that need to run at the same time. MPI has originated as an attempt to create common ground from the several distributed shared memory and message-passing

```
parameter x float range from 1 to 10 step 1;
parameter y float range from -4 to 5 step 1;

task main
  node:execute /bin/echo X:${x} Y:${y} > output
  copy node:output output.`expr ${y}\*10+${x}`
endtask
```

FIGURE 7.2

Nimrod/G task template definition.

```
<psm>
  <name>Aneka Blast</name>
  <description>BLAST simulation</description>
  <workspace>C:\Projects\Explorer\blast</workspace>
  <parameters>
    <single name="p" type="String" comment="The name of the program" value="blastn"/>
    <single name="d" type="String" comment="The database file" value="ecoli.nt"/>
    <range name="s" type="String" comment="The sequence file" from="0" to="2" interval="1"/>
  </parameters>
  <sharedFiles>
    <file path="blastall.exe" vpath="blastall.exe"/>
    <file path="ecoli.nt.nhr" vpath="ecoli.nt.nhr"/>
    <file path="ecoli.nt.nin" vpath="ecoli.nt.nin"/>
    <file path="ecoli.nt.nnd" vpath="ecoli.nt.nnd"/>
    <file path="ecoli.nt.nni" vpath="ecoli.nt.nni"/>
    <file path="ecoli.nt.nsd" vpath="ecoli.nt.nsd"/>
    <file path="ecoli.nt.nsi" vpath="ecoli.nt.nsi"/>
    <file path="ecoli.nt.nsq" vpath="ecoli.nt.nsq"/>
  </sharedFiles>
  <task>
    <inputs>
      <file path="seq($s).txt" vpath="seq($s).txt"/>
    </inputs>
    <outputs>
      <file path="output($s).txt" vpath="output($s).txt"/>
    </outputs>
    <commands>
      <execute cmd="blastall.exe" args="-p ($p) -d ($d) -i seq($s).txt -o output($s).txt"/>
    </commands>
  </task>
</psm>
```

FIGURE 7.3

Aneka parameter sweep file.

infrastructures available for distributed computing. Nowadays, MPI has become a *de facto* standard for developing portable and efficient message-passing HPC applications. Interface specifications have been defined and implemented for C/C++ and Fortran.

MPI provides developers with a set of routines that:

- Manage the distributed environment where MPI programs are executed

- Provide facilities for point-to-point communication
- Provide facilities for group communication
- Provide support for data structure definition and memory allocation
- Provide basic support for synchronization with blocking calls

The general reference architecture is depicted in Figure 7.4. A distributed application in MPI is composed of a collection of MPI processes that are executed in parallel in a distributed infrastructure that supports MPI (most likely a cluster or nodes leased from clouds).

MPI applications that share the same MPI runtime are by default as part of a global group called *MPI_COMM_WORLD*. Within this group, all the distributed processes have a unique identifier that allows the MPI runtime to localize and address them. It is possible to create specific groups as subsets of this global group—for example, for isolating all the MPI processes that belong to the same application. Each MPI process is assigned a rank within the group to which it belongs. The rank is a unique identifier that allows processes to communicate with each other within a group. Communication is made possible by means of a communicator component that can be defined for each group.

To create an MPI application it is necessary to define the code for the MPI process that will be executed in parallel. This program has, in general, the structure described in Figure 7.5. The section of code that is executed in parallel is clearly identified by two operations that set up the MPI environment and shut it down, respectively. In the code section defined within these two operations, it is possible to use all the MPI functions to send or receive messages in either asynchronous or synchronous mode.

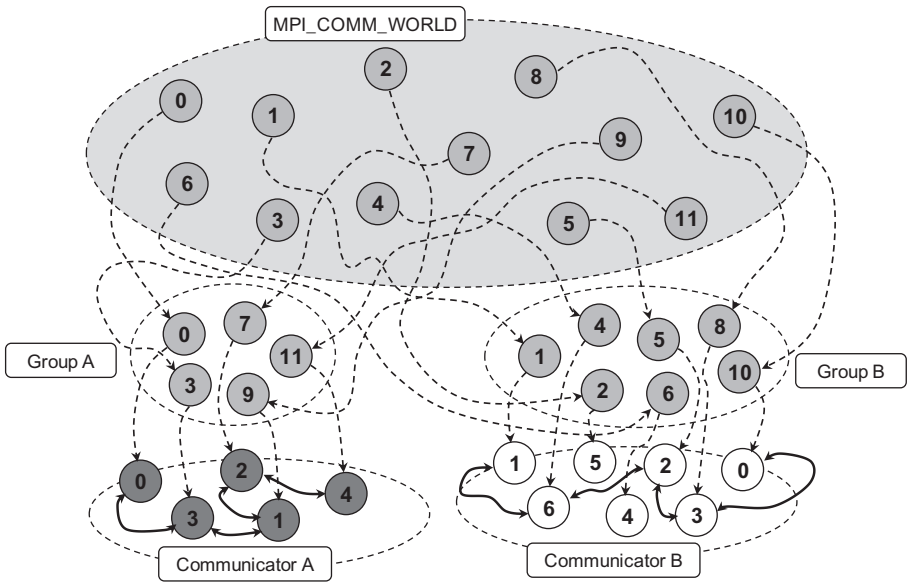

FIGURE 7.4

MPI reference scenario.

The diagram in Figure 7.5 might suggest that the MPI might allow the definition of completely symmetrical applications, since the portion of code executed in each node is the same. In reality, it is possible to implement distributed applications based on complex communication patterns by differentiating the operations performed by each node according to the rank of the program, which is known at runtime. A common model used in MPI is the *master-worker model*, whereby one MPI process (usually the one with rank 0) coordinates the execution of others that perform the same task.

Once the program has been defined in one of the available MPI implementations, it is compiled with a modified version of the compiler for the language. This compiler introduces additional code in order to properly manage the MPI runtime. The output of the compilation process can be run as a distributed application by using a specific tool provided with the MPI implementation.

A general installation that supports the execution of the MPI application is composed of a cluster. In this scenario MPI is normally installed in the shared file system and an MPI daemon is started on each node of the cluster in order to coordinate the parallel execution of MPI applications. Once the environment is set up, it is possible to run parallel applications by using the tools provided with the MPI implementation and to specify several options, such as the number of nodes to use to run the application.

At present there are several MPI implementations that can be leveraged to develop distributed applications, and the MPI specifications have currently reached version 2. One of the most popular MPI software environments (www.mcs.anl.gov/mpi/) is developed by the Argonne National Laboratory in the United States. MPI has gained a good deal of success as a parallel and distributed

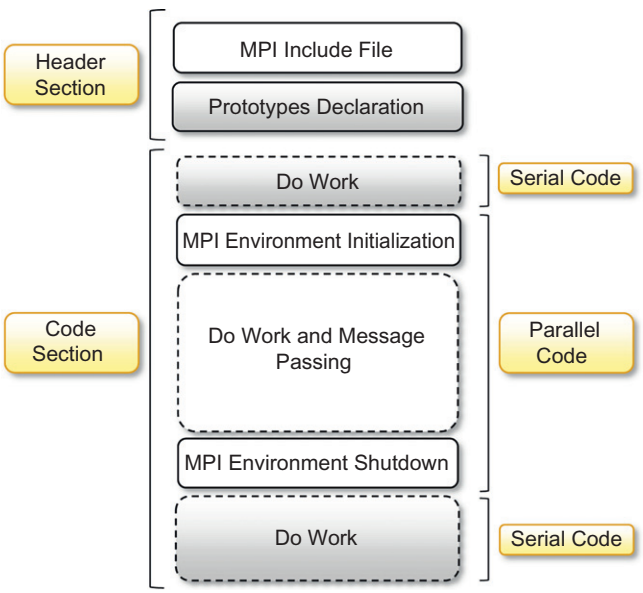

FIGURE 7.5

MPI program structure.

programming model for CPU-intensive mathematical computations such as linear systems solvers, matrix computations, finite element computations, linear algebra, and numerical simulations.

7.2.4 Workflow applications with task dependencies

Workflow applications are characterized by a collection of tasks that exhibit dependencies among them. Such dependencies, which are mostly data dependencies (i.e., the output of one task is a prerequisite of another task), determine the way in which the applications are scheduled as well as where they are scheduled. Concerns in this case are related to providing a feasible sequencing of tasks and to optimizing the placement of tasks so that the movement of data is minimized.

7.2.4.1 What is a workflow?

The term *workflow* has a long tradition in the business community, where the term is used to describe a composition of services that all together accomplish a business process. As defined by the Workflow Management Coalition, a workflow is *the automation of a business process, in whole or part, during which documents, information, or tasks are passed from one participant (a resource; human or machine) to another for action, according to a set of procedural rules* [64]. The concept of workflow as a structured execution of tasks that have dependencies on each other has demonstrated itself to be useful for expressing many scientific experiments and gave birth to the idea of *scientific workflow*. Many scientific experiments are a combination of problem-solving components, which, connected in a particular order, define the specific nature of the experiment. When such experiments exhibit a natural parallelism and need to execute a large number of operations or deal with huge quantities of data, it makes sense to execute them on a distributed infrastructure. In the case of scientific workflows, the process is identified by an application to run, the elements that are passed among participants are mostly tasks and data, and the participants are mostly computing or storage nodes. The set of procedural rules is defined by a workflow definition scheme that guides the scheduling of the application. A scientific workflow generally involves data management, analysis, simulation, and middleware supporting the execution of the workflow.

A scientific workflow is generally expressed by a *directed acyclic graph (DAG)*, which defines the dependencies among tasks or operations. The nodes on the DAG represent the tasks to be executed in a workflow application; the arcs connecting the nodes identify the dependencies among tasks and the data paths that connect the tasks. The most common dependency that is realized through a DAG is *data dependency*, which means that the output files of a task (or some of them) constitute the input files of another task. This dependency is represented as an arc originating from the node that identifies the first task and terminating in the node that identifies the second task.

The DAG in Figure 7.6 describes a sample Montage workflow.[2] *Montage* is a toolkit for assembling images into mosaics; it has been specially designed to support astronomers in composing the images taken from different telescopes or points of view into a coherent image. The toolkit provides several applications for manipulating images and composing them together; some of the applications perform background reprojection, perspective transformation, and brightness and color

[2]Montage workflow: http://vgrads.rice.edu/research/applications/images/montage-workflow/view.

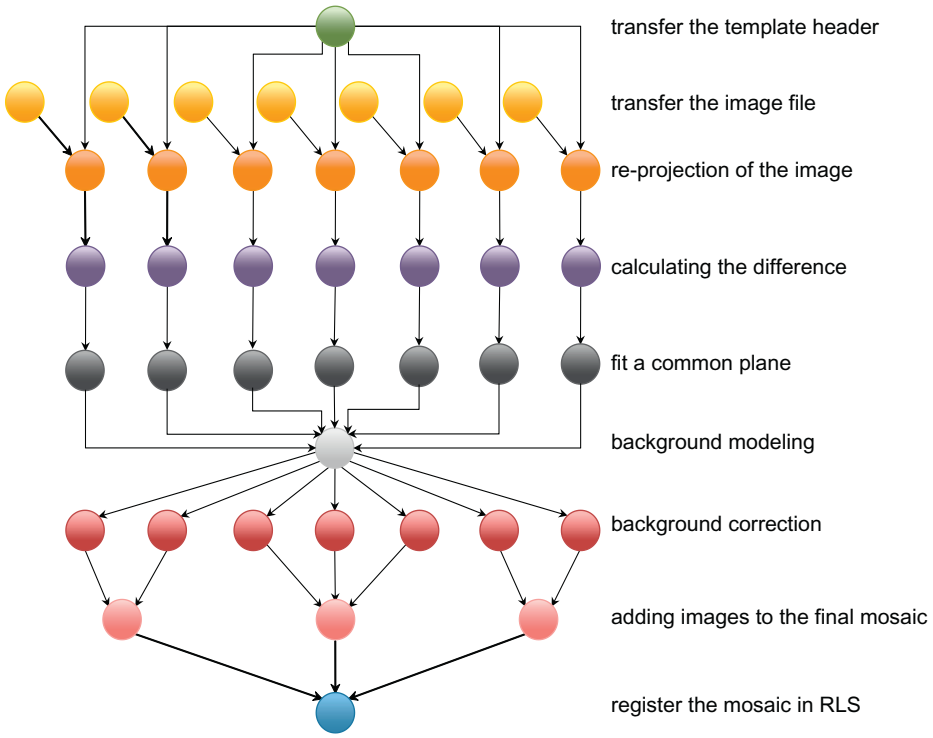

FIGURE 7.6

Sample Montage workflow.

correction. The workflow depicted in Figure 7.6 describes the general process for composing a mosaic; the labels on the right describe the different tasks that have to be performed to compose a mosaic. In the case presented in the diagram, a mosaic is composed of seven images. The entire process can take advantage of a distributed infrastructure for its execution, since there are several operations that can be performed in parallel. For each of the image files, the following process has to be performed: image file transfer, reprojection, calculation of the difference, and common plane placement. Therefore, each of the images can be processed in parallel for these tasks. Here is where a distributed infrastructure helps in executing workflows.

There might be another reason for executing workflows on a distributed infrastructure: It might be convenient to move the computation on a specific node because of data locality issues. For example, if an operation needs to access specific resources that are only available on a specific node, that operation cannot be performed elsewhere, whereas the rest of the operations might not have the same requirements. A scientific experiment might involve the use of several problem-solving components that might require the use of specific instrumentation; in this case all the tasks that have these constraints need to be executed where the instrumentation is available, thus creating a distributed execution of a process that is not parallel in principle.

7.2.4.2 Workflow technologies

Business-oriented computing workflows are defined as compositions of services, and there are specific languages and standards for the definition of workflows, such as *Business Process Execution Language (BPEL)* [65]. In the case of scientific computing there is no common ground for defining workflows, but several solutions and workflow languages coexist [66]. Despite such differences, it is possible to identify an abstract reference model for a workflow management system [67], as depicted in Figure 7.7. Design tools allow users to visually compose a workflow application. This specification is normally stored in the form of an XML document based on a specific workflow language and constitutes the input of the workflow engine, which controls the execution of the workflow by leveraging a distributed infrastructure. In most cases, the workflow engine is a client-side component that might interact directly with resources or with one or several middleware components for executing the workflow. Some frameworks can natively support the execution of workflow applications by providing a scheduler capable of directly processing the workflow specification.

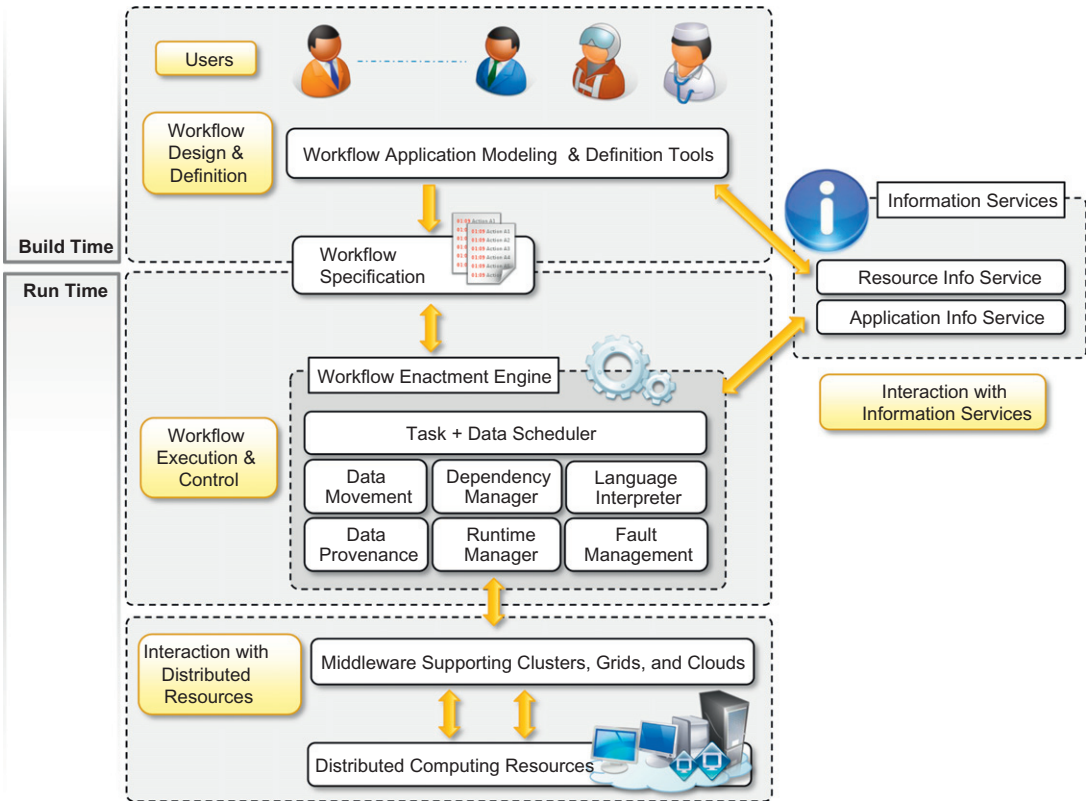

FIGURE 7.7

Abstract model of a workflow system.

Some of the most relevant technologies for designing and executing workflow-based applications are *Kepler*, *DAGMan*, *Cloudbus Workflow Management System*, and *Offspring*.

Kepler [68] is an open-source scientific workflow engine built from the collaboration of several research projects. The system is based on the *Ptolemy II* system [72], which provides a solid platform for developing dataflow-oriented workflows. Kepler provides a design environment based on the concept of actors, which are reusable and independent blocks of computation such as Web services, database calls, and the like. The connection between actors is made with ports. An actor consumes data from the input ports and writes data/results to the output ports. The novelty of Kepler is in its ability to separate the flow of data among components from the coordination logic that is used to execute workflow. Thus, for the same workflow, Kepler supports different models, such as synchronous and asynchronous models. The workflow specification is expressed using a proprietary XML language.

DAGMan (Directed Acyclic Graph Manager) [69], part of the Condor [5] project, constitutes an extension to the Condor scheduler to handle job interdependencies. Condor finds machines for the execution of programs but does not support the scheduling of jobs in a specific sequence. Therefore, DAGMan acts as a metascheduler for Condor by submitting the jobs to the scheduler in the appropriate order. The input of DAGMan is a simple text file that contains the information about the jobs, pointers to their job submission files, and the dependencies among jobs.

Cloudbus Workflow Management System (WfMS) [70] is a middleware platform built for managing large application workflows on distributed computing platforms such as grids and clouds. It comprises software tools that help end users compose, schedule, execute, and monitor workflow applications through a Web-based portal. The portal provides the capability of uploading workflows or defining new ones with a graphical editor. To execute workflows, WfMS relies on the Gridbus Broker, a grid/cloud resource broker that supports the execution of applications with quality-of-service (QoS) attributes over a heterogeneous distributed computing infrastructure, including Linux-based clusters, Globus, and Amazon EC2. WfMS uses a proprietary XML language for the specification of workflows.

A different perspective is taken by *Offspring* [71], which offers a programming-based approach to developing workflows. Users can develop strategies and plug them into the environment, which will execute them by leveraging a specific distribution engine. The advantage provided by Offspring over other solutions is the ability to define dynamic workflows. This strategy represents a semistructured workflow that can change its behavior at runtime according to the execution of specific tasks. This allows developers to dynamically control the dependencies of tasks at runtime rather than statically defining them. Offspring supports integration with any distributed computing middleware that can manage a simple bag-of-tasks application. It provides a native integration with Aneka and supports a simulated distribution engine for testing strategies during development. Because Offspring allows the definition of workflows in the form of plug-ins, it does not use any XML specification.

7.3 Aneka task-based programming

Aneka provides support for all the flavors of task-based programming by means of the *Task Programming Model*, which constitutes the basic support given by the framework for supporting

the execution of bag-of-tasks applications. Task programming is realized through the abstraction of the *Aneka.Tasks.ITask*. By using this abstraction as a basis support for execution of legacy applications, parameter sweep applications and workflows have been integrated into the framework. In this section, we introduce the fundamental concepts of the model and provide examples of how to develop applications for all the previously discussed application models.

7.3.1 Task programming model

The *Task Programming Model* provides a very intuitive abstraction for quickly developing distributed applications on top of Aneka. It provides a minimum set of APIs that are mostly centered on the *Aneka.Tasks.ITask* interface. This interface, together with the services supporting the execution of tasks in the middleware, constitutes the core feature of the model. Figure 7.8 provides an overall view of the components of the Task Programming Model and their roles during application execution.

Developers create distributed applications in terms of *ITask* instances, the collective execution of which describes a running application. These tasks, together with all the required dependencies (data files and libraries), are grouped and managed through the *AnekaApplication* class, which is specialized to support the execution of tasks. Two other components, *AnekaTask* and *TaskManager*, constitute the client-side view of a task-based application. The former constitutes the runtime wrapper Aneka uses to represent a task within the middleware; the latter is the underlying component that interacts with Aneka, submits the tasks, monitors their execution, and collects the results. In the middleware, four services coordinate their activities in order to execute task-based applications. These are *MembershipCatalogue*, *TaskScheduler*, *ExecutionService*, and *StorageService*.

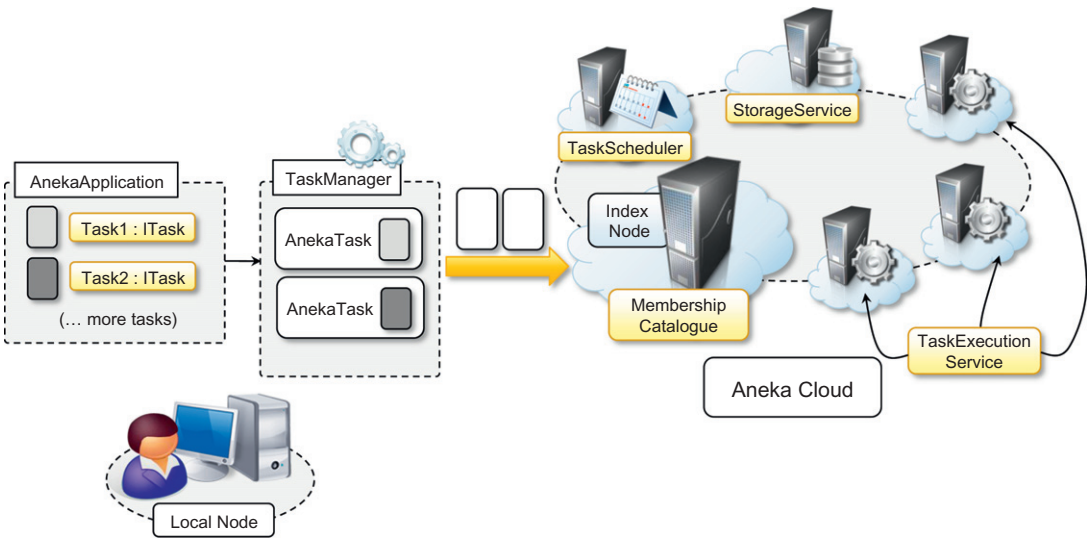

FIGURE 7.8

Task programming model scenario.

MembershipCatalogue constitutes the main access point of the cloud and acts as a service directory to locate the *TaskScheduler* service that is in charge of managing the execution of task-based applications. Its main responsibility is to allocate task instances to resources featuring the *ExecutionService* for task execution and for monitoring task state. If the application requires the data transfer support in the form of data files, input files, or output files, an available *StorageService* will be used as a staging facility for the application.

The features provided by the task model are completed by a Web service that allows any client to submit the execution of tasks to Aneka. The procedure for submitting tasks through the Web services is the same as that done by using the framework APIs. The user creates an application on Aneka and submits tasks within the context of this application. The Web service limits the type of tasks that can be submitted. Only a limited collection of tasks is available for submission; despite that, these tasks cover the functionality commonly found in other distributed computing systems.

7.3.2 Developing applications with the task model

Execution of task-based applications involves several components. The development of such applications is limited to the following operations:

- Defining classes implementing the *ITask* interface
- Creating a properly configured *AnekaApplication* instance
- Creating *ITask* instances and wrapping them into *AnekaTask* instances
- Executing the application and waiting for its completion

Moreover, from a design point of view, the process of defining a task application ultimately reduces to the definition of the classes that implement *ITask*, which will be those that contribute to form the workload generated by the application.

7.3.2.1 ITask and AnekaTask

Almost all the client-side features for developing task-based applications with Aneka are contained in the *Aneka.Tasks* namespace (*Aneka.Tasks.dll*). The most important component for designing tasks is the *ITask* interface, which is defined in Listing 7.1. This interface exposes only one method: *Execute*. The method is invoked in order to execute the task on the remote node.

The *ITask* interface provides a programming approach for developing native tasks, which means tasks implemented in any of the supported programming languages of the .NET framework. The restrictions on implementing task classes are minimal; other than implementing the *ITask* interface, they need to be serializable, since task instances are created and moved over the network. Listing 7.2 describes a simple implementation of a task class that computes the Gaussian distribution for a given point *x*.

ITask provides minimum restrictions on how to implement a task class and decouples the specific operation of the task from the runtime wrapper classes. It is required for managing tasks within Aneka. This role is performed by the *AnekaTask* class that represents the task instance in accordance with the Aneka application model APIs. This class extends the *Aneka.Entity.WorkUnit* class and provides the feature for embedding *ITask* instances. *AnekaTask* is mostly used internally, and for end users it provides facilities for specifying input and output files for the task.

```
namespace Aneka.Tasks
{
    ///<summary>
    ///Interface ITask. Defines the interface for implementing a task.
    ///</summary>
    public interface ITask
    {
        ///<summary>
        ///Executes the sine function.
        ///</summary>
        public void Execute();
    }
}
```

LISTING 7.1

ITask interface.

Listing 7.3 describes how to wrap an *ITask* instance into an *AnekaTask*. It also shows how to add input and output files specific to a given task. The Task Programming Model leverages the basic capabilities for file management that belong to the *WorkUnit* class, from which the *AnekaTask* class inherits. As discussed when we presented the Aneka Application Model (see Chapter 5). *WorkUnit* has two collections of files, *InputFiles* and *OutputFiles*; developers can add files to these collections and the runtime environment will automatically move these files where it is necessary. Input files will be staged into the Aneka Cloud and moved to the remote node where the task is executed. Output files will be collected from the execution node and moved to the local machine or a remote FTP server.

7.3.2.2 Controlling task execution

Task classes and *AnekaTask* define the computation logic of a task-based application, whereas the *AnekaApplication* class provides the basic feature for implementing the coordination logic of the application.

AnekaApplication is a generic class that can be specialized to support different programming models. In task programming, it assumes the form of *AnekaApplication* $<$ *AnekaTask, TaskManager* $>$. The operations provided for the task model as well as for other programming models are:

- Static and dynamic task submission
- Application state and task state monitoring
- Event-based notification of task completion or failure

By composing these features all together, it is possible to define the logic that is required to implement a specific task application. Static submission is a very common pattern in the case of task-based applications, and it involves the creation of all the tasks that need to be executed in one loop and their submission as a single bag. More complex task submission strategies are then required for implementing workflow-based applications, where the execution of tasks is determined by dependencies among them. In this case a dynamic submission of tasks is a more efficient

```
// File: GaussTask.cs
using System;
using Aneka.Tasks;

namespace GaussSample
{
   /// <summary>
   /// Class GaussTask. Implements the ITask interface for computing the Gauss function.
   /// </summary>
   [Serializable]
   public class GaussTask : ITask
   {
      /// <summary>
      /// Input value.
      /// </summary>
      private double x;
      /// <summary>
      /// Gets the input value of the Gauss function.
      /// </summary>
      public double X { get { return this.x; } set { this.x = value; } }
      /// <summary>
      /// Result value.
      /// </summary>
      private double y;
      /// <summary>
      /// Gets the result value of the Gauss function.
      /// </summary>
      public double Y { get { return this.y; } set { this.y = value; } }

      /// <summary>
      /// Executes the Gauss function.
      /// </summary>
      public void Execute()
      {
         this.y = Math.Exp(-this.x*this.x);
      }
   }
}
```

LISTING 7.2

ITask interface implementation.

technique and involves the submission of tasks as a result of the event-based notification mechanism implemented in the *AnekaApplication* class.

Listing 7.4 shows how to create and submit 400 Gauss tasks as a bag by using the static submission approach. The *AnekaApplication* class has a collection of tasks containing all the tasks that have been submitted for execution. Each task can be referenced using its unique identifier (*WorkUnit.Id*) by the indexer operator [] applied to the application class. In the case of static submission, the tasks are added to the application, and the method *SubmitExecution()* is called.

```
// create a Gauss task and wraps it into an AnekaTaskinstance
GaussTaskgauss = new GaussTask();
AnekaTask task = newAnekaTask(gauss);
// add one input and one output files
task.AddFile("input.txt", FileDataType.Input, FileAttributes.Local);
task.AddFile("result.txt", FileDataType.Output, FileAttributes.Local);
```

LISTING 7.3

Wrapping an *ITask* into an *AnekaTask* Instance.

```
// get an instance of the Configuration class from file
Configuration conf = Configuration.GetConfiguration("conf.xml");
// specify that the submission of task is static (all at once)
conf.SingleSubmission = true;
AnekaApplication<AnekaTask, TaskManager> app =
new AnekaApplication<Task,TaskManager>(conf);
for(int i=0; i<400; i++)
{
   GaussTaskgauss = new GaussTask();
   gauss.X = i;
   AnekaTask task = new AnekaTask(gauss);
   // add the task to the bag of work units to submit
   app.AddWorkunit(task);
}
// submit the entire bag
app.SubmitExecution();
```

LISTING 7.4

Static task submission.

A different scenario is constituted by dynamic submission, where tasks are submitted as a result of other events that occur during the execution—for example, the completion or the failure of previously submitted tasks or other conditions that are not related to the interaction of Aneka. In this case, developers have more freedom in selecting the most appropriate task submission strategy. For example, it is possible that an initial bag of tasks is submitted as described in Listing 7.4, and subsequently, as a result of the completion of some tasks, other tasks are generated and submitted. To implement this scenario it is necessary to rely on the event-based notification system provided by the *AnekaApplication* class and trigger the submission of tasks according to the firing of specific events. In particular, we are interested in the *WorkUnitFailed* and *WorkUnitCompleted* events.

Listing 7.5 extends the previous example and implements a dynamic task submission strategy for refining the computation of Gaussian distribution. Both static and dynamic task submissions are used: An initial bag of 400 Gauss tasks is submitted to Aneka, and as soon as these tasks are completed, a new task for the computation of an intermediate value of the distribution is submitted. To capture the failure and the completion of tasks, it is necessary to listen to the events

```
///<summary>
///Main method for submitting tasks.
///</summary>
public void SubmitApplication()
{
   // get an instance of the Configuration class from file
   Configuration conf = Configuration.GetConfiguration("conf.xml");
   // specify that the submission of task is dynamic
   conf.SingleSubmission = false;
   AnekaApplication<AnekaTask, TaskManager> app =
                  new AnekaApplication<Task,TaskManager>(conf);
   // attach methods to the event handler that notify the client code
   // when tasks are completed or failed
   app.WorkUnitFailed +=
                  new EventHandler<WorkUnitEventArgs<AnekaTask>>(this.OnWorkUnitFailed);
   app.WorkUnitFinished +=
                  new EventHandler<WorkUnitEventArgs<AnekaTask>>(this.OnWorkUnitFinished);
   for(int i=0; i<400; i++)
   {
      GaussTask gauss = new GaussTask();
      gauss.X = i;
      AnekaTask task = newAnekaTask(gauss);
      // add the task to the bag of work units to submit
      app.AddWorkunit(task);
   }
   // submit the entire bag
   app.SubmitExecution();
}
/// <summary>
/// Event handler for task failure.
/// </summary>
/// <param name="sender">Event source: the application instance.</param>
/// <param name="args">Event arguments.</param>
private void OnWorkUnitFailed(object sender, WorkUnitEventArgs<AnekaTask>args)
{
   // do nothing, we are not interested in task failure at the moment
   // just dump to console the failure.
   if (args.WorkUnit != null)
   {
      Exception error = args.WorkUnit.Exception;
      Console.WriteLine("Task {0} failed - Exception: {1}",
      args.WorkUnit.Name, (error == null ? "[Not given]" : error.Message);
   }
}
```

LISTING 7.5

Dynamic task submission.

WorkUnitFailed and *WorkUnitFinished*. The event signature requires the methods to have an object parameter (as for all the event handlers), which will contain the application instance, and a *WorkUnitEventArgs*$<$*AnekaTask*$>$ argument containing the information about the *WorkUnit* that triggered the event. This class exposes a *WorkUnit* property that, if not null, gives access to the

```
/// <summary>
/// Event handler for task completion.
/// </summary>
/// <param name="sender">Event source: the application instance.</param>
/// <param name="args">Event arguments.</param>
private void OnWorkUnitFinished(object sender, WorkUnitEventArgs<AnekaTask> args)
{
   // if the task is completed for sure we have a WorkUnit instance
   // and we do not need to check as we did before.
   GaussTask gauss = (GaussTask) args.WorkUnit.Task;
   // we check whether it is an initially submitted task or a task
   // that we submitted as a reaction to the completion of another task
   if (task.X - Math.Abs(task.X) == 0)
   {
      // ok it was an original task, then we increment of 0.5 the
      // value of X and submit another task

      GaussTask fraq = GaussTask();
      fraq.X = gauss.X + 0.5;
      AnekaTask task =new AnekaTask(fraq);

      // we call the ExecuteWorkUnit method that is used
      // for dynamic submission
      app.ExecuteWorkUnit(task);
   }
   Console.WriteLine("Task {0} completed - [X:{1},Y:{2}]",
   args.WorkUnit.Name, gauss.X, gauss.Y);
}
```

LISTING 7.5

Dynamic task submission.

task instance. The event handler for the task failure simply dumps the information that the task is failed to the console, with, if possible, additional information about the error that occurred. The event handler for task completion checks whether the task completed was submitted within the original bag, and in this case submits another task by using the *ExecuteWorkUnit(AnekaTask task)* method. To discriminate tasks submitted within the initial bag and other tasks, the value of *GaussTask.X* is used. If *X* contains a value with no fractional digits, it is an initial task; otherwise, it is not.

Static and dynamic submission influence the way the application termination condition is determined. In static submission the determination of this condition is automatic: Once all the initially submitted tasks are failed or completed, the application is terminated. It is important in this case to activate, in the configuration of the application, the *SingleSubmission* flag by setting it to *true*. This will tell the runtime to automatically determine the completion of the application. In dynamic submission, it is impossible for the runtime to determine the termination of the application, since it is always to possible to submit new tasks. In this case it is the responsibility of the developer to signal to the application class the termination of the application by invoking the *StopExecution* method when appropriate.

In designing the coordination logic of the application, it is important to note that the task submission identifies an asynchronous execution pattern, which means that the method *SubmitExecution*, as well as the method *ExecuteWorkUnit*, returns when the submission of tasks is completed, but not the actual completion of tasks. This requires the developer to put in place the proper synchronization logic to let the main thread of the application wait until all the tasks are terminated and the application is completed. This behavior can be implemented using the synchronization APIs provided by the *System.Threading* namespace: *System.Threading.AutoResetEvent* or *System.Threading.ManualResetEvent.* These two APIs, together with a minimal logic, count all the tasks to be collected and signal the main thread (put in waiting state by calling the method *WaitHandle.Wait()*) once all tasks are terminated. It can also provide the required infrastructure for properly managing the execution flow of the application. Listing 7.6 provides a complete implementation of the task submission program, implementing dynamic submission and the appropriate synchronization logic.

The listing provides a complete definition of the *GaussApp* class, which also contains the main entry point of the application. A very simple logic for controlling the execution of the application has been implemented. The *GaussApp* application keeps track of the number of currently running tasks by using the *taskCount* field. When this value reaches zero, there are no more tasks to wait for and the application is stopped by calling *StopExecution*. This method fires the *ApplicationFinished* event whose event handler (*OnApplicationFinished*) unblocks the main thread by signalling the semaphore. The value of *taskCount* is initially set to 400, which is the size of the initial bag of tasks. Every time a task fails or completes, this field is decremented by one unit; if there is a new task submission, the field is incremented by one unit. At the end of the two event handlers (*OnWorkUnitFailed* and *OnWorkUnitFinished*), the value of *taskCount* is checked to see whether it is equal to zero and it is necessary to stop the application. We can observe that, besides the use of the *ManualResetEvent*, there is no need for other synchronization structures. Because the code that manipulates the value of *taskCount* is executed in one single thread, there will not be any races while incrementing or decrementing the value.

A final aspect that can be considered for controlling the execution of the task application is the resubmission strategy that is used. By default the configuration of the application sets the resubmission strategy as manual; this means that if a task fails because of an exception that occurred during its execution, the task instance is sent back to the client application, and it is the developer's responsibility to resubmit the task if necessary. In automatic resubmission, Aneka will keep resubmitting the task until a maximum number of attempts is reached. If the task keeps failing, the task failure event will eventually be fired. This property can be controlled by setting the configuration value *Configuration.ResubmitMode* to *ResubmitMode.Manual* or *ResubmitMode.Auto*. As previously stated, this property is set to *ResubmitMode.Manual* by default.

7.3.2.3 File management

Task-based applications normally deal with files to perform their operations. As we've discussed, files may constitute input data for tasks, may contain the result of a computation, or may represent executable code or library dependencies. Therefore, providing support for file transfers for task-based applications is essential. Aneka provides built-in capabilities for file management in a distributed infrastructure, and the Task Programming Model transparently leverages these capabilities. Any model based on the *WorkUnit* and *ApplicationBase* classes has built-in support for file

```
// File: GaussApp.cs
using System;
using System.Threading;

using Aneka.Entity;
using Aneka.Tasks;

namespace GaussSample
{
   /// <summary>
   /// Class GaussApp. Defines the coordination logic of the
   /// distributed application for computing the gaussian distribution.
   /// </summary>
   public class GaussApp
   {
      /// <summary>
      /// Semaphore used to make the main thread wait while
      /// all the tasks are terminated.
      /// </summary>
      private ManualResetEvent semaphore;
      /// <summary>
      /// Counter of the running tasks.
      /// </summary>
      private int taskCount = 0;
      /// <summary>
      /// Aneka application instance.
      /// </summary>
      private AnekaApplication<AnekaTask, TaskManager> app;

      /// <summary>
      /// Main entry point for the application.
      /// </summary>
      /// <param name="args">An array of strings containing the command line.</param>
      public static void Main(string[] args)
      {
         try
         {
            // initialize the logging system
            Logger.Start();

            string confFile = "conf.xml";
            if (args.Length > 0)
            {
               confFile = args[0];
            }
            // get an instance of the Configuration class from file
            Configuration conf = Configuration.GetConfiguration(confFile);
            // create an instance of the GaussApp and starts its execution
            // with the given configuration instance
            GaussApp application = new GaussApp();
            application.SubmitApplication(conf);
         }
```

LISTING 7.6

GaussApplication.

```
catch(Exception ex)
{
   IOUtil.DumpErrorReport(ex, "Fatal error while executing application.");
}
finally
{
   // terminate the logging thread
   Logger.Stop();

   }
}

/// <summary>
/// Application submission method.
/// </summary>
/// <param name="conf">Application configuration.</param>
public void SubmitApplication(Configuration conf)
{
   // initialize the semaphore and the number of
   // task initially submitted
   this.semaphore = new ManualResetEvent(false);
   this.taskCount = 400;

   // specify that the submission of task is dynamic
   conf.SingleSubmission = false;
   this.app = new AnekaApplication<Task,TaskManager>(conf);
   // attach methods to the event handler that notify the client code
   // when tasks are completed or failed
   this.app.WorkUnitFailed +=
        new EventHandler<WorkUnitEventArgs<AnekaTask>>(this.OnWorkUnitFailed);
   this.app.WorkUnitFinished +=
        new EventHandler<WorkUnitEventArgs<AnekaTask>>(this.OnWorkUnitFinished);

   // attach the method OnAppFinished to the Finished event so we can capture
   // the application termination condition, this event will be fired in case of
   // both static application submission or dynamic application submission
   app.Finished += new EventHandler<ApplicationEventArgs>(this.OnAppFinished);

   for(int i=0; i<400; i++)
   {
      GaussTask gauss = new GaussTask();
      gauss.X = i;
      AnekaTask task = newAnekaTask(gauss);
      // add the task to the bag of work units to submit
      app.AddWorkunit(task);
   }
   // submit the entire bag
   app.SubmitExecution();

   // wait until signaled, once the thread is signaled the application is completed
   this.semaphore.Wait();
}
```

LISTING 7.6

(Continued)

```
/// <summary>
/// Event handler for task failure.
/// </summary>
/// <param name="sender">Event source: the application instance.</param>
/// <param name="args">Event arguments.</param>
private void OnWorkUnitFailed(object sender, WorkUnitEventArgs<AnekaTask> args)
{
   // do nothing, we are not interested in task failure at the moment
   // just dump to console the failure.
   if (args.WorkUnit != null)
   {
      Exception error = args.WorkUnit.Exception;
      Console.WriteLine("Task {0} failed - Exception: {1}",
      args.WorkUnit.Name,
      (error == null ? "[Not given]" : error.Message);
   }
   // we do not have to synchronize this operation because
   // events handlers are run all in the same thread, and there
   // will not be other threads updating this variable

   this.taskCount--;
   if (this.taskCount == 0)
   {
      this.app.StopExecution();
   }
}
/// <summary>
/// Event handler for task completion.
/// </summary>
/// <param name="sender">Event source: the application instance.</param>
/// <param name="args">Event arguments.</param>
private void OnWorkUnitFinished(object sender, WorkUnitEventArgs<AnekaTask> args)
{
   // we do not have to synchronize this operation because
   // events handlers are run all in the same thread, and there
   // will not be other threads updating this variable
   this.taskCount--;

   // if the task is completed for sure we have a WorkUnit instance
   // and we do not need to check as we did before.
   GaussTask gauss = (GaussTask) args.WorkUnit.Task;
   // we check whether it is an initially submitted task or a task
   // that we submitted as a reaction to the completion of another task
   if (task.X - Math.Abs(task.X) == 0)
   {
      // ok it was an original task, then we increment of 0.5 the
      // value of X and submit another task
      GaussTask fraq = GaussTask();
      fraq.X = gauss.X + 0.5;
      AnekaTask task =new AnekaTask(fraq);
```

LISTING 7.6

(Continued)

```
                this.taskCount++;
                // we call the ExecuteWorkUnit method that is used
                // for dynamic submission
                app.ExecuteWorkUnit(task);
            }
            Console.WriteLine("Task {0} completed - [X:{1},Y:{2}]",
                              args.WorkUnit.Name, gauss.X, gauss.Y);
            if (this.taskCount == 0)
            {
                this.app.StopExecution();
            }
        }
        /// <summary>
        /// Event handler for the application termination.
        /// </summary>
        /// <param name="sender">Event source: the application instance.</param>
        /// <param name="args">Event arguments.</param>
        private void OnWorkUnitFinished(object sender, ApplicationEventArgs args)
        {
            // unblock the main thread, because we have identified the termination
            // of the application
            this.semaphore.Set();
        }
    }
}
```

LISTING 7.6

(Continued)

management. It is possible to provide input files that are common to all the *WorkUnit* instances, through the *ApplicationBase.SharedFiles* collection, and instance-specific input and output files by leveraging the *WorkUnit.InputFiles* and *WorkUnit.OutputFiles* collections.

A fundamental component for the management of files is the *FileData* class, which constitutes the logic representation of physical files, as defined in the *Aneka.Data.Entity* namespace (*Aneka.Data.dll*). A *FileData* instance provides information about a file:

- Its nature: whether it is a shared file, an input file, or an output file
- Its path both in the local and in the remote file system, including a different name
- A collection of attributes that provides other information (such as the final destination of the file or whether the file is transient or not, etc.)

Using the *FileData* class, the user specifies the file dependencies of tasks and the application, and the Aneka APIs will automatically transfer them to and from the Aneka Cloud when needed. Aneka allows specifying of both local and remote files stored on FTP servers or Amazon S3. A *FileData* instance is identified by three elements: an owner, a name, and a type. By means of the corresponding ID, the owner identifies which is the computing element that needs the file: application instance or work unit. The type specifies whether the file is shared or an input or output file. The name represents the name of the corresponding physical file.

Listing 7.7 demonstrates how to add file dependencies to the application and to tasks. It is possible to add both *FileData* instances, thus having more control of the information attached to the file, or to use more intuitive approaches that simply require the name and the type of the file.

The general interaction flow for file management is as follows:

- Once the application is submitted, the shared files are staged into the Aneka Cloud.
- If the file is local it will be searched into the directory location identified by the property *Configuration.Workspace*; if the file is remote, the specific configuration settings mapped by the *FileData.StorageBucketId* property will be used to access the remote server and stage in the file.
- If there is any failure in staging input files, the application will be terminated with an error.
- For each of the tasks belonging to the application, the corresponding input files are staged into the Aneka Cloud, as is done for shared files.
- Once the task is dispatched for execution to a remote node, the runtime will transfer all the shared files of the application and the input files of the task into the working directory of the task and eventually get renamed if the *FileData.VirtualPath* property is not null.
- After the execution of the task, the runtime will look for all the files that have been added into the *WorkUnit.OutputFiles* collection. If not null, the value of the *FileData.VirtualPath* property will be used to locate the files; otherwise, the *FileData.FileName* property will be the reference. All the files that do not contain the *FileAttributes.Optional* attribute need to be present; otherwise, the execution of the task is classified as a failure.
- Despite the successful execution or the failure of a task, the runtime tries to collect and move to their respective destinations all the files that are found. Files that contain the *FileAttributes.Local* attribute are moved to the local machine from where the application is saved and stored in the directory location identified by the property *Configuration.Workspace*. Files that have a *StorageBucketId* property set will be staged out to the corresponding remote server.

The infrastructure for file management provides a transparent and extensible service for the movement of data. The architecture for file management is based on the concept of factories and storage buckets. A *factory*, namely *IFileTransferFactory*, is a component that is used to abstract the creation of client and server components for file transfer so that the entire architecture can work with interfaces rather than specific implementations. *Storage buckets* are collections of string properties that are used to specialize these components through configuration files. Storage buckets are specified by the user, either by the configuration file or by programmatically adding this information to the configuration object, before submitting the application. Factories are used by the *StorageService* to pull in remote input and shared files and to pull out remote output files.

Listing 7.8 shows a sample configuration file containing the settings required to access the remote files through the FTP and S3 protocols. Within the *<Groups>* tag, there is a specific group named *StorageBuckets*; this group maintains the configuration settings for each storage bucket that needs to be used in for file transfer. Each *<Group>* tag represents a storage bucket and the *name* property contains the values referenced in the *FileData.StorageBucketId* property. The content of each of these groups is specific to the type of storage bucket used, which is identified by the *Scheme* property. These values are used by the specific implementations for the FTP and S3 protocols to access the remote servers and transfer the files.

```
// get an instance of the Configuration class from file
Configuration conf = Configuration.GetConfiguration("conf.xml");
AnekaApplication<Task,TaskManager>app =new AnekaApplication<Task,TaskManager>(conf);

// attach shared files with different methods by using the FileData class and directly
// using the API provided by the AnekaApplication class

// create a local shared file whose local and remote name is "pi.tab"
FileDatapiTab = newFileData("pi.tab",FileDataType.Shared);
app.AddSharedFile(piTab);
// once the file is added to the collection of shared files, its OwnerId property
// references app.Id

// create a remote shared file by specifying the attributes whose name is "pi.dat"
FileDatapiDat = newFileData("pi.dat",FileDataType.Shared, FileAttributes.None);
// the StorageBucketId property points a specific configuration section that is
// used to store the information for retrieving the file from the remote server
piDat.StorageBucketId = "FTPStore";
app.AddSharedFile(piDat);
// once the file is added to the collection of shared files, its OwnerId property
// references app.Id

// adds a local shared file
app.AddSharedFile("pi.xml");

for(int i=0; i<400; i++)
{
   GaussTask gauss = new GaussTask();
   gauss.X = i;
   AnekaTask task = newAnekaTask(gauss);

   // adds a local input file for the current task whose name is "<i>.txt"
   // where <i> is the value of the loop variable
   FileData input = new FileData(string.Format("{0}.txt", i);
   FileDataType.Input, FileAttributes.Local);
   // once transferred to the remote node, the file will have the name
   // "input.txt". Since tasks are executed in separate directories there
   // will be no name clashing
   input.VirtualPath ="input.txt";
   task.AddFile(input);
   // once the file is added to the task, it will be stored in the InputFiles
   // collection and its OwnerId property will referenced task.Id

   // adds anoutput file for the current task whose name is "out.txt" that will
   // be stored on S3
```

LISTING 7.7

File dependencies management.

```
    FileData output =new FileData("out.txt", FileDataType.Input, FileAttributes.None);
    // once transferred to the remote server, the file will have the name
    // "<i>.out"where <i> is the value of the loop variable. In this way we
    // easily avoid name clashing while storing output files into a single
    // directory
    output.VirtualPath =string.Format("{0}.out", i);
    output.StorageBucketId = "S3Store";
    task.AddFile(output);
    // once the file is added to the task, it will be stored in the InputFiles
    // collection and its OwnerId property will referenced task.Id

    // adds a localoutput file for the current task whose name is "trace.log".
    // The file is optional, this means that if after the execution of the task the file

    // is not present no exception or task failure will be risen.
    FileData trace =new FileData("trace.log", FileDataType.Input,
    FileAttributes.Local | FileAttributes.Optional);
    // once transferred to the local machine, the file will have the name
    // "<i>.log"where <i> is the value of the loop variable. In this way we
    // easily avoid name clashing while storing output files into a single
    // directory
    trace.VirtualPath =string.Format("{0}.log", i);
    task.AddFile(trace);
    // once the file is added to the task, it will be stored in the InputFiles
    // collection and its OwnerId property will referenced task.Id

    // add the task to the bag of work units to submit
    app.AddWorkunit(task);
}

// submit the entire bag, files will be moved automatically by the Aneka APIs
app.SubmitExecution();
```

LISTING 7.7

(Continued)

7.3.2.4 Task libraries

Aneka provides a set of ready-to-use tasks for performing the most basic operations for remote file management. These tasks are part of the *Aneka.Tasks.BaseTasks* namespace, which is part of the *Aneka.Tasks.dll* library. The following operations are implemented:

- *File copy.* The *LocalCopyTask* performs the copy of a file on the remote node; it takes a file as input and produces a copy of it under a different name or path.
- *Legacy application execution.* The *ExecuteTask* allows executing external and legacy applications by using the *System.Diagnostics.Process* class. It requires the location of the executable file to run, and it is also possible to specify command-line parameters. *ExecuteTask* also collects the standard error and standard output produced by the execution of the application.

```
<?xmlversion="1.0"encoding="utf-8"?>
<Aneka>
  <UseFileTransfer value="true" />
  <Workspace value="." />
  <SingleSubmission value="false" />
  <ResubmitMode value="Manual" />
  <PollingTime value="1000" />
  <LogMessages value="true" />
  <SchedulerUri value="tcp://localhost:9090/Aneka"/>
  <UserCredential type="Aneka.Security.UserCredentials" assembly="Aneka.dll">
    <UserCredentials username="Administrator" password=""/>
  </UserCredential>
  <Groups>
    <Group name="StorageBuckets">
      <Groups>
        <Group name="FTPStore">
          <Property name="Scheme" value="ftp"/>
          <Property name="Host" value="www.remoteftp.org"/>
          <Property name="Port" value="21"/>
          <Property name="Username" value="anonymous"/>
          <Property name="Password" value="nil"/>
        </Group>
        <Group name="S3Store">
          <Propertyname="Scheme" value="S3"/>
          <Propertyname="Host"value="www.remoteftp.org"/>
          <Propertyname="Port"value="21"/>
          <Propertyname="Username"value="anonymous"/>
          <Propertyname="Password"value="nil"/>
        </Group>
      </Groups>
    </Group>
  </Groups>
</Aneka>
```

LISTING 7.8

Aneka application configuration file.

- *Substitute operation.* The *SubstituteTask* performs a search-and-replace operation within a given file by saving the resulting file under a different name. It is possible to specify a collection of string-based name-value pairs representing the strings to search together with their corresponding replacements.
- *File deletion.* The *DeleteTask* deletes a file that is accessible through the file system on the remote node.
- *Timed delay.* The *WaitTask* introduces a timed delay. This task can be used in several scenarios; for example, it can be used for profiling or simply for simulation of the execution. In addition, it can also be used to introduce a pause between the execution of two applications if needed.

- *Task composition.* The *CompositeTask* implements the composite pattern[3] and allows expressing a task as a composition of multiple tasks that are executed in sequence. This task is very useful to perform complex tasks involving the combination of operations implemented in other tasks.

The base task library does not provide any support for data transfer since this operation is automatically performed by the infrastructure when needed. Besides these simple tasks, the Aneka API allows for the creation of any user-defined task by simply implementing the *ITask* interface and supporting object serialization.

7.3.2.5 Web services integration

Aneka provides integration with other technologies and applications by means of Web services, which allow some of the services hosted in the Aneka Cloud to be accessible in platform-independent fashion. Among these, the task submission Web service allows third-party applications to submit tasks as they happen in traditional computing grids.

The task submission Web service is an additional component that can be deployed in any ASP.NET Web server and that exposes a simple interface for job submission, which is compliant with the Aneka Application Model. The task Web service provides an interface that is more compliant with the traditional way fostered by grid computing. Therefore, the new concept of the term *job*, which is a collection of predefined tasks, is introduced. The reference scenario for Web-based submission is depicted in Figure 7.9. Users create a distributed application instance on the cloud and, within the context of this application, they can submit jobs querying the status of the application or a single job. It is up to the users to then terminate the application when all the jobs are completed or to abort it if there is no need to complete job execution.

Jobs can be created by putting together the tasks defined in the basic task library. Operations supported through the Web service interface are the following:

- Local file copy on the remote node
- File deletion
- Legacy application execution through the common shell services
- Parameter substitution

It is also possible specify input and output files for each job. The only restriction in this case is that both input and output files need to reside in remote FTP servers. This enables Aneka to automatically stage the files from these servers without user intervention. Figure 7.10 gives detailed information about the object model exposed through the Web service for job submission.

Traditional grid technologies such as the Gridbus Broker [15] and the Workflow Engine [70] can make use of a task Web service to submit their tasks for execution on Cloud nodes managed by Aneka.

[3]In software engineering, the composite pattern is a software design pattern that allows expressing a combination of components as a single component. The advantage of using such a pattern resides in creating a software infrastructure that allows forwarding the execution of an operation to a group of objects by treating it as single unit and in a completely transparent manner. Reference: E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Software Design*, Addison-Wesley, 1995, ISBN: 0201633612.

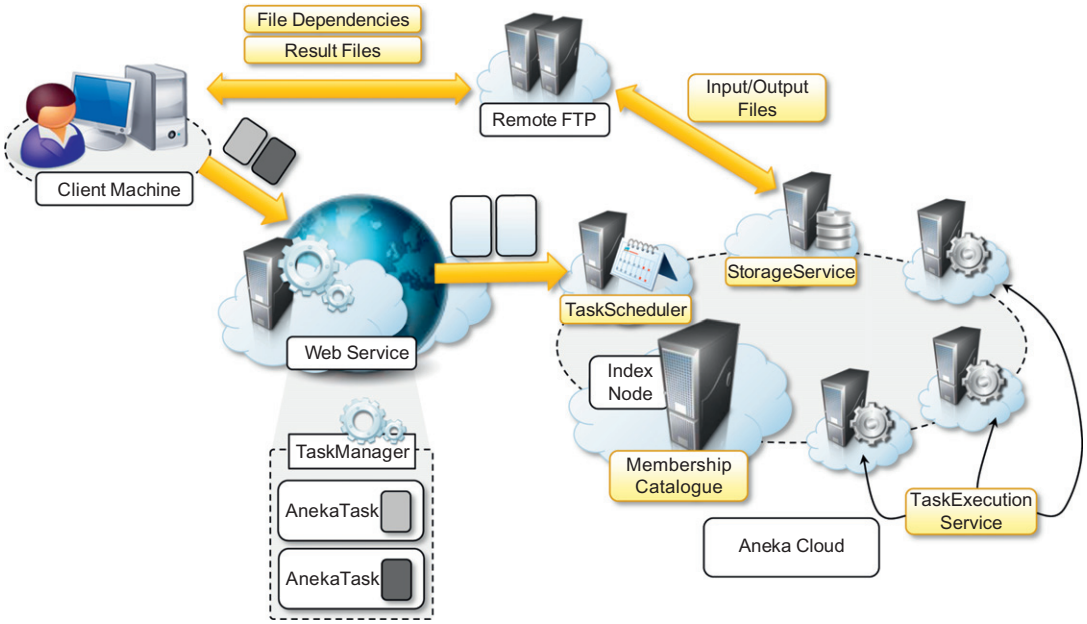

FIGURE 7.9

Web service submission scenario.

7.3.3 Developing a parameter sweep application

Aneka integrates support for parameter-sweeping applications on top of the task model by means of a collection of client components that allow developers to quickly prototype applications through either programming APIs or graphical user interfaces (GUIs). The set of abstractions and tools supporting the development of parameter sweep applications constitutes the *Parameter Sweep Model (PSM)*.

The PSM is organized into several namespaces under the common root *Aneka.PSM*. More precisely:

- *Aneka.PSM.Core (Aneka.PSM.Core.dll)* contains the base classes for defining a template task and the client components managing the generation of tasks, given the set of parameters.
- *Aneka.PSM.Workbench (Aneka.PSM.Workbench.exe)* and *Aneka.PSM.Wizard (Aneka.PSM.Wizard.dll)* contain the user interface support for designing and monitoring parameter sweep applications. Mostly they contain the classes and components required by the *Design Explorer*, which is the main GUI for developing parameter sweep applications.
- *Aneka.PSM.Console (Aneka.PSM.Console.exe)* contains the components and classes supporting the execution of parameter sweep applications in console mode.

These namespaces define the support for developing and controlling parameter sweep applications on top of Aneka.

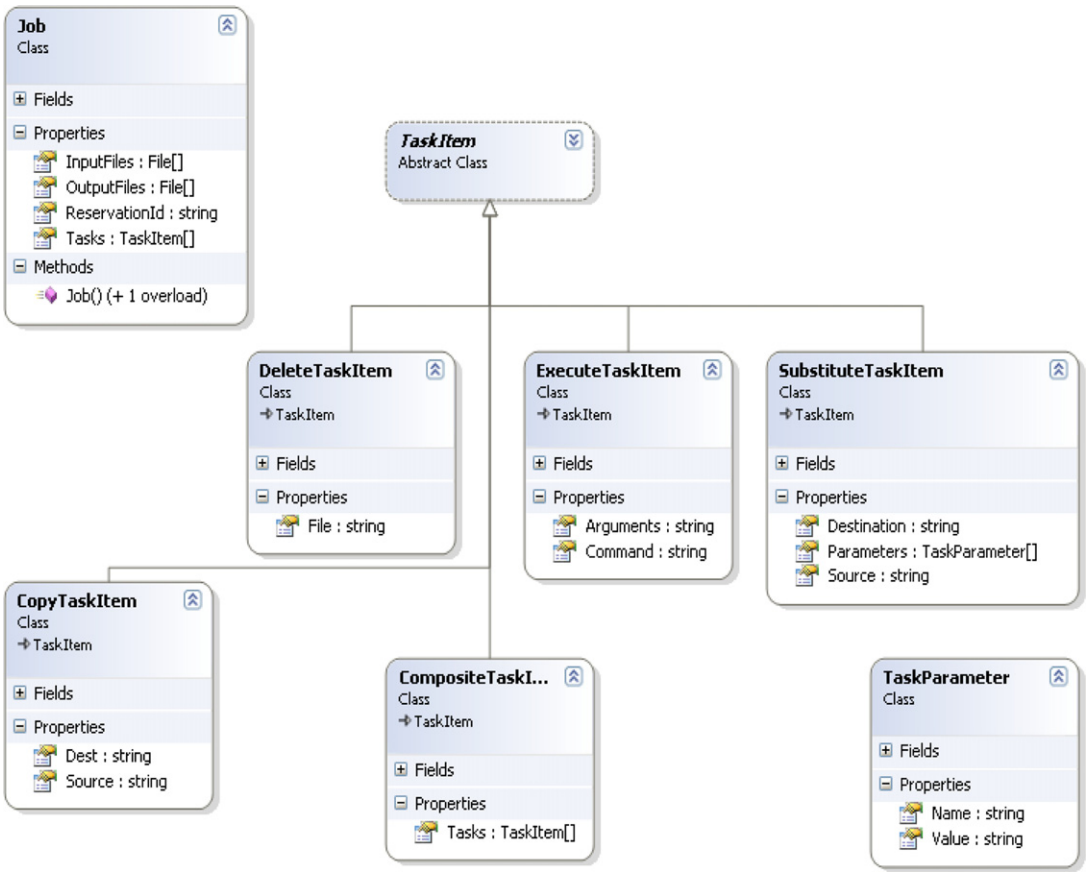

FIGURE 7.10

Job object model.

7.3.3.1 Object model

The fundamental elements of the Parameter Sweep Model are defined in the *Aneka.PSM.Core* namespace. This model introduces the concept of *job (Aneka.PSM.Core.PSMJobInfo)*, which identifies a parameter sweep application. A job comprises file dependencies and parameter definitions, together with their admissible domains, and the definition of the template task. Figure 7.11 shows the most relevant components of the object model.

The root component for application design is the *PSMJobInfo* class, which contains information about shared files and input and output files (*PSMFileInfo*). In accordance with the Aneka Application Model, shared files are common to all the instances of the template task, whereas input

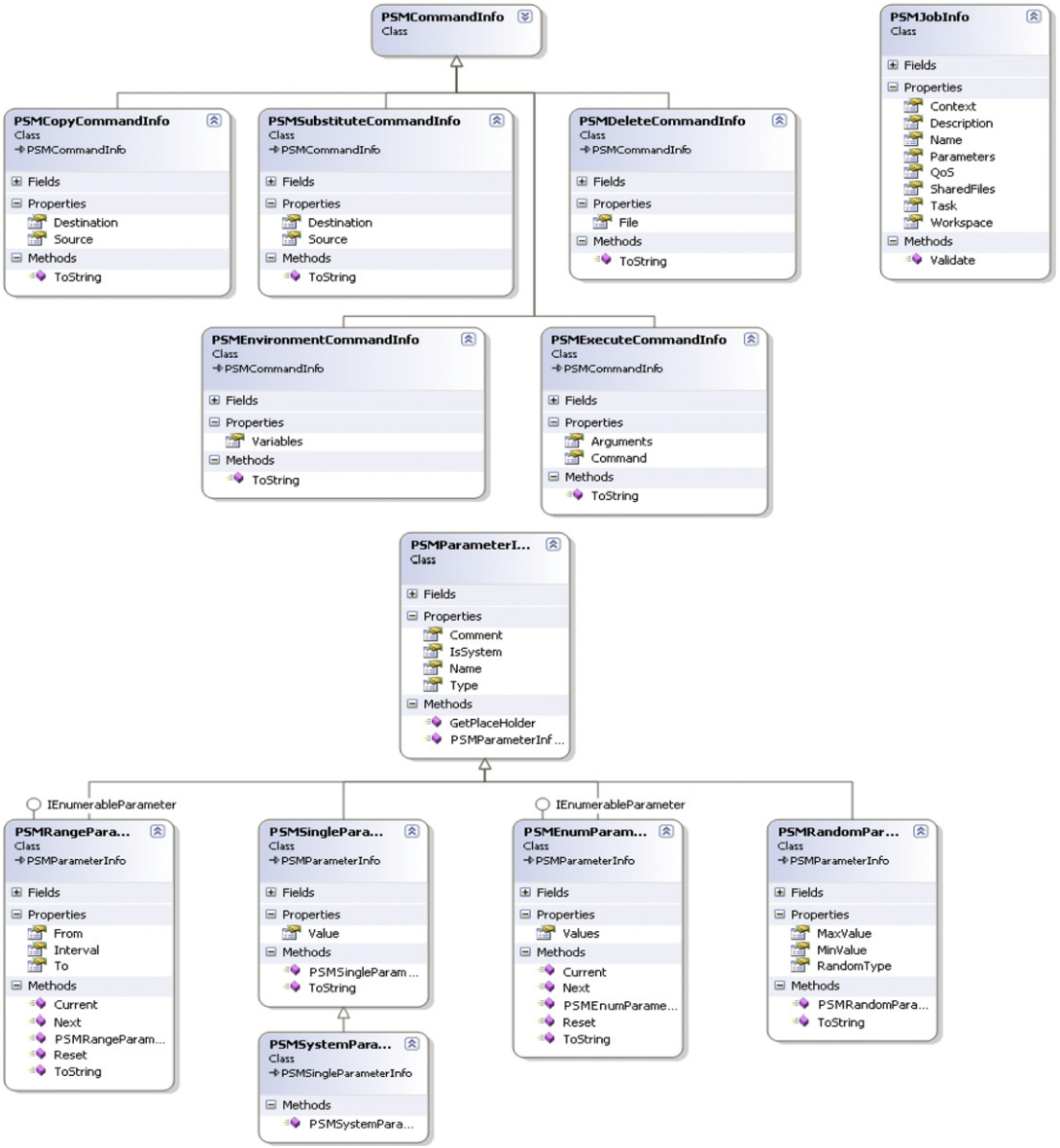

FIGURE 7.11

PSM object model (relevant classes).

and output files are specific to a given task instance. Therefore, these files can be expressed as a function of the parameters. Currently, it is possible to specify five different types of parameters:

- *Constant parameter* (*PSMSingleParameterInfo*). This parameter identifies a specific value that is set at design time and will not change during the execution of the application.
- *Range parameter* (*PSMRangeParameterInfo*). This parameter allows defining a range of allowed values, which might be integer or real. The parameter identifies a domain composed of discrete values and requires the specification of a lower bound, an upper bound, and a step for the generation of all the admissible values.
- *Random parameter* (*PSMRandomParameterInfo*). This parameter allows the generation of a random value in between a given range defined by a lower and an upper bound. The value generated is real.
- *Enumeration parameter* (*PSMEnumParameterInfo*). This parameter allows for specifying a discrete set of values of any type. It is useful to specify discrete sets that are not based on numeric values.
- *System parameter* (*PSMSystemParameterInfo*). This parameter allows for mapping a specific value that will be substituted at runtime while the task instance is executed on the remote node.

Other than these parameters, the object model reserves special parameters that are used to identify specific values of the PSM object model, such as the task identifier and other data. Parameters have access to the execution environment by means of an execution context (*PSMContext*) that is responsible for providing default and runtime values. The task template is defined as a collection of commands (*PSMCommandInfo*), which replicate and extend the features available in the base task library. The available commands for composing the task template perform the following operations:

- Local file copy on the remote node (*PSMCopyCommandInfo*)
- Remote file deletion (*PSMDeleteCommandInfo*)
- Execution of programs through the shell (*PSMExecuteCommandInfo*)
- Environment variable setting on the remote node (*PSMEnvironmentCommandInfo*)
- String pattern replacement within files (*PSMSubstituteCommandInfo*)

By following the same approach described for the creation of tasks, it is possible to define the task template by composing these basic blocks. All the properties exposed by these commands can include the previously defined parameters, the values of which will be provided during the generation of the task instances.

A parameter sweep application is executed by means of a job manager (*IJobManager*), which interfaces the developer with the underlying APIs of the task model. Figure 7.12 shows the relationships among the PSM APIs, with a specific reference to the job manager, and the task model APIs.

Through the *IJobManager* interface it is possible to specify user credentials and configuration for interacting with the Aneka middleware. The implementation of *IJobManager* will then create a corresponding Aneka application instance and leverage the task model API to submit all the task instances generated from the template task. The interface also exposes facilities for controlling and monitoring the execution of the parameter sweep application as well as support for registering the statistics about the application.

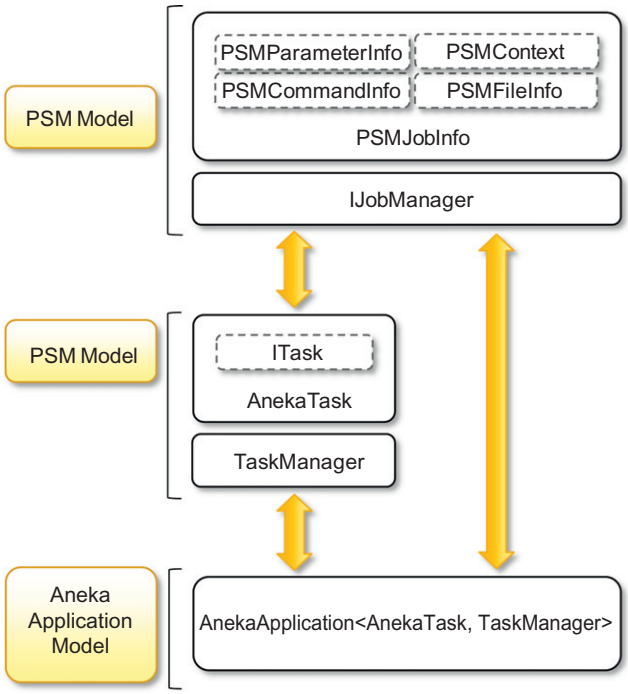

FIGURE 7.12

Parameter sweep model APIs.

7.3.3.2 Development and monitoring tools

The core libraries allow developers to directly program parameter sweep applications and embed them into other applications. Additional tools simplify design and development of parameter sweep applications by providing support for visual design of the applications and interactive and noninteractive application execution. These tools are the *Aneka Design Explorer* and the *Aneka PSM Console*.

The *Aneka Design Explorer* is an integrated visual environment for quickly prototyping parameter sweep applications, executing them, and monitoring their status. It provides a simple wizard that helps the user visually define any aspect of parameter sweep applications, such as file dependencies and result files, parameters, and template tasks. The environment also provides a collection of components that help users monitor application execution, aggregate statistics about application execution, gain detailed task transition monitoring, and gain extensive access to application logs.

The *Aneka PSM Console* is a command-line utility designed to run parameter sweep applications in noninteractive mode. The console offers a simplified interface for running applications with essential features for monitoring their execution. With respect to the Design Explorer, the console offers less support for keeping and visualizing aggregate statistics, but it exposes the same data in a more simplified textual form.

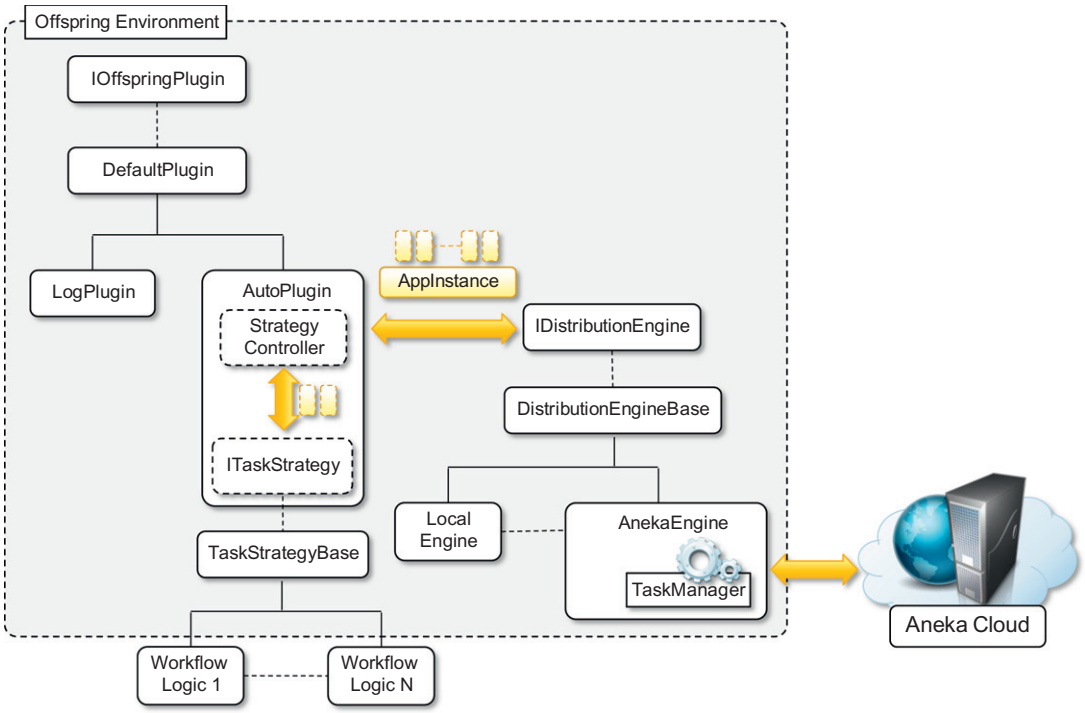

FIGURE 7.13

Offspring architecture.

7.3.4 Managing workflows

Support for workflow in Aneka is not native but is obtained with plug-ins that allow client-based workflow managers to submit tasks to Aneka. Currently, two different workflow managers can leverage Aneka for task execution: the *Workflow Engine* [70] and *Offspring* [71]. The former leverages the task submission Web service exposed by Aneka; the latter directly interacts with the Aneka programming APIs. The Workflow Engine plug-in for Aneka constitutes an example of the integration capabilities offered by the framework, which allows client applications developed with any technology and language to leverage Aneka for task execution. The integration developed for Offspring constitutes another example of how it is possible to construct another programming model on top of the existing APIs available in the framework. Therefore, we discuss this solution in more detail.

Figure 7.13 describes the Offspring architecture. The system is composed of two types of components: plug-ins and a distribution engine. Plug-ins are used to enrich the environment of features; the distribution engine represents access to the distributed computing infrastructure leveraged for task execution. Among the available plug-ins, *AutoPlugin* provides facilities for the definition of workflows in terms of strategies. A *strategy* generates the tasks that are submitted for execution and defines the logic, in terms of sequencing, coordination, and dependencies, used to submit the task through the engine. A specific component, the *StrategyController*, decouples the strategies

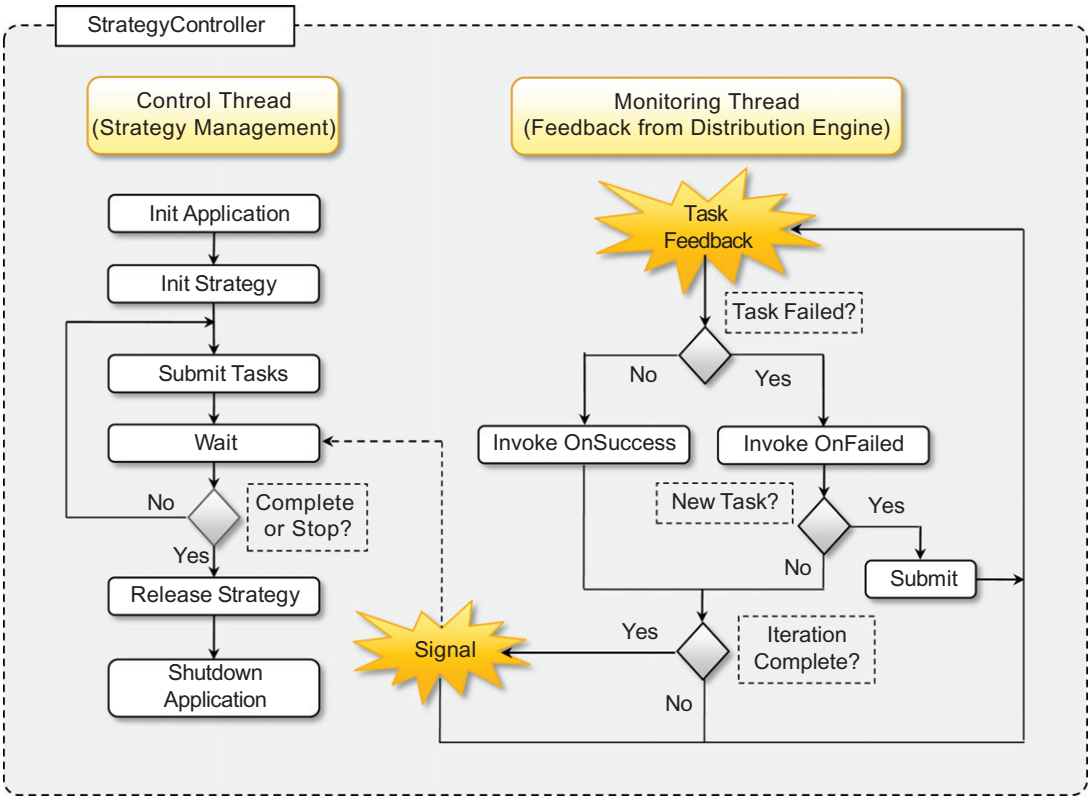

FIGURE 7.14

Workflow coordination.

from the distribution engine; therefore, strategies can be defined independently of the specific middleware used. The connection with Aneka is realized through the *AnekaEngine*, which implements the operations of *IDistributionEngine* for the Aneka middleware and relies on the services exposed by the task model programming APIs.

The system allows for the execution of a dynamic workflow, the structure of which is defined as the workflow executes. Two different types of tasks can be defined: native tasks and legacy tasks. *Native tasks* are completely implemented in managed code. *Legacy tasks* manage file dependencies and wrap all the data necessary for the execution of legacy programs on a remote node. Furthermore, a strategy may define shared file dependencies that are necessary to all the tasks generated by the workflow. The dependencies among tasks are implicitly defined by the execution of the strategy by the *StrategyController* and the events fired by the distributed engine.

Figure 7.14 describes the interactions among these components. Two main execution threads control the execution of a strategy. A *control thread* manages the execution of the strategy, whereas a *monitoring thread* collects the feedback from the distribution engine and allows for the dynamic

reaction of the strategy to the execution of previously submitted tasks. From the workflow developer's point of view, the logic is quite simple. The execution of a strategy is composed of three macro steps: setup, execution, and finalization. The first step involves the setup of the strategy and the application mapping it. Correspondingly, the finalization step is in charge of releasing all the internal resources allocated by the strategy and shutting down the application. The core of the workflow execution resides in the execution step, which is broken down into a set of iterations. During each of the iterations a collection of tasks is submitted; these tasks do not have dependencies from each other and can be executed in parallel. As soon as a task completes or fails, the strategy is queried to see whether a new set of tasks needs to be executed. In this way, dependencies among tasks are implemented. If there are more tasks to be executed, they are submitted and the controller waits for feedback from the engine; otherwise, an iteration of the strategy is completed. At the end of each iteration, the controller checks to see whether the strategy has completed the execution, and in this case, the finalization step is performed.

The *AnekaEngine* creates an instance of the *AnekaApplication* class for each execution of a strategy and configures the template class with a specific implementation of the *TaskManager*, which overrides the behavior implemented for file management and optimizes the staging of output files. To support the implementation of workflows without any dependency from the distribution engine, the application configuration settings are controlled by the distribution engine and shared among all the strategies executed through the engine.

SUMMARY

This chapter introduced the concept of task-based programming and provided an overview of the technologies supporting the development of distributed applications based on the concept of tasks. Task-based programming constitutes the most intuitive approach to distributing the computation of an application over a set of nodes. The main abstraction of task-based programming is the concept of a *task*, which represents a group of operations that can be isolated and executed as a single unit. A task can be a simple program that is executed through the shell or a more complex piece of code requiring a specific runtime environment to execute. Quite often, tasks require input files for their execution and produce output files as a result. According to this model, an application is expressed as a collection of tasks; the way these tasks are interrelated and their specific nature and characteristics differentiate the various models that are an expression of task-based programming.

Traditionally, the task-based programming model has been successfully used in the development of distributed applications in many areas. We identified three major computing categories in which the task model can be utilized. *High-performance computing (HPC)* refers to the use of distributed computing facilities for solving problems that need large computing power. Common HPC applications feature a large collection of compute-intensive tasks, the duration of which is relatively short. *High-throughput computing (HTC)* identifies scenarios in which distributed computing facilities are used to support the execution of applications that need large computing power for a long period of time. Tasks may not be numerous, but they have a long duration, and infrastructure reliability becomes fundamental. *Many-task computing (MTC)* is the latest emergent trend and identifies a heterogeneous set of applications and requirements for applications, which fills the gap between HPC and HTC.

We have briefly reviewed common models related to task programming. *Embarrassingly parallel* applications are composed of a collection of tasks that do not relate to each other, can be executed in any order, and do not require co-allocation. *Parameter sweep* applications are a special instance of the embarrassingly parallel model. They are characterized by a collection of independent tasks that are automatically generated from a template task by varying the combination of parameter values. In this case, the executed task is the same in terms of computation logic, but it operates on different data. Therefore, a parameter sweep application can also be considered an expression of the *single program, multiple data (SPMD)* model. *MPI* applications are characterized by a collection of tasks that need to be executed all together and that exchange data by message passing. Even though the program executed by an MPI application tasks might be the same, it is quite common to provide an implementation logic that differentiates the behavior of each task according to its rank. *Workflow* applications are characterized by a collection of tasks for which the dependencies can be expressed in terms of a directed acyclic graph. Dependencies are mostly represented by files, which are produced as output of a specific task, and are required for the computation of the dependent tasks. The nature of the tasks and the kind of computation performed by each task differ generally.

We introduced the task model and the services implemented in Aneka that support task-based programming as a practical example of a framework that enables the development and execution of distributed applications based on tasks. The task model comprises a set of services (directory, scheduling, execution, and storage) of which the coordination constitutes runtime support for the execution of embarrassingly parallel applications. The fundamental features of the task model in terms of task definition, submission, execution, and file dependencies management was demonstrated with a practical example. On top of this infrastructure, client-side components and integration with other technologies allow providers to support parameter sweep and workflow applications. Parameter sweep applications are realized through the Parameter Sweep Model (PSM), which is characterized by a collection of client-side components that provide different, and more suitable, interfaces for this kind of application. Workflow applications are not natively supported by Aneka, but integration with other technologies allows us to leverage Aneka for workflow execution. For example, a plug-in using the Aneka task submission Web service allows the Workflow Engine to use Aneka as a back-end for workflow execution. The Aneka distribution engine implemented in Offspring provides another example of how it is possible to quickly prototype another programming model (in this case, a workflow-based model) by leveraging the base APIs of the task model.

Review questions

1. What is a task? How does task computing relate to distributed computing?
2. List and explain the computing categories that relate to task computing.
3. What are the main functionalities of a framework that supports task computing?
4. List some of the most popular frameworks for task computing.
5. What does the term *bag of tasks* mean?
6. Give an example of a parameter sweep application.
7. What is MPI? What are its main characteristics?

8. What is a workflow? What are the additional properties of this application model with respect to an embarrassingly parallel application?
9. Describe the reference model of a workflow management system.
10. How does Aneka support task computing?
11. What are the main components of the Task Programming Model?
12. Discuss the differences between ITask and AnekaTask.
13. Discuss the differences between static and dynamic task submission.
14. Discuss the facilities and the general architecture provided by Aneka for movement of data for task-based applications.
15. How it is possible to run a legacy application using the Task Programming Model?
16. Does Aneka provide any feature for leveraging the Task Programming Model from other technologies and platforms?
17. Using the Task Programming Model, design and implement a simple application that performs the discrete computation of the integral according to the method proposed by Riemann[4] of a given function over a specified interval.
18. What are the features provided by Aneka for the execution of parameter sweep applications?
19. Does Aneka provide native support for the execution of workflows?
20. By taking as a reference the Montage workflow described in Figure 7.6, design a sketch of the control flow of an Offspring strategy that can be used to execute a workflow on Aneka.

[4] http://en.wikipedia.org/wiki/Riemann_inegral.

CHAPTER

Data-Intensive Computing
MapReduce Programming

8

Data-intensive computing focuses on aa 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 fact, 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 using the 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 amounts of information. To facilitate the classification and search, relevant bits of information, called *metadata*, are attached to datasets.

Data-intensive computations occur in many application domains. Computational science is one of the most popular ones. People conducting 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; 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. Customer data for any telecom company would easily be in the range of 10−100 terabytes. This volume of information is not only processed to generate billing statements, but it is also mined to identify scenarios, trends, and patterns that help these companies provide better service. Moreover, it is reported that U.S. handset mobile traffic has reached 8 petabytes per month and it is expected to grow up to 327 petabytes per month by 2015.[1] The scale of petabytes is even more common when we consider 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.[2] Social networking and gaming are two other sectors in which 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 reach to 36 petabytes.[3] Zynga, *a* social gaming platform, moves 1 petabyte of data daily and it has been reported to add 1,000 servers every week to store the data generated by games like Farmville and Frontierville.[4]

8.1.1 Characterizing data-intensive computations

Data-intensive applications 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 on 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

[1]Coda Research Consultancy, www.codaresearch.co.uk/usmobileinternet/index.htm.
[2]Google's Blog, http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html.
[3]OSCON 2010, David Recordon (Senior Open Programs Manager, Facebook): Today's LAMP Stack, Keynote Speech. Available at www.oscon.com/oscon2010/public/schedule/speaker/2442.
[4]http://techcrunch.com/2010/09/22/zynga-moves-1-petabyte-of-data-daily-adds-1000-servers-a-week/.

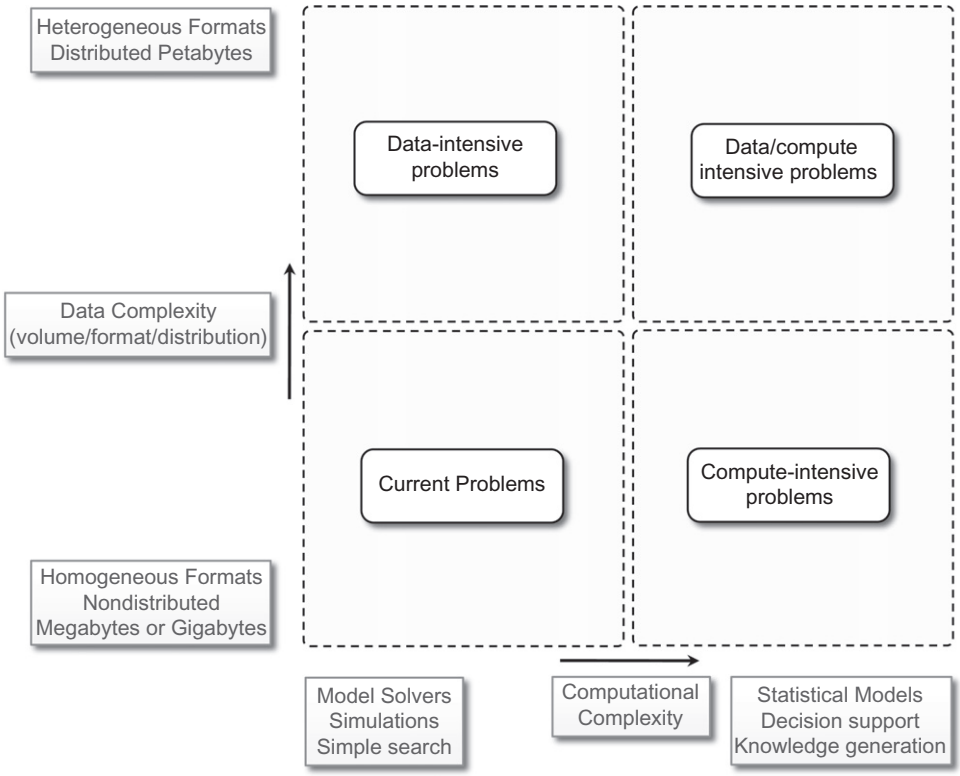

FIGURE 8.1

Data-intensive research issues.

- Advances in high-performance computing platforms aimed at providing a better support for accessing in-memory multiterabyte data structures
- High-performance, highly reliable, petascale distributed file systems
- Data signature-generation techniques for data reduction and rapid processing
- New approaches to software mobility for delivering algorithms that are able to move the computation to where the data are located
- Specialized hybrid interconnection architectures that provide better support for filtering multigigabyte datastreams coming from high-speed networks and scientific instruments
- Flexible and high-performance software integration techniques that facilitate the combination of software modules running on different platforms to quickly form analytical pipelines

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.

8.1.3.1 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 for 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). On that occasion, the remote visualization of large and complex scientific datasets (a high-resolution magnetic resonance image, or MRI, scan of the human brain) was set up between the Pittsburgh Supercomputing Center (PSC) and Albuquerque, New Mexico, 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 local and remote users access to data; 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 System (DPSS)* was developed, later used to support *TerraVision* [78], a terrain visualization application that lets users explore and navigate a tridimensional real landscape.

Another important milestone was set with the Clipper project,[5] 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). Clipper's main focus 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.

8.1.3.2 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 that support data-intensive computing. A data grid provides services that help users discover, transfer, and manipulate large datasets stored in distributed repositories as well as 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

[5] www.nersc.gov/news/annual_reports/annrep98/16clipper.html.

discovery and management mechanisms for easy access to distributed datasets [80]. Because data grids span different administration boundaries, access control and security are important concerns.

Data grids mostly provide storage and dataset management facilities as support for scientific experiments that produce huge volumes of data. The reference scenario might be one depicted in Figure 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 help in determining the locations of the closest datasets of interest for their experiments. Datasets are replicated by the infrastructure to provide better availability. Since processing of this information also requires a large computational power, specific computing sites can be accessed to perform analysis and experiments.

Like 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)*. Besides heterogeneity and security, data grids have their own characteristics and introduce new challenges [79]:

- *Massive datasets.* The size of datasets can easily be on 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.
- *Shared data collections.* Resource sharing includes distributed collections of data. For example, repositories can be used to both store and read data.
- *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 filenames for the purpose of replication and accessibility.
- *Access restrictions.* Even though one of the purposes of data grids is to facilitate sharing of results and data for experiments, some users might want to ensure confidentiality for their data and restrict access to them 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 to data-intensive computing. As a result, several scientific research fields, including high-energy physics, biology, and astronomy, leverage data grids, as briefly discussed here:

- *The LHC Grid.* A project funded by the European Union to develop a worldwide grid computing environment for use by high-energy physics researchers around the world who are 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 U.S. National Institutes of Health (NIH), BIRN provides a data-sharing infrastructure, software tools, and strategies and advisory services (www.birncommunity.org).

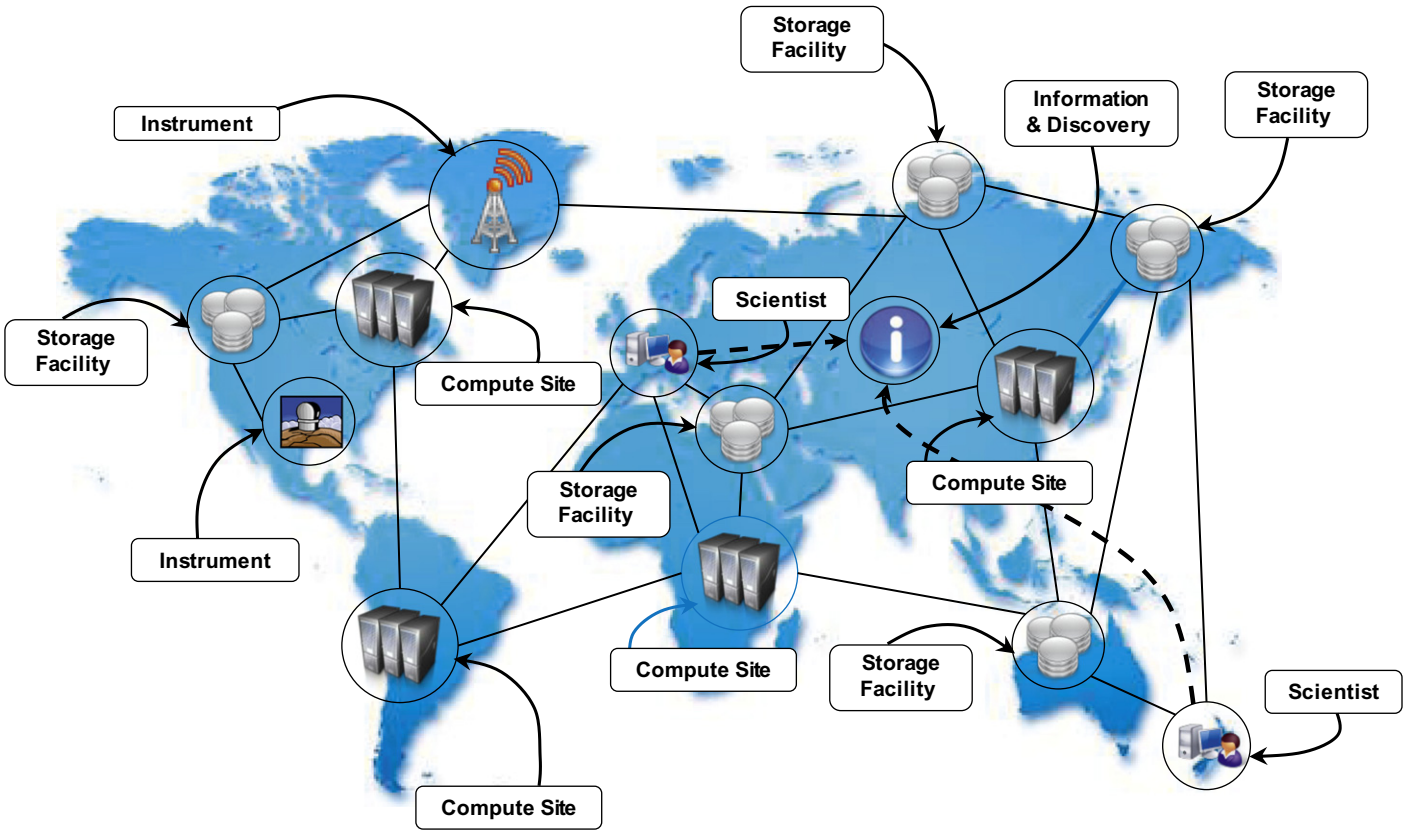

FIGURE 8.2

Data grid reference scenario.

- *International Virtual Observatory Alliance (IVOA)*. IVOA is an organization that aims to provide improved access to the ever-expanding astronomical data resources available online. It does so by promoting standards for *virtual observatories*, which are a collection of interoperating data archives and software tools that use the Internet to form a scientific research environment in which astronomical research programs can be conducted. This allows scientists to discover, access, analyze, and combine lab data from heterogeneous data collections (www.ivoa.net).

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

8.1.3.3 Data clouds and "Big Data"

Large datasets have mostly been the domain of scientific computing. This scenario has recently started to change as massive amounts of data are being produced, mined, and crunched by companies that provide Internet services such as searching, online advertising, and social media. It is critical for such companies to efficiently analyze these huge datasets because they constitute a precious source of information about their customers. *Log analysis* is an example of a 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 using their distributed infrastructure. As a result, they settled upon an analytic infrastructure, which differs from the grid-based infrastructure used by the scientific community.

Together with the diffusion of cloud computing technologies that support data-intensive computations, the term *Big Data* [82] has become popular. This term characterizes the nature of data-intensive computations today 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, instead requiring "massively parallel software running on tens, hundreds, or even thousands of servers" [82].

Big Data problems are found in nonscientific application domains such as weblogs, radio frequency identification (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 ecommerce. Other than the massive size, what characterizes all these examples is that new data are accumulated with time rather than replacing the old data. In general, the term *Big Data* applies to datasets of which the size is beyond the ability of commonly used software tools to capture, manage, and process 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, which 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 amounts of data. These APIs are mostly coupled with a specific storage infrastructure 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 for 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 on commodity hardware. The architecture also integrates MapReduce-like functionality into its platform. A similar architecture has been deployed by *Aster*, which uses an MPP-based data-warehousing appliance that supports MapReduce and targets 1 PB of data.

8.1.3.4 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 participating 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 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 on 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 no longer a rare case but instead 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 data management.

- *Growing importance of data analytics in the business chain.* The management of data is no longer considered a cost but a key element of 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 to support 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 are 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 no longer rare to build software applications and services that are dynamically deployed on hundreds or thousands of nodes, which might belong to the system for a few hours or days. Classical database infrastructures are not designed to provide support to such a volatile environment.

All these factors identify the need for new data management technologies. This not only implies a new research agenda in database technologies and a more holistic approach to the management of information but also leaves room for 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 toward support for data-intensive computing.

8.2.1.1 High-performance distributed file systems and storage clouds

Distributed file systems constitute the primary support for data management. They provide an interface whereby 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.

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 the most powerful supercomputer in the June 2012 list.[6] Lustre is designed to provide access to petabytes (PBs) of storage to serve thousands of clients with an I/O throughput of hundreds of gigabytes per second (GB/s). The system is composed of a metadata server that contains the metadata 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

[6]Top 500 supercomputers list: www.top500.org (accessed in June 2012).

kernel or through a library. The file system implements a robust failover strategy and recovery mechanism, making server failures and recoveries transparent to clients.

IBM General Parallel File System (GPFS). GPFS [88] is the high-performance distributed file system developed by IBM that provides support for the RS/6000 supercomputer and Linux computing clusters. GPFS is a multiplatform distributed file system built over several years of academic research and provides advanced recovery mechanisms. GPFS is built on the concept of shared disks, in which a collection of disks is attached to the file system nodes by means of some switching fabric. The file system makes this infrastructure transparent to users and stripes large files over the disk array by replicating portions of the file 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 the metadata of the entire file system and provides transparent access to it, thus eliminating a single point of failure.

Google File System (GFS). GFS [54] is the storage infrastructure that supports the execution of distributed applications in Google's computing cloud. The system has been designed to be a fault-tolerant, highly 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 Google's needs 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.

The architecture of the file system is organized into a single master, which contains 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 of a collection of software daemons, which implement either the master server or the chunk server. A file is a collection of chunks for which the 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 a 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, specific attention has been given to 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's recovery from failures.

Sector. Sector [84] is the storage cloud that supports the execution of data-intensive applications defined according to the Sphere framework. 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 users to customize the replication strategy for better performance. The system's architecture is composed of 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, whereas master servers coordinate and serve the I/O requests of clients, which ultimately interact with slave nodes to access files. The protocol used to exchange data with slave nodes is UDT [89], which is a lightweight connection-oriented protocol optimized for wide-area networks.

Amazon Simple Storage Service (S3). Amazon S3 is the online 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 identified by a unique key. Objects are identified by unique URLs and exposed through HTTP, thus allowing very simple *get-put* semantics. Because of the use of HTTP, there is no need for any specific library for accessing the storage system, the objects of which can also be retrieved through the Bit Torrent protocol.[7] 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 accessibility of objects are linked to AWS accounts, and the owner of a bucket can decide to make it visible to other accounts or 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 that have architecture that 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 to provide direct access to the storage space. The architecture is completed by client libraries, which provide a simple interface for accessing the file system, which 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. The most important aspect common to all these different implementations is the ability to provide fault-tolerant and highly available storage systems.

8.2.1.2 NoSQL systems

The term *Not Only SQL (NoSQL)* was originally coined in 1998 to identify a relational database that 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 information according to the relational model, but rather is a collection of scripts that

[7] 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.

allow users to manage most of the simplest and more common database tasks by using text files as information stores. Later, in 2009, the term *NoSQL* was reintroduced with the intent of labeling 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. Their 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 avoid joins to increase the performance and scale horizontally.

Two main factors 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 grown considerably in the last decade. These two factors made software engineers look to alternatives that were more suitable to specific application domains they were working on. As a result, several different initiatives explored the use of nonrelational storage systems, which considerably differ from each other. A broad classification is reported by Wikipedia,[8] which distinguishes NoSQL implementations into:

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

Let us now examine some prominent implementations that support data-intensive applications.

Apache CouchDB and MongoDB. Apache CouchDB [91] and MongoDB [90] are two examples of document stores. Both provide a schema-less store whereby 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 represent data in JSON format. Both 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.

Amazon Dynamo. Dynamo [92] is the distributed key-value store that supports the management of information of several of the business services offered by Amazon Inc. The main goal of 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 requests per day. Dynamo provides a simplified

[8] http://en.wikipedia.com/wiki/NoSQL.

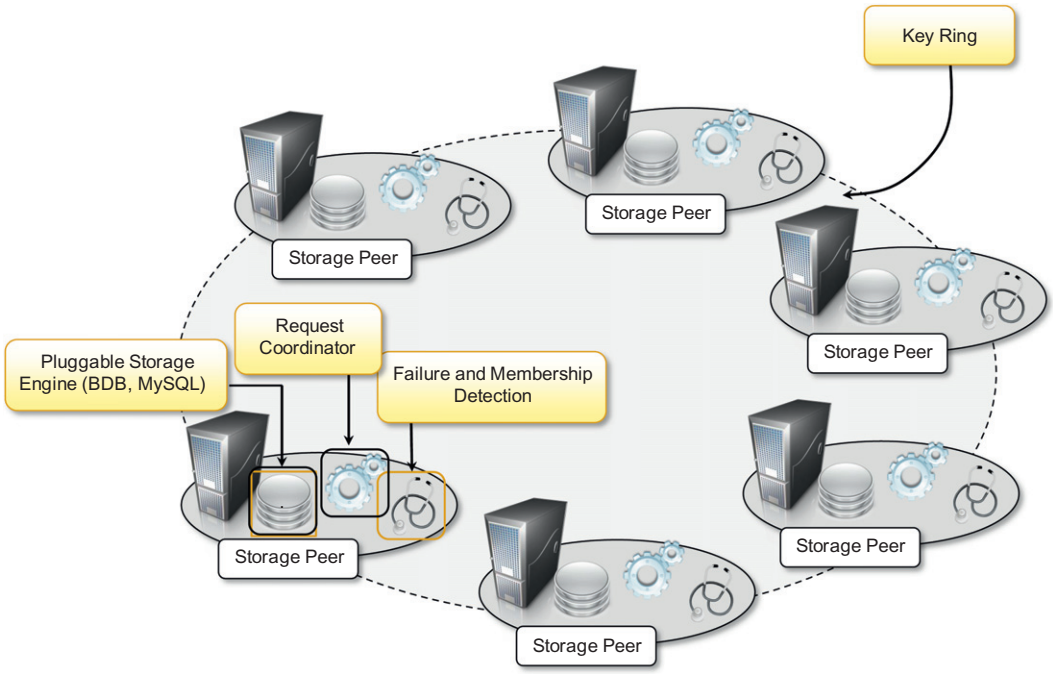

FIGURE 8.3

Amazon Dynamo architecture.

interface based on *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 of these systems. For example, ACID properties on data have been sacrificed in favor of a more reliable and efficient infrastructure. This creates what it is called an *eventually consistent* model, which means that in the long term all the users will see the same data.

The architecture of the Dynamo system, shown in Figure 8.3, is composed of a collection of storage peers organized in a ring that shares 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. Furthermore, each node provides facilities for distributing the updates among the rings 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 the background. The downside of such an approach is the simplicity of the storage model, which requires applications to build their own data models on top of the simple building blocks provided by the store. For example, there are no referential integrity constraints, relationships 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.

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 that expose different types of workload: from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. Bigtable's key design goals are wide applicability, scalability, high performance, and high availability. To achieve these goals, Bigtable organizes the data storage in tables of which the 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 that is represented by a string of arbitrary length. A table is organized into rows and columns; columns can be grouped in column family, which allow for specific optimization for better access control, the storage and the indexing of data. A simple data access model constitutes the interface for client applications that can address data at the granularity level of the single column of a row. 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[9] [95] scripting language or the MapReduce APIs.

Figure 8.4 gives an overview of the infrastructure that enables Bigtable. The service is the result of a collection of processes that coexist 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 (commonly from 10 to 1,000). The master server is responsible for keeping track of the status of the tablet servers and of the allocation of tablets to tablet 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.

Chubby [96]—a distributed, highly available, and persistent lock service—supports the activity of the master and tablet servers. System monitoring and data access are filtered through Chubby, which is also responsible for managing replicas and providing consistency among them. At the very bottom layer, the data are stored in 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 storage of data.

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

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 offer a highly reliable service. Cassandra was initially developed by Facebook; 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

[9] *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.

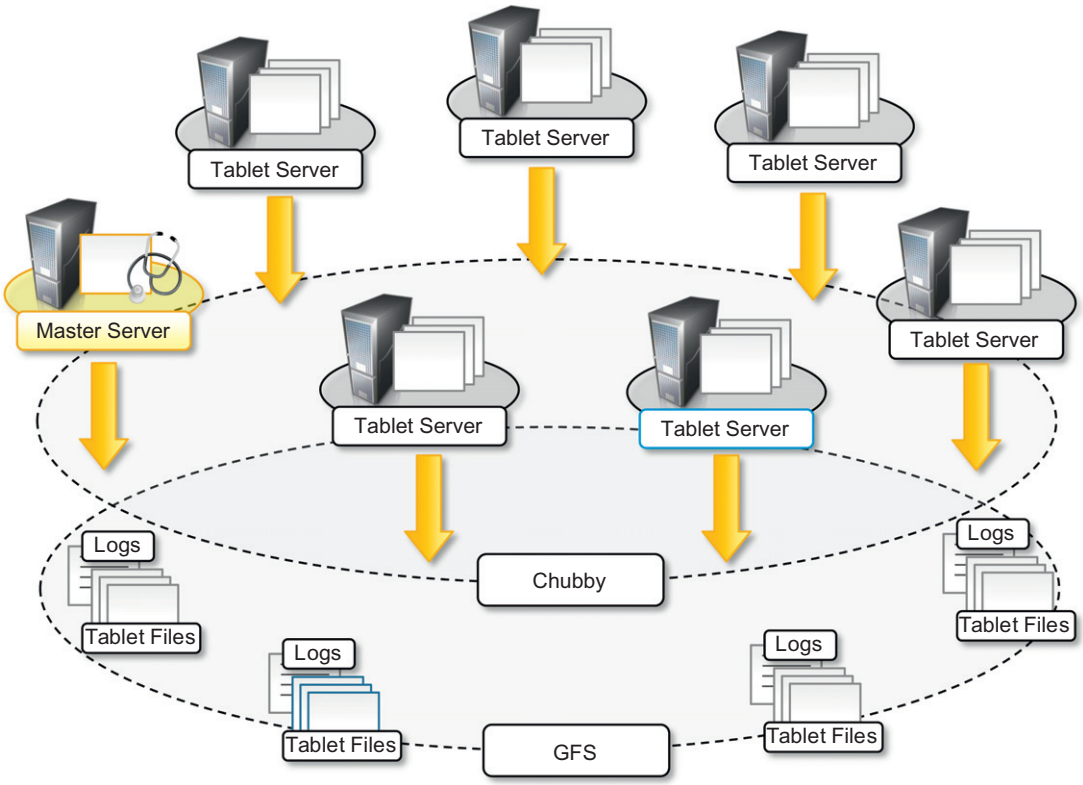

FIGURE 8.4

Bigtable architecture.

second-generation distributed database that builds on the concept of Amazon Dynamo, which follows a fully distributed design, and Google Bigtable, from which it inherits the "column family" concept. The data model exposed by Cassandra is based on the concept of a table that is implemented as a distributed multidimensional 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 the row level; retrieval and deletion can operate at the column level.

In terms 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 replicates its data up to N other nodes. Replication uses different strategies; 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 or datacenter or not to consider the geo-location of nodes. As in Dynamo,

node membership information is based on gossip protocols.[10] Cassandra makes also 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 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. 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 to achieve a completely distributed and highly reliable storage system. The largest Cassandra deployment to our knowledge manages 100 TB of data distributed over a cluster of 150 machines.

Hadoop HBase. HBase is the distributed database that supports the storage needs of the Hadoop distributed programming platform. HBase is designed by taking inspiration from Google Bigtable; 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 services of GFS.

In this section, we discussed the storage solutions that support the management of data-intensive applications, especially those referred as *Big Data*. Traditionally, database systems, most likely based on the relational model, have been the primary solution for handling large quantities of data. As we discussed, when it comes to extremely huge quantities of unstructured data, relational databases become impractical and provide poor performance. Alternative and more effective solutions have significantly reviewed the fundamental concepts at the base of distributed file systems and storage systems. The next level comprises providing programming platforms that, by leveraging the discussed storage systems, can capitalize on developers' efforts to handle massive amounts 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 that help express the computation over a large quantity of information and runtime systems able to efficiently manage huge volumes of data. Traditionally, database management systems based on the relational model have been used to express the structure and connections between the entities of a data model. This approach has proven unsuccessful in the case of Big Data, where information is mostly found unstructured or semistructured and where data are most likely to be organized in files of large size or a huge number of medium-sized files rather than rows in a database. Distributed workflows have often been 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

[10] 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 alternative to distributed and propagate information that is efficient compared to flooding or other kinds of algorithms.

eventually incorporated capabilities to leverage the elastic features offered by cloud computing [70]. These systems are fundamentally based on the abstraction of a *task*, which puts a big 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 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.

8.2.2.1 The MapReduce programming model

MapReduce [55] is a programming platform Google introduced for processing large quantities of data. It expresses the computational logic of an application in two simple functions: *map* and *reduce*. Data transfer and management are 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 no longer have to handle 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 into a workflow of *map* and *reduce* operations that is entirely controlled by the runtime system; developers need only specify how the *map* and *reduce* functions operate on the key-value pairs.

More precisely, the MapReduce 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 of 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 as to 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 constitutes 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 of 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 that supports 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

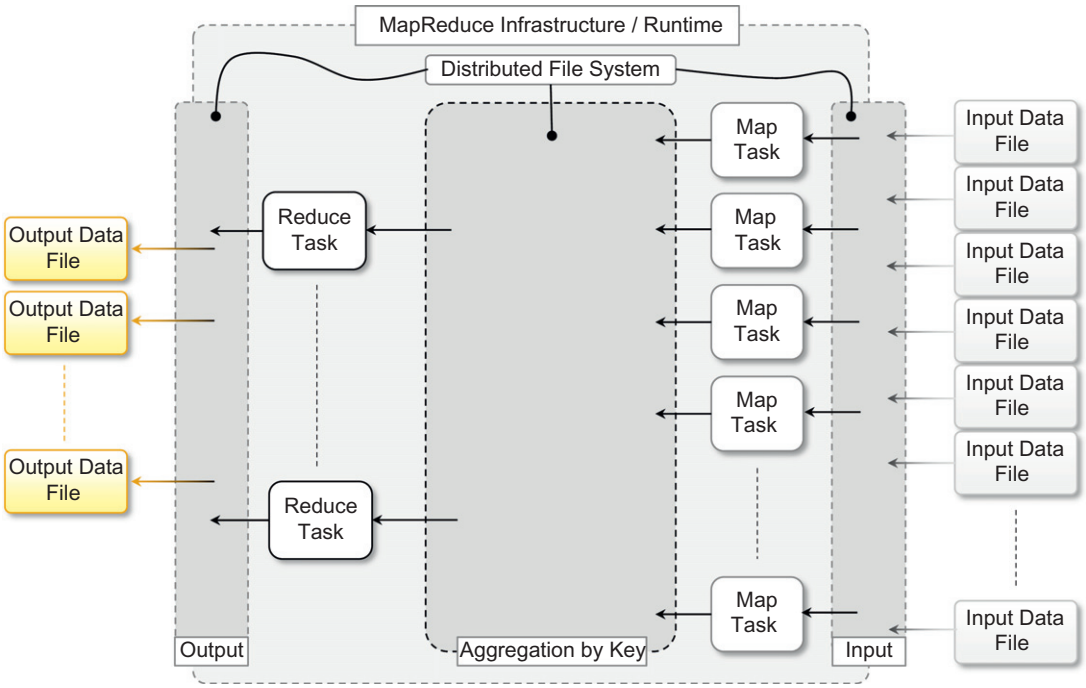

FIGURE 8.5

MapReduce computation workflow.

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 files are partitioned with respect to these tasks, and the number of map tasks connected to a single reduce task are the responsibilities of the MapReduce runtime. In addition, the way files are stored and moved is the responsibility of the distributed file system that supports MapReduce.

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

- *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.
- *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.

- *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 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 one or more files containing these mappings.
- *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.
- *Inverted index.* The inverted index contains information about the presence of words in documents. This information is useful to allow fast full-text searches 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)*$>$.
- *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 reported example 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 a 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.[11] 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:

- *Analysis.* This phase operates directly on 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.

[11]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/.

- *Aggregation.* This phase operates on the intermediate results and is characterized by operations that are aimed at aggregating, summing, and/or elaborating the data obtained at the previous stage to present the data in their final form. This is the task performed by the *reduce* function.

Adaptations to this model are mostly concerned with identifying the appropriate keys, creating reasonable keys when the original problem does not have such a model, and finding ways to partition the 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 it demands a lot of common tasks, which are of concern in the management of a distributed application to the MapReduce runtime, allowing the user to specify only configuration parameters to control the behavior of applications. These tasks are managing data transfer and scheduling map and reduce tasks over a distributed infrastructure. Figure 8.6 gives a more complete overview of a MapReduce infrastructure, according to the implementation proposed by Google [55].

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 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 buffers for map tasks are periodically dumped to disk to create intermediate files. Intermediate files are partitioned using a user-defined function to evenly split the output of map tasks. The locations of these pairs are then notified to the master process, which forwards this information to the reduce tasks, 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 in the global file system. This process is completely automatic; users may control it through configuration parameters that allow specifying (besides the *map* and *reduce* functions) the number of map tasks, the number of partitions into which to separate the final output, and the *partition* function for the intermediate key range.

Besides orchestrating the execution of map and reduce tasks as previously described, the MapReduce runtime ensures a reliable execution of applications by providing a fault-tolerant

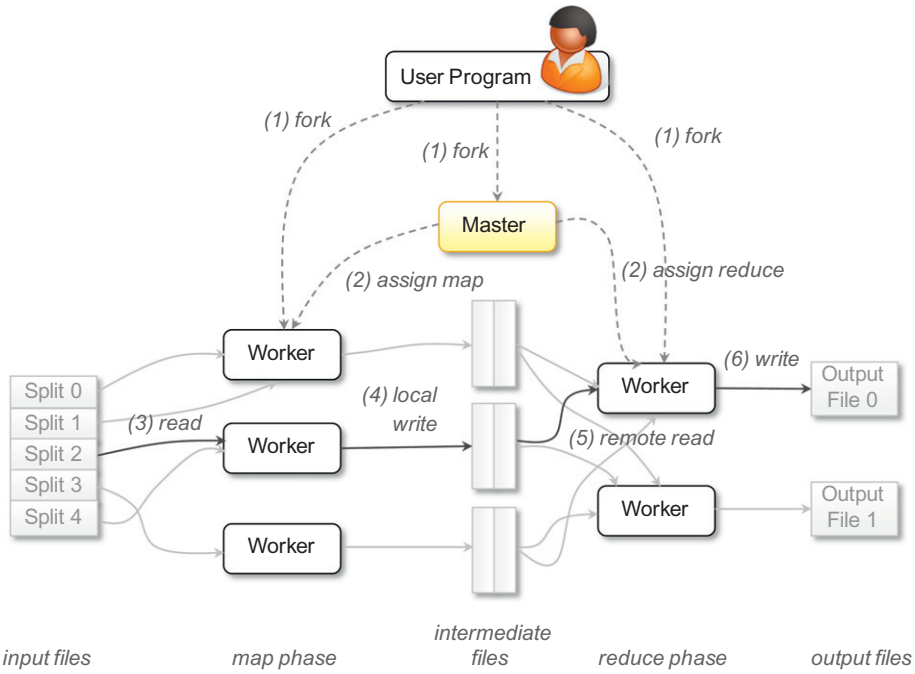

FIGURE 8.6

Google MapReduce infrastructure overview.

infrastructure. Failures of both master and worker processes are handled, as are machine failures that make intermediate outputs inaccessible. 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 using checkpointing, which allows restarting the MapReduce job with a minimum loss of data and computation.

8.2.2.2 Variations and extensions of MapReduce

MapReduce constitutes a simplified model for processing large quantities of data and imposes constraints on the way distributed algorithms should be organized to run over a MapReduce infrastructure. Although the model can be applied to several different problem scenarios, it still exhibits limitations, mostly due to 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 to and variations of the original MapReduce model have been proposed. They aim at extending the MapReduce application space and providing developers with an easier interface for designing distributed algorithms. In this

section, we briefly present a collection of MapReduce-like frameworks and discuss how they differ from the original MapReduce model.

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]; the latter provides the same features and abstractions as Google MapReduce. Initially developed and supported by Yahoo!, Hadoop now constitutes the most mature and large data cloud application and has a very robust community of developers and users supporting it. Yahoo! now runs the world's largest Hadoop cluster, composed of 40,000 machines and more than 300,000 cores, made available to academic institutions all over the world. Besides the core projects of Hadoop, a collection of other projects related to it provides services for distributed computing.

Pig. Pig[12] is a platform that allows the analysis of large datasets. Developed as an Apache project, Pig consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs. The 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 is characterized by major expressiveness, reduced programming effort, and a familiar interface with respect to MapReduce.

Hive. Hive[13] is another Apache initiative that provides a data warehouse 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 as a classical data warehouse, it does not exhibit the same performance, especially in terms of query latency, and for this reason does not constitute a valid solution for online transaction processing. Hive's major advantages reside in the ability to scale out, since it is based on the Hadoop framework, and in the ability to provide a data warehouse infrastructure in environments where there is already a Hadoop system running.

Map-Reduce-Merge. Map-Reduce-Merge [98] is an extension of the MapReduce model, introducing a third phase to the standard MapReduce pipeline—the Merge phase—that allows efficiently 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.

Twister. Twister [99] is an extension of 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`

[12] http://pig.apache.org/.
[13] http://hive.apache.org/.

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.

8.2.2.3 Alternatives to MapReduce

MapReduce, other abstractions provide support for processing large datasets and execute data-intensive workloads. To different extents, these alternatives exhibit some similarities to the MapReduce approach.

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 developers to express the computation in terms of *user-defined functions (UDFs)*, 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 Sector's API for data access. UDFs 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 that is client controlled; a Sphere client sends a request for processing to the master node, which returns the list of available slaves, and the client will choose the slaves on which to execute Sphere processes and orchestrate the entire distributed execution.

All-Pairs. All-Pairs [100] is an abstraction and a runtime 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:

$$All\text{-}pairs(A{:}set, B{:}set, F{:}function) \rightarrow M{:}matrix$$

Examples of problems that can be represented in this model can be found in the field of biometrics, where similarity matrices are composed as a result of the comparison of several images that contain subject pictures. Another example is 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 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: (1) model the system; (2) distribute the data; (3) dispatch batch jobs; and (4) 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 are opportunistically distributed in order to create the optimal number of tasks to assign to each node and optimize the utilization of the infrastructure.

DryadLINQ. Dryad [101] is a Microsoft Research project that investigates programming models for writing parallel and distributed programs to scale from a small cluster to a large datacenter. Dryad's aim is to provide an infrastructure for automatically parallelizing the execution of applications 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 by means of channels. More precisely, a Dryad computation can be expressed in terms of a directed acyclic graph in which 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 that is 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 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 terms of *Mapper* and *Reducer* classes that are extended from the Aneka MapReduce APIs. The runtime support is composed of 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

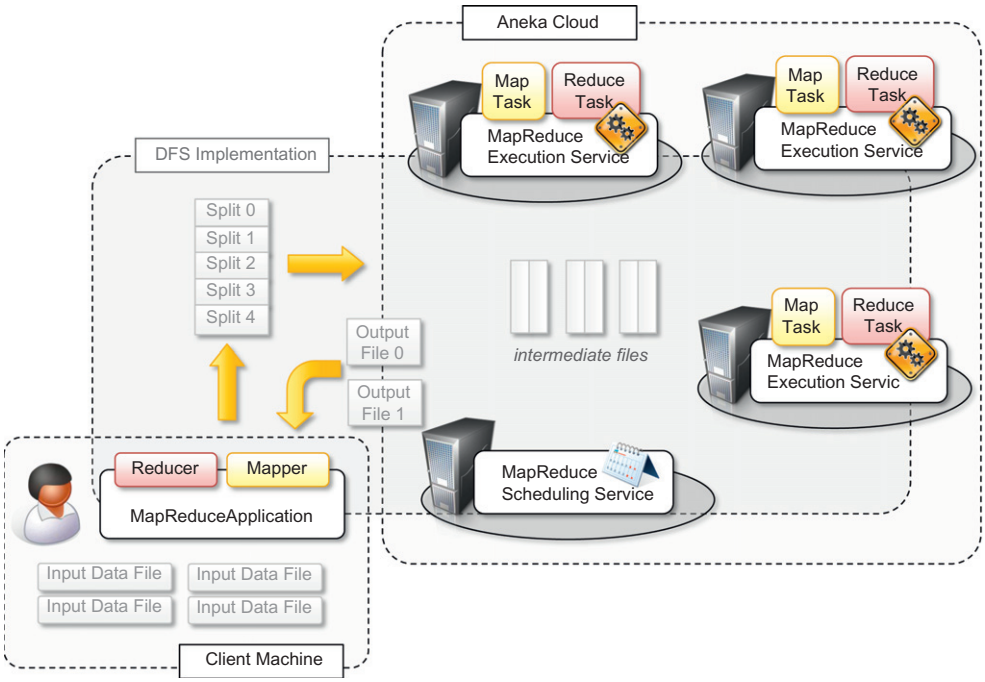

FIGURE 8.7

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 sections, we introduce these major components and describe how they collaborate to execute MapReduce jobs.

8.3.1.1 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 a MapReduce job—used in Google MapReduce and Hadoop—and the Aneka application concept. Unlike 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.

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 amounts 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 the 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 8.1 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. 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 8.2 shows the implementation of the *Mapper<K,V>* component for the Word Counter sample. This sample counts the frequency of words in a set of large text files. The text files are divided into lines, each of which will become the value component of a key-value pair, whereas 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 the key type and a *string* for the value. To count the frequency of words, the *map* function will emit a new key-value pair for each word contained in the line by using the word as the key and the number 1 as the 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 appropriately sum all these occurrences.

Listing 8.3 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 collections. Listing 8.4 shows how to implement the reducer function for the word-counter example.

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 file.

It is important to note 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 provide facilities for defining the computation performed by a MapReduce job. To submit, execute, and monitor its progress, Aneka provides

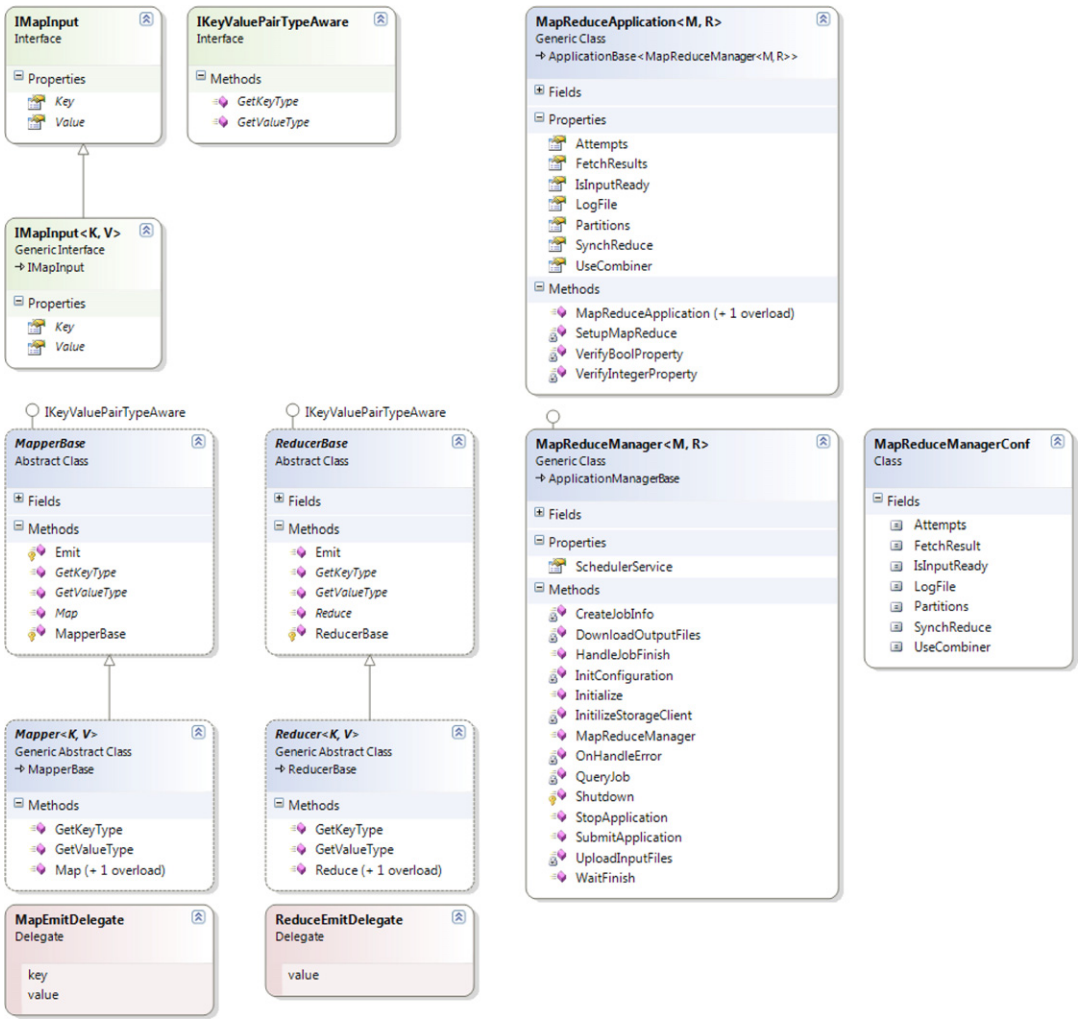

FIGURE 8.8

MapReduce Abstractions Object Model.

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 8.5 shows the interface of *MapReduceApplication* $< M,R >$.

The interface of the class exhibits only the MapReduce-specific settings, whereas the control logic is encapsulated in the *ApplicationBase* $< M >$ class. From this class it is possible set the

```
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); }
```

LISTING 8.1

Map Function APIs.

```
        /// <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 8.1

(Continued)

behavior of MapReduce for the current execution. The parameters that can be controlled are the following:

- *Partitions.* This property stores an integer number containing the number of partitions into which to divide the final results. This value also determines the number of reducer tasks that will be created by the runtime infrastructure. The default value is 10.
- *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.
- *UseCombiner.* This property stores a Boolean value that indicates 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*.
- *SynchReduce.* This property stores a Boolean value that indicates 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.
- *IsInputReady.* This is a Boolean property that indicates whether the input files are already stored in the distributed file system or must be uploaded by the client manager before the job can be executed. The default value is set to *false*.
- *FetchResults.* This is a Boolean property that indicates 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*.
- *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.

```
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 8.2

Simple *Mapper<K,V>* Implementation.

Developers can control the application by using the methods and the properties exposed by the *ApplicationBase<M>* class. Listing 8.6 displays the collection of methods that are of interest in this class for the execution of MapReduce jobs.

Besides the constructors and the common properties that are of interest for all the applications, the two methods in bold in Listing 8.6 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; 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

```
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
    }
}
```

LISTING 8.3

Reduce Function APIs.

```
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);
        }
    }
}
```

LISTING 8.4

Simple *Reducer<K,V>* Implementation.

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 8.7 shows how to create a MapReduce application for running the word-counter example defined by the previous *WordCounterMapper* and *WordCounterReducer* classes.

The lines of interest are those put in evidence in the *try { . . . } catch { . . . } finally { . . .}* block. As shown, 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.

8.3.1.2 Runtime support

The runtime support for the execution of MapReduce jobs comprises the collection of services that deal with scheduling and executing MapReduce tasks. These are the *MapReduce Scheduling*

```
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 { … } }
```

LISTING 8.5

MapReduceApplication < M,R >.

```
        /// <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…
    }
}
```

LISTING 8.5

(Continued)

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.

Job and Task Scheduling. The scheduling of jobs and tasks is the responsibility of the *MapReduce Scheduling Service*, which covers the same role as 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 Aneka requires to expose a software

```
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)
```

LISTING 8.6

ApplicationBase < M >.

```
        { … }
        /// <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.

    }

}
```

LISTING 8.6

(Continued)

```
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();
        }
```

LISTING 8.7

WordCounter Job.

```
        }
        /// <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() { … }
    }
}
```

LISTING 8.7

(Continued)

component as a service; 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 the two components is depicted in Figure 8.9.

The core functionalities for job and task 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; resubmitting failed tasks; and reporting execution statistics. All these operations are performed asynchronously and triggered by events happening in the Aneka middleware.

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 performs other operations, such as sorting and merging intermediate files. The service is internally organized, as described in Figure 8.10.

There are three major components that coordinate together for executing tasks: *MapReduceSchedulerService*, *ExecutorManager*, and *MapReduceExecutor*. The *MapReduceSchedulerService* interfaces the *ExecutorManager* with the Aneka middleware; 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* instance, and this is helpful in the case of multicore nodes, where more than one task can be executed at the same time.

8.3.1.3 Distributed file system support

Unlike the other programming models Aneka supports, 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

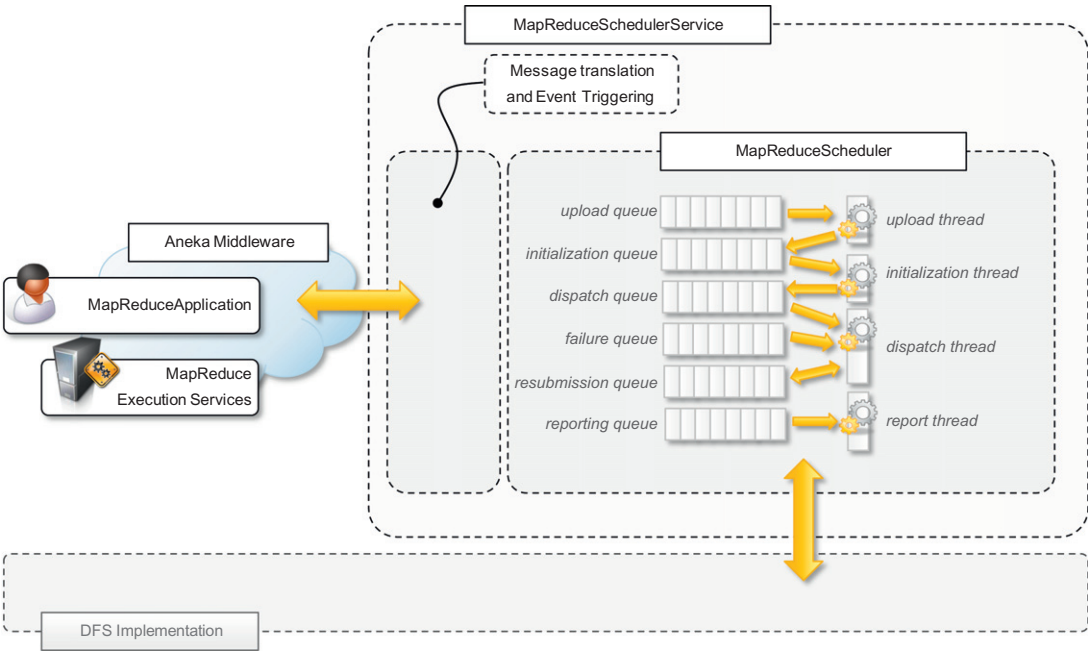

FIGURE 8.9

MapReduce Scheduling Service architecture.

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 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 Chapter 5 (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; the second operation is required for the usual I/O operations to and from data files. In 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.

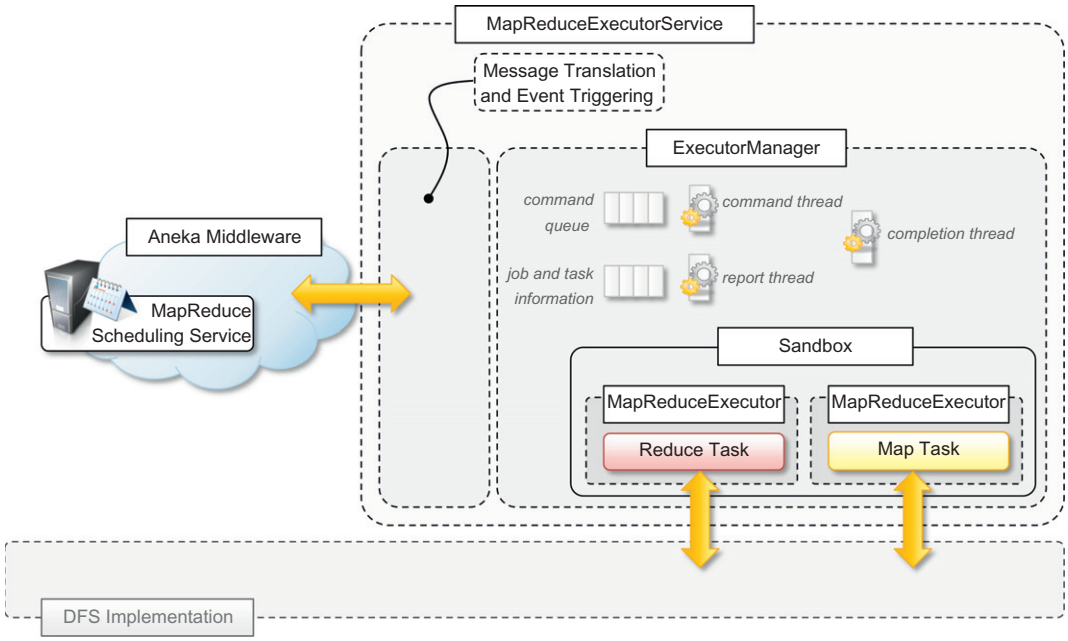

FIGURE 8.10

MapReduce Execution Service architecture.

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 Figure 8.11.

An Aneka MapReduce file is composed of a header, used to identify the file, and a sequence of record blocks, each storing a key-value pair. The header is composed of 4 bytes: the first 3 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, whereas the key is automatically generated and assigned to the position in bytes where the line starts in the file. In the write operation, the writing of the key is skipped and the values are saved as single lines.

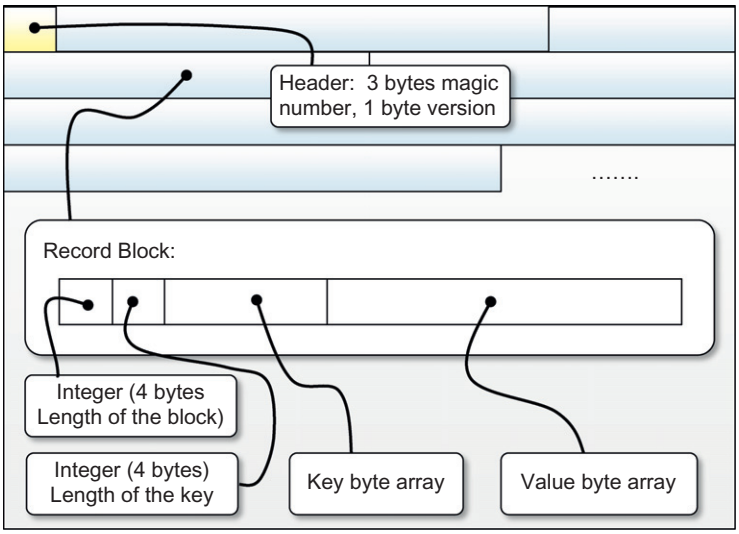

FIGURE 8.11

Aneka MapReduce data file format.

Listing 8.8 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()*. *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 8.9 shows a practical use of the *SeqReader* class by implementing the callback used in the word-counter example. 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.

The *OnDone* callback checks to see whether the application has terminated successfully. If 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 semistructured form, such as logs or Web pages. 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

```
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() { … }
```

LISTING 8.8

SeqReader and *SeqWriter* Classes.

```
        /// <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) { … }
```

LISTING 8.8

(Continued)

```
        /// <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 8.8

(Continued)

detail the problem to be addressed and design the *Mapper* and *Reducer* classes that are used to execute log-parsing and data extraction operations.

8.3.2.1 Parsing Aneka logs

Aneka components (daemons, container instances, and services) produce a lot of information that is stored in the form of log files. The most relevant information is stored in the container instances logs, which store the information about the applications that are executed on the Cloud. In this example, we parse these logs 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 Aneka containers, the system is configured to produce a log file that is partitioned into chunks every time the container instance restarts. Moreover, the information contained in the log file can be customized in its appearance. Currently the default layout is the following:

DD MMM YY hh:mm:ss level – message

```
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();
            }
        }
```

LISTING 8.9

WordCounter Job.

```
/// <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");
        }
    }
}
```

LISTING 8.9

(Continued)

```
        /// <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() { … }
    }
}
```

LISTING 8.9

(Continued)

Some examples of formatted log messages are:

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 parts 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 we discussed, we can perform both analyses at the same time instead of developing two different MapReduce jobs. Note that the operation performed by the *Reducer* class is the same in both cases, whereas 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, we can differentiate 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.

8.3.2.2 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 pairs. This is because some of the log lines do not record the name of the component that entered the line; for these lines, only the key-value pair corresponding to the log level will be emitted.

Listing 8.10 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, to differentiate the log-level labels from component names, an underscore is prefixed to the name of the key in this second case.

8.3.2.3 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.

As we see in Listing 8.11, the operation to perform is very simple and actually is 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 types of information assembled in the output files.

8.3.2.4 Driver program

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.

```
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 <c olon>
            // 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 8.10

Log-Parsing Mapper 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);
        }
    }
}
```

LISTING 8.11

Aneka Log-Parsing Reducer Implementation.

Listing 8.12 shows the implementation of the driver program. With respect to the previous examples, there are three things to be noted:

- The configuration 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 using a specific section of the configuration file named *MapReduce*. Within this section, two subsections 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)* 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 with 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; their meaning was 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 Listing 8.13.

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 simply entering 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 8.12 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 with an exception or information about the types of the exception thrown. In our example, the reporting features, which are triggered in case of exceptions, are implemented in the *ReportError* method. This method utilizes the facilities offered by the *IOUtil* class 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*.

8.3.2.5 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 continuing to run an Aneka Cloud for a few days, it is quite easy to collect enough data to mine for our sample application. Moreover, this

```
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, LogParsingReducer>("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();
            }
```

LISTING 8.12

Driver Program Implementation.

```
        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.
```

LISTING 8.12

(Continued)

```
        /// </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(componentStats))
                        {
                            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);
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
```

LISTING 8.12

(Continued)

```
                    // 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();j
            Environment.ExitCode = -1;
        }
    }
}
```

LISTING 8.12

(Continued)

scenario also constitutes a real case study for MapReduce, since one of its most common practical applications is extracting semistructured information from logs and traces of execution.

In the execution of the test, we used a distributed infrastructure consisting of seven 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 that are stored in the *loglevels.txt* and *components.txt* files and represented graphically in Figures 8.12 and 8.13, respectively.

```
<?xml version="1.0" encoding="utf-8" ?>
<Aneka>
  <UseFileTransfer value="false" />
  <Workspace value="Workspace" />
  <SingleSubmission value="AUTO" />
  <PollingTime value="1000" />
  <LogMessages value="false" />
  <SchedulerUri value="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>
```

LISTING 8.13

Driver Program Configuration File (conf.xml).

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 use 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. We can then infer 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.

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.

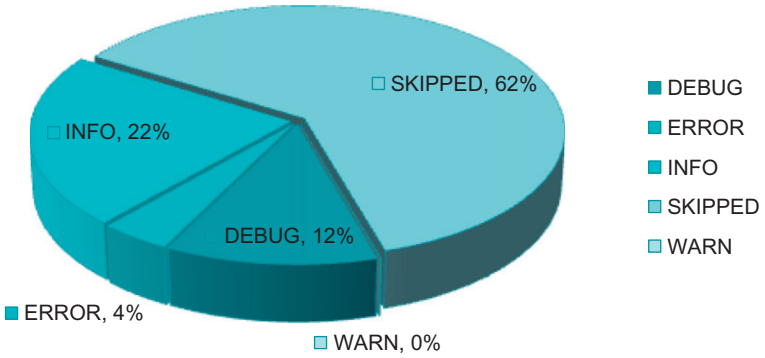

FIGURE 8.12

Log-level entries distribution.

SUMMARY

This chapter 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 amounts of data that have triggered the definition of data-intensive computation has changed over time, together with the technologies and the programming and storage models used for data-intensive computing. Data-intensive computing is a field that was originally prominent in high-speed WAN applications. It is now the domain of storage clouds, where the dimensions of data reach the size of terabytes, if not petabytes, and are referred to 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, online advertising, social media, and social networking.

One of the interesting characteristics of our Big Data world is that the data are represented in a semistructured 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 toward 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 was proposed by Google and provides a simple approach to 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 in key-value pairs, which are eventually aggregated together in the *reduce* phase. This model, even though constraining, proves 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

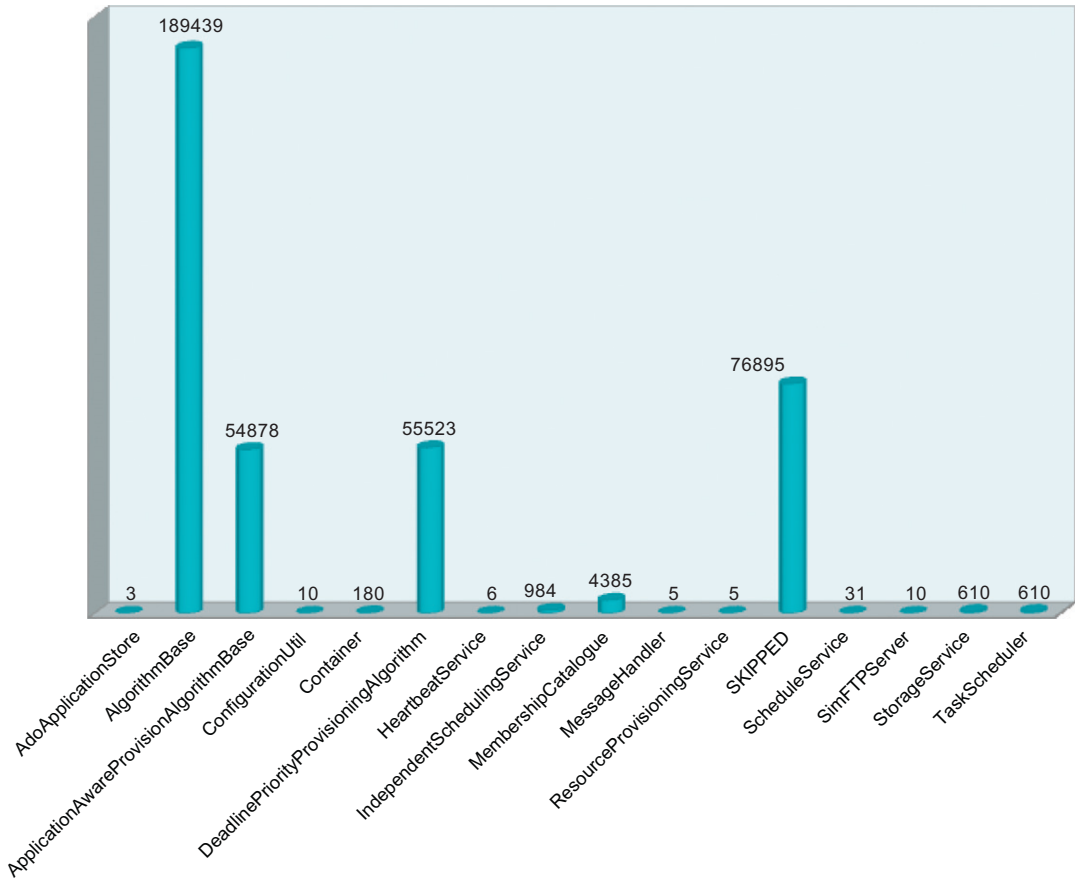

FIGURE 8.13

Component entries distribution.

and task programming models in Aneka, we discussed the programming abstractions supporting the design and implementation of MapReduce applications. We presented the structure and organization of Aneka's runtime services for the execution of MapReduce jobs. Finally, we included step-by-step examples of how to design and implement applications using Aneka MapReduce APIs.

Review questions

1. What is a data-intensive computing? Describe the characteristics that define this term.
2. Provide an historical perspective on the most important technologies that support data-intensive computing.

3. What are the characterizing features of so-called Big Data?
4. List some of the important storage technologies that support 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 that supports data-intensive computations?
10. What is MapReduce?
11. Describe the kinds of problems MapReduce can solve and give some real examples.
12. List some of the variations on 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.

This page intentionally left blank

PART

Industrial Platforms and New Developments 3

This page intentionally left blank

CHAPTER

Cloud Platforms in Industry

9

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 types of service they offer are shown in Table 9.1. A cloud computing system can be developed 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 the market. It provides some insights into and practical issues surrounding the architecture of the major cloud computing technologies and their service offerings.

9.1 Amazon web services

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

Figure 9.1 shows all the services available in the AWS ecosystem. At the base of the solution stack 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 more elastic computing systems. 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 semistructured 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*

Table 9.1 Some Example Cloud Computing Offerings

Vendor/Product	Service Type	Description
Amazon Web Services	IaaS, PaaS, SaaS	Amazon Web Services (AWS) is a collection of Web services that provides developers with compute, storage, and more advanced services. AWS is mostly popular for IaaS services and primarily for its elastic compute service EC2.
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 that provides 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 that allows prototyping of CRM applications. It leverages the Force.com platform, which is made available for developing new components and capabilities for CRM applications.
Heroku	PaaS	Heroku is a scalable runtime environment for building applications based on Ruby.
RightScale	IaaS	Rightscale is a cloud management platform with a single dashboard to manage public and hybrid clouds.

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 examining the solutions proposed by AWS regarding compute, storage, communication, and complementary services.

9.1.1 Compute services

Compute services constitute the fundamental element of cloud computing systems. The fundamental service in this space is Amazon EC2, which delivers an IaaS solution that has served as 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 a preinstalled operating system and a software stack, and instances can be configured for memory, number of processors, and storage. Users are provided with credentials to remotely access the instance and further configure or install software if needed.

9.1.1.1 Amazon machine images

Amazon Machine Images (AMIs) are templates from which it is possible to create a virtual machine. They are stored in Amazon S3 and identified by a unique identifier in the form of *ami-xxxxx* and

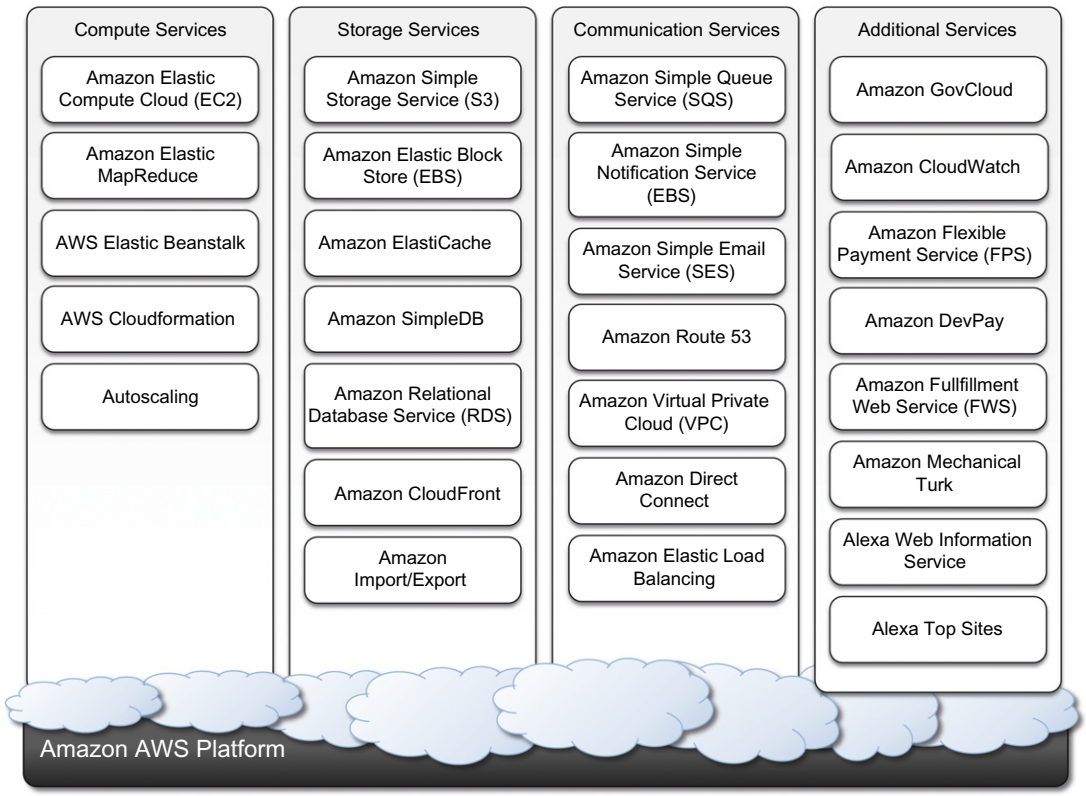

FIGURE 9.1

Amazon Web Services ecosystem.

a manifest XML file. An AMI contains a physical file system layout with a predefined operating system installed. These 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. A common practice is to prepare new AMIs to create an instance from a preexisting AMI, log into it once it is booted and running, and install all the software needed. Using the tools provided by Amazon, we can convert the instance into a new image. Once an AMI is created, it is stored in 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.

9.1.1.2 EC2 instances

EC2 instances represent virtual machines. They are created 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; it is used to express a predictable quantity of real CPU power that 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 standards set by the times. Over time, the hardware supporting the underlying infrastructure will be replaced by more powerful hardware, and the use of ECUs helps 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. One ECU is defined as giving the same performance as a 1.0−1.2 GHz 2007 Opteron or 2007 Xeon processor.[1]

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

- *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.
- *Micro instances*. This class is suitable for those applications that consume a limited amount of computing power and memory and occasionally need bursts in CPU cycles to process surges in the workload. Micro instances can be used for small Web applications with limited traffic.
- *High-memory instances*. This class targets applications that need to process huge workloads and require large amounts of memory. Three-tier Web applications characterized by high traffic are the target profile. Three categories of increasing memory and CPU are available, with memory proportionally larger than computing power.
- *High-CPU instances.* This class targets compute-intensive applications. Two configurations are available where computing power proportionally increases more than memory.
- *Cluster Compute instances*. This class is used to provide virtual cluster services. Instances in this category are characterized by high CPU compute power and large memory and an extremely high I/O and network performance, which makes it suitable for HPC applications.
- *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 this class 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 the cost of the entire hour. The hourly expense charged for one instance is constant. Instance owners are responsible for providing 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 for these instances; as long as the current price (the 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

[1] http://aws.amazon.com/ec2/faqs/#What_is_an_EC2_Compute_Unit_and_why_did_you_introduce_it.

Table 9.2 Amazon EC2 (On-Demand) Instances Characteristics

Instance Type	ECU	Platform	Memory	Disk Storage	Price (U.S. East) (USD/hour)
Standard instances					
Small	1(1 × 1)	32 bit	1.7 GB	160 GB	$0.085 Linux $0.12 Windows
Large	4(2 × 2)	64 bit	7.5 GB	850 GB	$0.340 Linux $0.48 Windows
Extra Large	8(4 × 2)	64 bit	15 GB	1,690 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	6.5(2 × 3.25)	64 bit	17.1 GB	420 GB	$0.500 Linux $0.62 Windows
Double Extra Large	13(4 × 3.25)	64 bit	34.2 GB	850 GB	$1.000 Linux $1.24 Windows
Quadruple Extra Large	26(8 × 3.25)	64 bit	68.4 GB	1,690 GB	$2.000 Linux $2.48 Windows
High-CPU instances					
Medium	5(2 × 2.5)	32 bit	1.7 GB	350 GB	$0.170 Linux $0.29 Windows
Extra Large	20(8 × 2.5)	64 bit	7 GB	1,690 GB	$0.680 Linux $1.16 Windows
Cluster instances					
Quadruple Extra Large	33.5	64 bit	23 GB	1,690 GB	$1.600 Linux $1.98 Windows
Cluster GPU instances					
Quadruple Extra Large	33.5	64 bit	22 GB	1,690 GB	$2.100 Linux $2.60 Windows

them active, there is no such guarantee for spot instances. Therefore, implementing backup and checkpointing strategies is inevitable.

EC2 instances can be run either by using the command-line tools provided by Amazon, which connects the Amazon Web Service that provides remote access to the EC2 infrastructure, or via 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 shut down, the content of the disk will be lost. Alternatively, it is possible to attach an EBS volume to the instance, the content of which will be stored in S3. If the default AKI and ARI are not suitable, EC2 provides capabilities to run EC2 instances by specifying a different AKI and ARI, thus giving flexibility in the creation of instances.

9.1.1.3 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 security in terms of access control and network connectivity.

By default, instances are created with an internal IP address, which makes them capable of communicating within the EC2 network and accessing the Internet as clients. It is possible to associate an *Elastic IP* to each instance, which can then be remapped to a different instance over time. Elastic IPs allow instances running in EC2 to act as servers reachable from the Internet and, since they are not strictly bound to specific instances, 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. Currently, there are five availability zones that are priced differently: two in the United States (Virginia and Northern California), one in Europe (Ireland), and two in Asia Pacific (Singapore and Tokyo).

Instance owners can partially control where to deploy instances. Instead, they have a finer control over 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.

9.1.1.4 Advanced compute services

EC2 instances and AMIs constitute the basic blocks for building an IaaS computing cloud. On top of these, Amazon Web Services provide more sophisticated services that allow the easy packaging and deploying of applications and a computing platform that supports the execution of MapReduce-based applications.

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 that describe 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.

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 provides 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 cloud deployment, Beanstalk simplifies tedious tasks without removing the user's capability of accessing—and taking over 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.

Amazon elastic MapReduce

Amazon Elastic MapReduce provides AWS users with a cloud computing platform for MapReduce applications. It utilizes Hadoop as the MapReduce engine, deployed on a virtual infrastructure composed of EC2 instances, and uses Amazon S3 for storage needs.

Apart from supporting all the application stack connected to Hadoop (Pig, Hive, etc.), Elastic MapReduce introduces elasticity and allows users to dynamically size the Hadoop cluster according to their needs, as well as select 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 applications 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 formats. The core components of S3 are two: *buckets* and *objects*. Buckets represent virtual containers in which to store objects; objects represent the content that is actually stored. Objects can also be enriched with metadata that can be used to tag the stored content with additional information.

9.1.2.1 S3 key concepts

As the name suggests, S3 has been designed to provide a simple storage service that's accessible through a Representational State Transfer (REST) interface, which is quite similar to a distributed file system but which 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 fact, there are few limitations in naming objects, and this allows users to simulate directories and create logical groupings.
- *Stored objects 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 it 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, because it is 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 practice introduces latencies when adding objects to the store—especially large ones—which are not available instantly across the entire globe.
- *Requests will occasionally fail.* Due to the large distributed 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 is generally not identified as a persistent failure.

Access to S3 is provided with RESTful Web services. These express all the operations that can be performed on 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, and *DELETE* requests are used to remove elements or information attached to them.

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. To express a bucket name in this form, the name has to do all of the following:
  - Be between 3 and 63 characters long
  - Contain only letters, numbers, periods, and dashes
  - Start with a letter or a number
  - Contain at least one letter
  - Have no fragments between periods that start with a dash or end with a dash or that are empty strings

  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 of 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 a 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 a part of the resource component of a URI. Since a 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 fact, they can still be used to create logical groupings that look like directories.

Finally, specific information about a given object, such as its access control policy or the server logging settings defined for a bucket, can be referenced using a specific parameter. 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.

Buckets

A *bucket* is a container of objects. It can be thought of as a virtual drive hosted on the S3 distributed storage, which provides users with a flat store to which they can add objects. Buckets are top-level elements of the S3 storage architecture and do not support nesting. That is, it is not possible to create "subbuckets" or other kinds of physical divisions.

A bucket is located in a specific geographic location and eventually replicated for fault tolerance and better content distribution. Users can select the location at which to create buckets, which by default are created in Amazon's U.S. 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 specifying the name of the bucket. Once created, the bucket cannot be renamed or relocated. 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 can be successful if and only if the bucket is empty.

Objects and metadata

Objects constitute the content elements stored in S3. Users either store files or push to the 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 1,024 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 for the lack of a structured file system, since directories can be emulated by properly naming objects.

Users create an object via a *PUT* request that specifies the name of the object together with the bucket name, its contents, 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. It is possible to retrieve an object via a *GET* request; deleting an object 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, whereas the second ones are meaningful to the user, who can store up to 2 KB per metadata property represented by a key-value pair of strings.

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* grants all of the preceding permissions.

Grantees can be either single users or groups. Users can be identified by their canonical IDs or the email addresses they provided when they signed up for S3. For groups, only three options are available: all users, authenticated users, and log delivery users.[2]

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 retrieval of the ACP; a *PUT* method allows uploading of a new ACP to replace the existing one. Alternatively, it is possible to use a predefined set of permissions called *canned policies* to set the ACP at the time a resource is created. These policies represent the most common access patterns for S3 resources.

ACPs provide a set of powerful rules to control S3 users' access to resources, but they do not exhibit fine grain in the case of nonauthenticated users, who cannot be differentiated and are considered as a group. 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.

[2]This group identifies a specific group of accounts that automated processes use to perform bucket access logging.

Advanced features

Besides the management of buckets, objects, and ACPs, S3 offers other additional features that can be helpful. These features are server access logging and integration with the *BitTorrent* file-sharing network.

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; it 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 in which to save the logging files and the file name prefix. A *GET* request to the same URI allows the user to retrieve 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 using the *BitTorrent* protocol. This is done by appending *?torrent* to the URI of the S3 object. To actually download the object, its ACP must grant read permission to everyone.

9.1.2.2 Amazon elastic block store

The Amazon Elastic Block Store (EBS) allows AWS users to provide EC2 instances with persistent storage in the form of volumes that can be mounted at instance startup. 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 life cycle and is persisted into S3. EBS volumes can be cloned, used as boot partitions, and constitute 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 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 the background and according to the request made by the operating system. This reduces the number of I/O requests that go to the network. Volume images cannot be shared among instances, but multiple (separate) active volumes can be created from them. In addition, 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 an operation.

The expense related to a volume comprises the cost generated by the amount of storage occupied in S3 and by the number of I/O requests performed against the volume. Currently, Amazon charges $0.10/GB/month of allocated storage and $0.10 per 1 million requests made to the volume.

9.1.2.3 Amazon ElastiCache

ElastiCache is an implementation of an elastic in-memory cache based on a cluster of EC2 instances. It provides 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. Furthermore, 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 types of instances; Table 9.3 provides an overview of the pricing options.

The prices indicated in Table 9.3 are related to the Amazon offerings during 2011–2012, and the amount of memory specified represents the memory available after taking system software overhead into account.

9.1.2.4 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 back-end 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*.

Preconfigured EC2 AMIs

Preconfigured EC2 AMIs are predefined templates featuring an installation of a given database management system. EC2 instances created from these AMIs 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, who has to configure, maintain, and manage the relational database, but offers the greatest variety of products to choose from.

Amazon RDS

RDS is relational database service that relies on the EC2 infrastructure and is managed by Amazon. Developers do not have to worry about configuring the storage for high availability, designing

Table 9.3 Amazon EC2 (On-Demand) Cache Instances Characteristics, 2011–2012

Instance Type	ECU	Platform	Memory	I/O Capacity	Price (U.S. East) (USD/hour)
Standard instances					
Small	1(1 × 1)	64 bit	1.3 GB	Moderate	$0.095
Large	4(2 × 2)	64 bit	7.1 GB	High	$0.380
Extra Large	8(4 × 2)	64 bit	14.6 GB	High	$0.760
High-Memory instances					
Extra Large	6.5(2 × 3.25)	64 bit	16.7 GB	High	$0.560
Double Extra Large	13(4 × 3.25)	64 bit	33.8 GB	High	$1.120
Quadruple Extra Large	26(8 × 3.25)	64 bit	68 GB	High	$2.240
High-CPU instances					
Extra Large	26(8 × 3.25)	64 bit	6.6 GB	High	$0.760

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 solutions. The high-availability solution is implemented by keeping in standby synchronized copies of the services in different availability zones that are activated if 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 instances for long terms (one to three years) by paying up-front at discounted hourly rates.

With respect to the previous solution, users are not responsible for managing, configuring, and patching the database management software, but these operations are performed by the AWS. In addition, 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.

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 semistructured data, the model for which is based on the concept of *domains*, *items*, and *attributes*. With respect to the relational model, this model provides fewer constraints on the structure of data entries, thus obtaining improved performance in 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 stores.

Table 9.4 Amazon RDS (On-Demand) Instances Characteristics, 2011–2012

Instance Type	ECU	Platform	Memory	I/O Capacity	Price (U.S. East) (USD/hour)
Standard instances					
Small	1(1 × 1)	64 bit	1.7 GB	Moderate	$0.11
Large	4(2 × 2)	64 bit	7.5 GB	High	$0.44
Extra Large	8(4 × 2)	64 bit	15 GB	High	$0.88
High-Memory instances					
Extra Large	6.5(2 × 3.25)	64 bit	17.1 GB	High	$0.65
Double Extra Large	13(4 × 3.25)	64 bit	34 GB	High	$1.30
Quadruple Extra Large	26(8 × 3.25)	64 bit	68 GB	High	$2.60

SimpleDB uses *domains* as top-level elements to organize a data store. These domains are roughly comparable to tables in the relational model. Unlike tables, they allow items not to 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 make snapshots of 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 operators: $=$, $!=$, $<$, $>$, $<=$, $>=$, *like*, *not like*, *between*, *is null*, *is not null*, and *every()*. Here is a simple example on how to query data:

```
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.

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 is 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 preexisting values of attributes for an item.

Table 9.5 provides an overview of the pricing options for the SimpleDB service for data transfer during 2011–2012. The service charges either for data transfer or stored data. Data transfer within the AWS network is not charged. In addition, SimpleDB also charges users for machine usage. The first 25 SimpleDB instances per month are free; after this threshold there is an hourly charge ($0.140 hour in the U.S. 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 information for clarifying the different nature of SimpleDB with respect to S3: The former has been designed to provide fast access to semistructured collections of small objects and not for being a long-term storage option for large objects.

9.1.2.5 Amazon CloudFront

CloudFront is an implementation of a content delivery network on top of the Amazon distributed storage infrastructure. It leverages a collection of edge servers strategically located around the globe to better serve requests for static and streaming Web content so that the transfer time is reduced as much as possible.

Table 9.5 Amazon SimpleDB Data Transfer Charges, 2011–2012

Instance Type	Price (U.S. 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

AWS provides users with simple Web service APIs to manage CloudFront. 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 the 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 (Real Time Messaging Protocol, or 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 access to the distribution to only one or a few of the available protocols, or they can set up access rules for finer control. It is also possible to invalidate content to remove it from the distribution or force its update before expiration.

Table 9.6 provides a breakdown of the pricing during 2011–2012. Note that CloudFront is cheaper than S3. This reflects its different purpose: CloudFront is designed to optimize the distribution of very popular content that is frequently downloaded, potentially from the entire globe and not only the Amazon network.

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*.

9.1.3.1 Virtual networking

Virtual networking comprises a collection of services that allow AWS users to control the connectivity to and between compute and storage services. *Amazon Virtual Private Cloud (VPC)* and

Table 9.6 Amazon CloudFront On-Demand Pricing, 2011–2012

Pricing Item	United States	Europe	Hong Kong and Singapore	Japan	South America
Requests					
Per 10,000 HTTP requests	$0.0075	$0.0090	$0.0090	$0.0095	$0.0160
Per 10,000 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

Amazon Direct Connect provide connectivity solutions in terms of infrastructure; *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 providers prepare either 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 network address translation (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. There are only two available ports located in the United States, but users can leverage external providers that offer guaranteed high bandwidth to these ports. Two different bandwidths can be chosen: 1 Gbps, priced at $0.30 per hour, and 10 Gbps, priced at $2.25 per hour. Inbound traffic is free; outbound traffic is priced at $0.02 per GB.

Amazon Route 53 implements dynamic DNS services that allow 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.[3] EC2 instances are likely to be more dynamic than the physical machines, and S3 buckets might also exist for a limited time. To cope with such a volatile nature, the service provides AWS users with the capability of dynamically mapping names to resources as instances are launched on EC2 or as new buckets are created in S3. By interacting with the Route 53 Web service, users can manage a set of *hosted zones*, which represent the user domains controlled by the service, and edit the resources made available through it. Currently, a single user can have up to 100 zones. The costing model includes a fixed amount ($1 per zone per month) and a dynamic component that depends on the number of queries resolved by the service for the hosted zones ($0.50 per million queries for the first billion of queries a month, $0.25 per million queries over 1 billion of queries a month).

9.1.3.2 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 disconnected model for exchanging messages between applications by means of message queues, hosted within the AWS infrastructure. 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 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, and they can be accessed 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 a publish-subscribe method for connecting heterogeneous applications. With respect to Amazon SQS, where it is necessary to continuously poll a given queue for a 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 whereby 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 can be automatically notified. The service provides subscribers with different notification models (HTTP/HTTPS, email/email JSON, and SQS).

Amazon SES provides AWS users with a scalable email service that leverages the AWS infrastructure. Once users are signed up for the service, they have to provide an email that SES will use to send emails on their behalf. To activate the service, SES will send an email to verify the given address and provide the users with the necessary information for the activation. Upon verification, the user is given an SES sandbox to test the service, and he can request access to the production version. Using SES, it is possible to send either SMTP-compliant emails or raw emails by specifying email headers and Multipurpose Internet Mail Extension (MIME) types. Emails are queued for

[3] 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 a DNS server is therefore authoritative for these names because it can directly resolve the names. For resolving the other names, the nearest authoritative DNS is contacted.

delivery, and the users are notified of any failed delivery. SES also provides a wide range of statistics that help users to improve their email campaigns for effective communication with customers.

With regard to the costing, all three services do not require a minimum commitment but are based on a pay-as-you go model. Currently, users are not charged until they reach a minimum threshold. In addition, data transfer-in is not charged, but data transfer-out is charged by ranges.

9.1.4 Additional services

Besides compute, storage, and communication services, AWS provides a collection of services that allow users to utilize services in aggregation. The two relevant services are *Amazon CloudWatch* and *Amazon Flexible Payment Service (FPS)*.

Amazon CloudWatch is a service that provides a comprehensive set of statistics that help developers understand and optimize the behavior of their application hosted on AWS. CloudWatch collects information from several other AWS services: EC2, S3, SimpleDB, CloudFront, and others. Using CloudWatch, developers can see a detailed breakdown of their usage of the service they are renting on AWS and can devise more efficient and cost-saving applications. Earlier services of CloudWatch were offered only through subscription, but now it is made available for free to all the AWS users.

Amazon FPS infrastructure allows AWS users to leverage Amazon's billing infrastructure to sell goods and services to other AWS users. Using Amazon FPS, developers do not have to set up alternative payment methods, and they can charge users via a billing service. The payment models available through FPS include one-time payments and 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 control and configure the computing infrastructure hosted in the cloud. They can leverage other 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, but other options, such as Amazon SQS, SNS, and SES, provide solutions for dynamically connecting applications from both inside and outside the AWS Cloud. Network connectivity to AWS applications is addressed by Amazon VPC and Amazon Direct Connect.

9.2 Google AppEngine

Google AppEngine is a PaaS implementation that provides 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 number of requests by allocating more computing resources to them and balancing the load among them. The runtime is completed by a collection of services that allow 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 usage of Google resources and services is metered by AppEngine, which bills users when their applications finish their free quotas.

9.2.1 Architecture and core concepts

AppEngine is a platform for developing scalable applications accessible through the Web (see Figure 9.2). The platform is 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.

9.2.1.1 Infrastructure

AppEngine hosts Web applications, and its primary function is to serve users requests efficiently. To do so, AppEngine's infrastructure takes advantage of many servers available within Google datacenters. For each HTTP request, AppEngine locates the servers hosting the application that processes the request, evaluates their load, and, if necessary, allocates additional resources (i.e., servers) or 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, which can redirect each of the requests to any of the servers hosting the target application or even allocate a new one.

The infrastructure is also responsible for monitoring application performance and collecting statistics on which the billing is calculated.

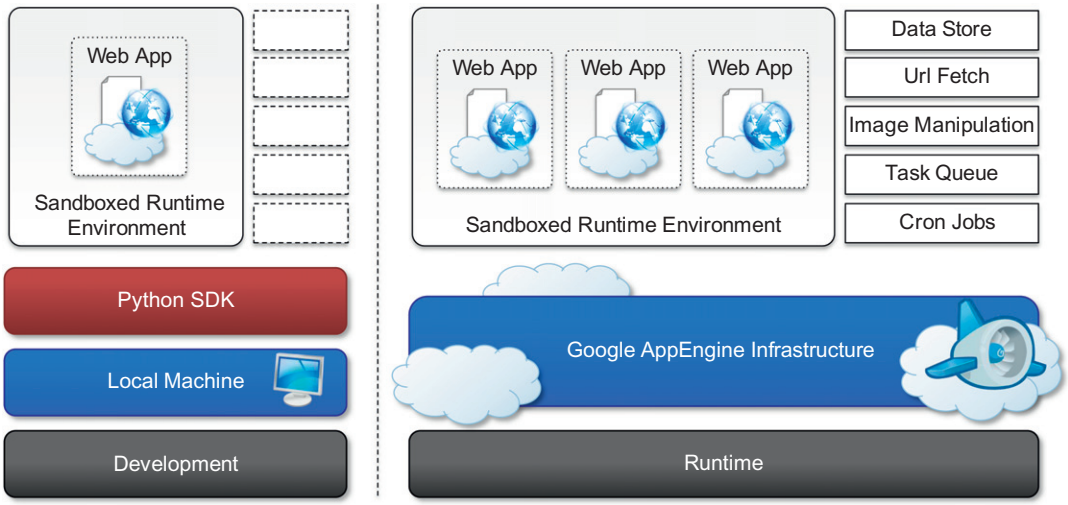

FIGURE 9.2

Google AppEngine platform architecture.

9.2.1.2 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 executing and terminates once the handler has completed.

Sandboxing

One of the major responsibilities of the runtime environment is to provide the application environment with an isolated and protected context in which it can execute without causing 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 for 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 that is considered potentially harmful, an exception is thrown and the execution is interrupted. Some of the operations that are not allowed in the sandbox include 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.

Supported runtimes

Currently, it is possible to develop AppEngine applications 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 application development in Java, such as the *Java Server Pages (JSP)*, and the applications interact with the environment by using the *Java Servlet* standard. Furthermore, access to AppEngine services is provided by means of Java libraries that expose specific interfaces of provider-specific implementations of a given abstraction layer. Developers can create applications with the AppEngine Java SDK, which allows developing applications 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 Python 2.5.2 interpreter. As with 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 to call specific methods generate exceptions. To support application development, AppEngine offers a rich set of libraries connecting applications to AppEngine services. In addition, developers can use a specific Python Web application framework, called *webapp*, simplifying the development of Web applications.

The Go runtime environment allows applications developed with the Go programming language to be hosted and executed in AppEngine. Currently the release of Go that is 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 with the Python environment, some of the functionalities have been removed or generate a runtime exception. In addition, developers can include third-party libraries in their applications as long as they are implemented in pure Go.

9.2.1.3 Storage

AppEngine provides various types of storage, which operate differently depending on the volatility of the data. There are three different levels of storage: in memory-cache, storage for semistructured data, and long-term storage for static data. In this section, we describe *DataStore* and the use of static file servers. We cover *MemCache* in the application services section.

Static file servers

Web applications are composed of dynamic and static data. Dynamic data are a result of the logic of the application and the interaction with the user. Static data often are mostly constituted of 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.

DataStore

DataStore is a service that allows developers to store semistructured data. The service is designed to scale and optimized to quickly access data. DataStore can be considered as a large object database in which 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 were taken in order to obtain a more scalable and efficient data store. The underlying infrastructure of *DataStore* is based on *Bigtable* [93], a redundant, distributed, and semistructured data store that organizes data in the form of tables (see Section 8.2.1).

DataStore provides high-level abstractions that simplify 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 that define the data it stores. Properties are defined according to one of the several primitive types supported by the service. Each entity is associated with a key, which is either provided by the user or created automatically by AppEngine. An entity is associated with a *named kind* that AppEngine uses 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 into account. Entities 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 versions of the same values. Finally, keys are immutable elements and, once created, they cannot be changed.

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 by specifying either the key or conditions on the values of the properties. Returned result sets can be sorted 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; 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 queries made against the simulated data store and creates an index for them. The structure of the indexes is saved in a configuration file and can be further changed by the developer before uploading the application. The use of precomputed 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 atomically. Multiple operations on the same entity can be performed within the context of a transaction. It is also possible to update multiple entities atomically. 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 regard to concurrency, AppEngine uses an *optimistic concurrency control*: If one user tries to update an entity that is already being updated, the control returns and the operation fails. Retrieving an entity never incurs into exceptions.

9.2.1.4 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.

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 fragments of HTML generated in different ways. They can be directly obtained from a remote server or rendered from an XML document retrieved from a Web service, or they can be 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. Therefore, it is 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 it does provide 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 deadlines for requests so that they can be completed (or aborted) within a given time. Moreover, the ability to perform such requests asynchronously allows the applications to continue with their logic while the resource is retrieved in the 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.

MemCache

AppEngine provides developers with access to 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 are persisted to disk redundantly in order to ensure reliability and availability of data against failures. This design poses a limit on how much faster the store can be compared to other solutions, especially for objects that are frequently accessed—for example, 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 developers with a volatile store for the objects that are frequently accessed. The caching algorithm implemented by MemCache will automatically remove the objects that are rarely accessed. 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 lookups.

Mail and instant messaging

Communication is another important aspect of Web applications. It is common to use email for following up with users about operations performed by the application. Email can also be used to trigger activities in Web applications. To facilitate the implementation of such tasks, AppEngine provides developers with the ability to send and receive mails through *Mail*. The service allows sending email on behalf of the application to specific user accounts. It is also possible to include several types of attachments and to 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 Extensible Messaging and Presence Protocol (XMPP). Any chat service that supports 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 a communication medium 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.

Account management

Web applications often keep various data that customize their interaction with users. These data normally go under the user profile and are attached to an account. AppEngine simplifies account management by allowing developers to leverage Google account management by means of *Google Accounts*. The integration with the service also allows Web applications to offload the implementation of authentication capabilities to Google's authentication system.

Using Google Accounts, Web applications can conveniently store profile settings in the form of key-value pairs, attach them to a 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 have a Google account, but it does not require any further implementation. The use of Google Accounts is particularly advantageous for 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.

Image manipulation

Web applications render pages with graphics. Often simple operations, such as adding watermarks or applying simple filters, are required. 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 is optimized for speed.

9.2.1.5 Compute services

Web applications are mostly designed to interface applications with users by means of a ubiquitous channel, that is, the Web. Most of the interaction is performed synchronously: Users navigate the Web pages and get instantaneous feedback in response to 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 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 immediate feedback and a notification once the required operation is completed. AppEngine offers additional services such as *Task Queues* and *Cron Jobs* that simplify the execution of computations that are off-bandwidth or those that cannot be performed within the timeframe of the Web request.

Task queues

Task Queues allow applications to submit a task for a later execution. This service is particularly useful for long computations that cannot be completed 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 reexecute the task in case of failure in order to avoid transient failures preventing the task from a successful completion.

Cron jobs

Sometimes the length of computation might not be the primary reason that 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. This service operates similarly to Task Queues but invokes the request handler specified in the task at a given time and does not reexecute the task in case of failure. This behavior can be useful to implement maintenance operations or send periodic notifications.

9.2.2 Application life cycle

AppEngine provides support for almost all the phases characterizing the life cycle of an application: testing 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: Java SDK and Python SDK.

9.2.2.1 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 tune 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 contains a complete set of monitoring features that are helpful to profile the behavior of applications, especially regarding access to the DataStore service and the queries performed against it. This is a particularly important feature that will be of relevance in deploying the application to AppEngine. As discussed earlier, AppEngine builds indexes for each of the queries performed by a given application in order to speed up 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 analyzes application behavior while running and traces all the queries made during testing and development, thus providing the required information about the indexes to be built.

Java SDK

The Java SDK provides developers with the facility for building applications with the Java 5 and 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. Using the Eclipse software installer, it is possible to download and install Java SDK, Google Web Toolkit, and Google AppEngine plug-ins into Eclipse. 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 a common development model. Together with servlets, many other features are available to build applications. Moreover, developers can easily create Web applications by using the *Eclipse Web Platform*, which provides a set of tools and components.

The plug-in 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.

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 applications 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 them with the default code editor for editing application files. In addition, the launcher provides access to some important services for application monitoring and analysis, such as the logs, the SDK console, and the dashboard. The log console captures all the information that is logged by the application while it is running. The console SDK provides developers with a Web interface via which they can see the application profile

in terms of utilized resource. This feature is particularly useful because it allows developers to preview the behavior of the applications once they are deployed on AppEngine, and it can be used to tune applications 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 are dozens of available Python Web frameworks that can be used. However, due to the restrictions enforced by the sandboxed environment, all of them cannot be used seamlessly. The *webapp* framework has been reimplemented 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*.[4]

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.

9.2.2.2 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 because it allows unique identification of the application while it's interacting with AppEngine. Developers use an app identifier to upload and update applications. Besides being unique, it also needs to be compliant to 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 that is descriptive of the application; the 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 using either the respective development environment (*GoogleAppEngineLauncher* and *Google AppEngine* plug-in) or the command-line tools. Once the application is uploaded, nothing else needs to be done 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 insight into resource usage (CPU, bandwidth) and services and other useful counters. It is also possible to manage multiple versions of a single application, 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 to set up a billing account and obtain more allowance and be 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.

[4] www.djangoproject.com.

An application is measured against *billable quotas*, *fixed quotas*, and *per-minute quotas*. Google AppEngine uses these quotas 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 the application can carry out on the infrastructure (services and runtime). These quotas are generally bigger than billable quotas and are set by AppEngine to avoid applications impacting each other's performance or overloading the infrastructure. The costing model also includes per-minute quotas, which are defined in order to avoid applications consuming all their credit in a very limited period of time, monopolizing a resource, and creating 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 is replenished. Once a resource is depleted, subsequent requests to that resource will generate an error or an exception. Resources such as CPU time and incoming or outgoing bandwidth will return an "HTTP 403" error page to users; all the other resources and services will generate an exception that can be trapped in code to provide 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 limits, 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, a framework for developing scalable Web applications, 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 that help developers build applications that are easy to scale. One of the characteristic elements of AppEngine is the use of simple interfaces that allow applications to perform specific operations that are optimized and designed to scale. Building on top of these blocks, developers can build applications and let AppEngine scale them out when 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 become familiar with the capabilities of AppEngine and implement the required features in a way that conforms with the AppEngine application model.

9.3 Microsoft Azure

Microsoft Windows Azure is a cloud operating system built on top of Microsoft datacenters' infrastructure and provides developers with a collection of services for building applications with cloud technology. Services range from compute, storage, and networking to application connectivity,

access control, and business intelligence. Any application that is built on the Microsoft technology can be scaled 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.

Figure 9.3 provides an overview of services provided by Azure. These services can be managed and controlled through the *Windows Azure Management Portal*, which acts as an administrative console for all the services offered by 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 made up of a foundation layer and a set of developer services that can be used to build 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 datacenters and accessible through the Windows Azure Management Portal. Alternatively, developers can recreate a Windows Azure environment (with limited capabilities) on their own machines for development and testing purposes. In this section, we provide an overview of the Azure middleware and its services.

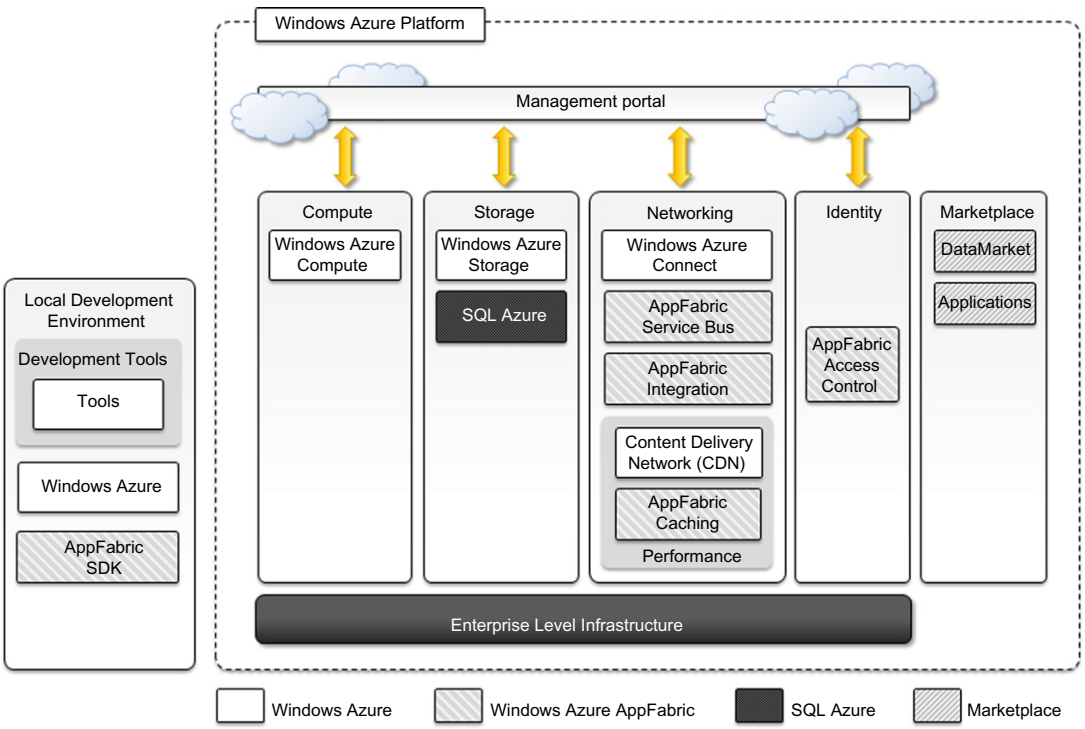

FIGURE 9.3

Microsoft Windows Azure Platform Architecture.

9.3.1.1 Compute services

Compute services are the core components of Microsoft Windows Azure, and they are delivered by means of the abstraction of *roles*. A role is a 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*.

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 on 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 that are not integrated with IIS can still be hosted on Azure (i.e., Java Server Pages on Apache Tomcat), but there is no advantage to using a Web role over a Worker role.

Worker role

Worker roles are designed to host general compute services on Azure. They can be 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 with Web roles.

Developing a worker role is like a developing a service. Compared to a Web role whose computation is triggered by the interaction with an HTTP client (i.e., a browser), a Worker role runs continuously from the creation of its instance 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 its startup as well as being notified of changes in the hosting environment. As with 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.

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 with all the required applications and components, save it into a Virtual Hard Disk (VHD)

Table 9.7 Windows Azure Compute Instances Characteristics, 2011–2012

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 × 1.6 GHz	3.5 GB	490 GB	High	$0.24
Large	4 × 1.6 GHz	7 GB	1,000 GB	High	$0.48
Extra Large	8 × 1.6 GHz	14 GB	2,040 GB	High	$0.96

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.

Compared to the Worker and Web roles, the VM role provides finer control of the compute service and resource that are deployed on the Azure Cloud. An additional administrative effort is required for configuration, installation, and management of services.

9.3.1.2 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 that is useful for the current execution cycle of a role. If the role is restarted and activated on a different physical machine, this information is lost.

Windows Azure provides different types of storage solutions that complement compute services with a more durable and redundant option compared to local storage. Compared to local storage, these services can be accessed by multiple clients at the same time and from everywhere, thus becoming a general solution for storage.

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 or binary files. Two types of blobs are available:

- *Block blobs.* Block blobs are composed of blocks and are optimized for sequential access; therefore they are appropriate for media streaming. Currently, blocks are of 4 MB, and a single block blob can reach 200 GB in 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 into multiple pages or constituted of a single page. This type of blob is optimized for random access and can be used to host data different from streaming. Currently, the maximum dimension of a page blob can be 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, to optimize its distribution, blobs storage can leverage the Windows Azure CDN so that blobs are kept close to users requesting them and can be served efficiently.

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 persistent and durable storage. A page blob mounted as part of an NTFS tree is called an *Azure Drive*.

Tables

Tables constitute a semistructured storage solution, allowing users to store information in the form of entities with a collection of properties. Entities are stored as rows in the table and are 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 are no schema enforcing constraints on the properties of entities and there is no facility for representing relationships among entities. For this reason, tables are more similar to spreadsheets rather than SQL tables.

The service is designed to handle large amounts of data and queries returning huge result sets. This capability is supported by 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.

Currently, 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 a row key and partition keys is 1 KB.

Queues

Queue storage allows applications to communicate by exchanging messages through durable queues, thus avoiding lost or unprocessed messages. Applications enter messages into a queue, and other applications can read them in a first-in, first-out (FIFO) style.

To ensure that messages get processed, when an application reads a message it is marked as invisible; hence it will not be available to other clients. Once the application has completed processing the message, it needs to explicitly delete the message from the queue. This two-phase process ensures that messages get processed before they are removed from the queue, and the client failures do not prevent messages from being processed. At the same time, this is also a reason that the queue does not enforce a strict FIFO model: Messages that are read by applications that crash during processing are made available again after a timeout, during which other messages can be read by other clients. An alternative to reading a message is *peeking*, which allows retrieving the message but letting it stay visible in the queue. Messages that are peeked are not considered processed.

All the services described are geo-replicated three times to ensure their availability in case of major disasters. *Geo-replication* involves the copying of data into a different datacenter that is hundreds or thousands of miles away from the original datacenter.

9.3.1.3 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

multitenancy; 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.

Access control

AppFabric provides the capability of encoding access control to resources in Web applications and services into a set of rules that are expressed outside the application code base. These rules give a great degree of flexibility in terms of the ability to secure components of the application and define 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 easy building of hybrid systems, with some parts existing in the private premises and others deployed in the public cloud.

Service bus

Service Bus constitutes the messaging and connectivity infrastructure provided with AppFabric for building distributed and disconnected applications 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 security and reliability and letting 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 from 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 multicast message delivery in one-way communications, and asynchronous messaging to decouple application components.

In order to leverage these features, applications need to be connected to the bus, which provides these services. A connection is the Service Bus element that is priced by Azure on a pay-as-you-go basis. Users are billed on a connections-per-month basis, and they can buy advance "connection packs," which have a discounted price, if they can estimate their needs in advance.

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 the 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 Windows Azure storage or in SQL Azure. The service implements a distributed in-memory cache of which the 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 by applications. Azure Cache is delivered as a service, and it can be

easily integrated with applications. This is a 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 the size of cache allocated by applications per month, despite their effective use of the cache. Currently, several cache sizes are available, ranging from 128 MB ($45/month) to 4 GB ($325/month).

9.3.1.4 Other services

Compute, storage, and middleware services constitute the core components of the Windows Azure platform. Besides these, other services and components simplify the development and integration of applications with the Azure Cloud. An important area for these services is applications connectivity, including virtual networking and content delivery.

Windows Azure virtual network

Networking services for applications are offered under the name *Windows Azure Virtual Network*, which includes *Windows Azure Connect* and *Windows Azure Traffic Manager*.

Windows Azure Connect allows easy setup of 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 the 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 to 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 are available for free only by invitation.

Windows Azure content delivery network

Windows Azure Content Delivery Network (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 service allows serving of 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 applications built for SQL Server can transparently migrate to SQL Azure. Moreover, the service is fully manageable using REST APIs, allowing developers to control databases deployed in the Azure Cloud as well as the firewall rules set up for their accessibility.

Figure 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

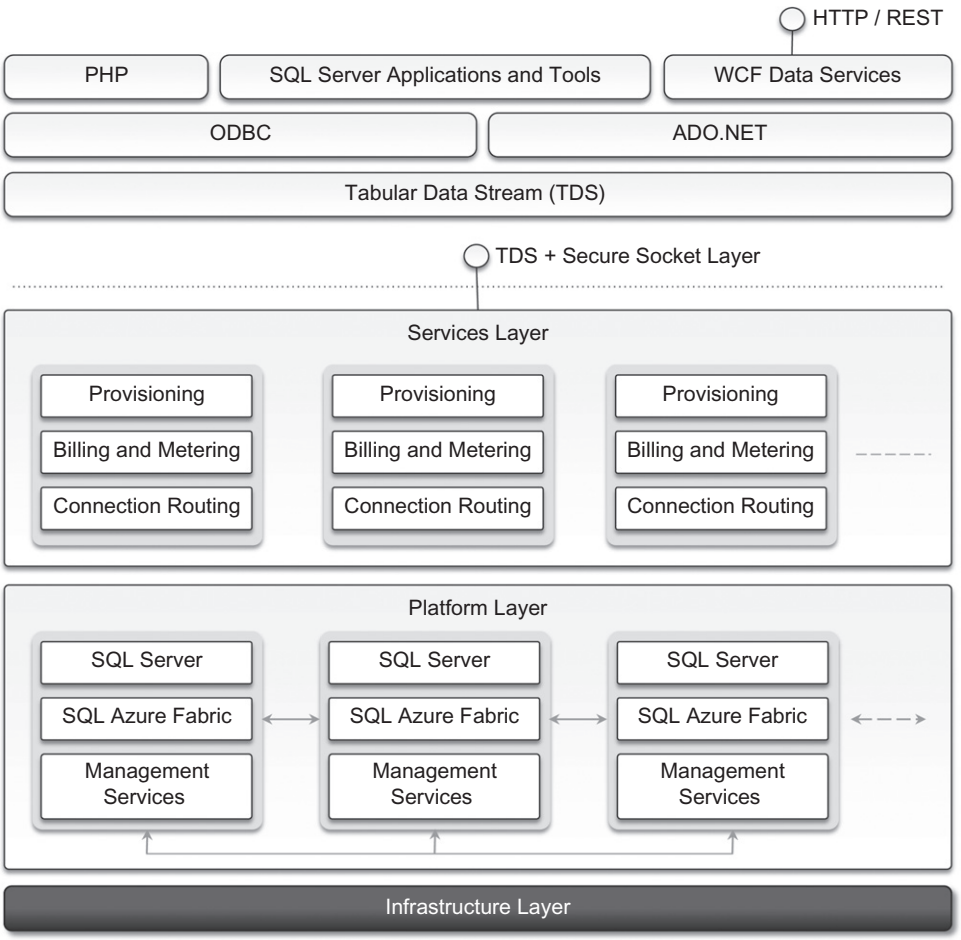

FIGURE 9.4

SQL Azure architecture.

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 datacenters.

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 and logins and to configure access to servers. SQL Azure servers are abstractions

that closely resemble physical SQL Servers: They have a fully qualified domain name under the *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.

Currently, the SQL Azure service is billed according to 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 GB 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 is based on the peak size the database reaches 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 constitutes the cloud infrastructure governing the physical servers of the datacenter. The 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 who want to have their own cloud computing infrastructure.

As introduced earlier, 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 production middleware, but it is designed for developing and testing the functionalities of applications that will eventually be deployed on Azure. The Azure appliance is instead a full-featured implementation of Windows Azure. Its goal is to replicate Azure on a third-party infrastructure and make available its services beyond the boundaries of the Microsoft Cloud. The appliance addresses two major scenarios: institutions that have very large computing needs (such as government agencies) and institutions that cannot afford to transfer their data outside their premises.

9.3.4 Observations

Windows Azure is Microsoft's solution for developing cloud computing applications. Azure is an implementation of the PaaS layer and provides the developer with a collection of services and scalable middleware hosted on Microsoft datacenters that address compute, storage, networking, and identity management needs of applications. The services Azure offers can be used either 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 composed of compute services, storage services, and 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, back-end 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 constitutes the distributed and scalable middleware of Azure.

SQL Azure is another important element of 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 Windows systems, even though other technologies and systems can be supported. For this reason, Azure constitutes the solution of choice for migrating to the cloud applications that are already based on the .NET technology.

SUMMARY

This chapter introduced some 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 (AWS) provides solutions for building infrastructure in the Amazon Cloud. Amazon EC2 and Amazon S3 represent AWS's core value offering. The former allows developers to create virtual servers and customize their computing stack as required. The latter is a storage solution that allows 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 in either Java or Python, first locally 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 deployed on Microsoft datacenters for building dynamically scalable applications. Azure's core components are represented by compute services expressed in terms of roles, storage services, and the AppFabric, 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 semistructured data, Windows Azure also provides storage for relational data by means of the SQL Azure service.

AppEngine and Windows Azure are PaaS solutions. AWS extends its services across all three layers of the Cloud Computing Reference Model, although it is well known for its IaaS offerings, 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 are the differences between Amazon SimpleDB and Amazon RDS?
5. What type of problems does the Amazon Virtual Private Cloud address?
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 types 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 the Windows Azure Platform Appliance? For which kinds of scenarios was this appliance designed?

This page intentionally left blank

CHAPTER

Cloud Applications 10

Cloud computing has gained huge popularity in industry due to its ability to host applications for which the services can be delivered to consumers rapidly at minimal cost. This chapter discusses some application case studies, detailing their architecture and how they leveraged various cloud technologies. Applications from a range of domains, from scientific to engineering, gaming, and 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 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 to use cloud resources is even more appealing because minimal changes need to be made to existing applications in order to leverage cloud resources.

The most relevant option is IaaS 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 with distributed computing middleware capable of interacting with cloud-based infrastructures. PaaS solutions have been considered as well. 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, the 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 in which 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 to support doctors in providing more effective diagnostic processes. In particular, here 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. ECG data analysis and monitoring constitute a case that naturally fits into 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 over time and that represents the heartbeat. The analysis of the shape of the ECG waveform is used to identify arrhythmias and is the most common way to detect heart disease. Cloud computing technologies allow the remote monitoring of a patient's heartbeat data, data analysis in minimal time, and the notification of first-aid personnel and doctors should these data reveal potentially dangerous conditions. This way a patient at risk can be constantly monitored without going to a hospital for ECG analysis. At the same time, doctors and first-aid personnel can instantly be notified of cases that require their attention.

An illustration of the infrastructure and model for supporting remote ECG monitoring is shown in Figure 10.1. Wearable computing devices equipped with ECG sensors constantly monitor the patient's heartbeat. Such information is transmitted to the patient's mobile device, which 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 constitute the SaaS application that will store ECG data in the Amazon S3 service and issue a processing request to the scalable cloud platform. The runtime platform is composed of 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 of 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 heartbeat data and the comparison of the waveform with a reference waveform to detect anomalies. If 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 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 these systems not only ubiquitous, but they can also be easily integrated with other systems maintained on the hospital's premises. Finally, cost savings constitute another reason for the use of cloud technology in healthcare. Cloud services are priced on a pay-per-use basis and with volume prices for 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.

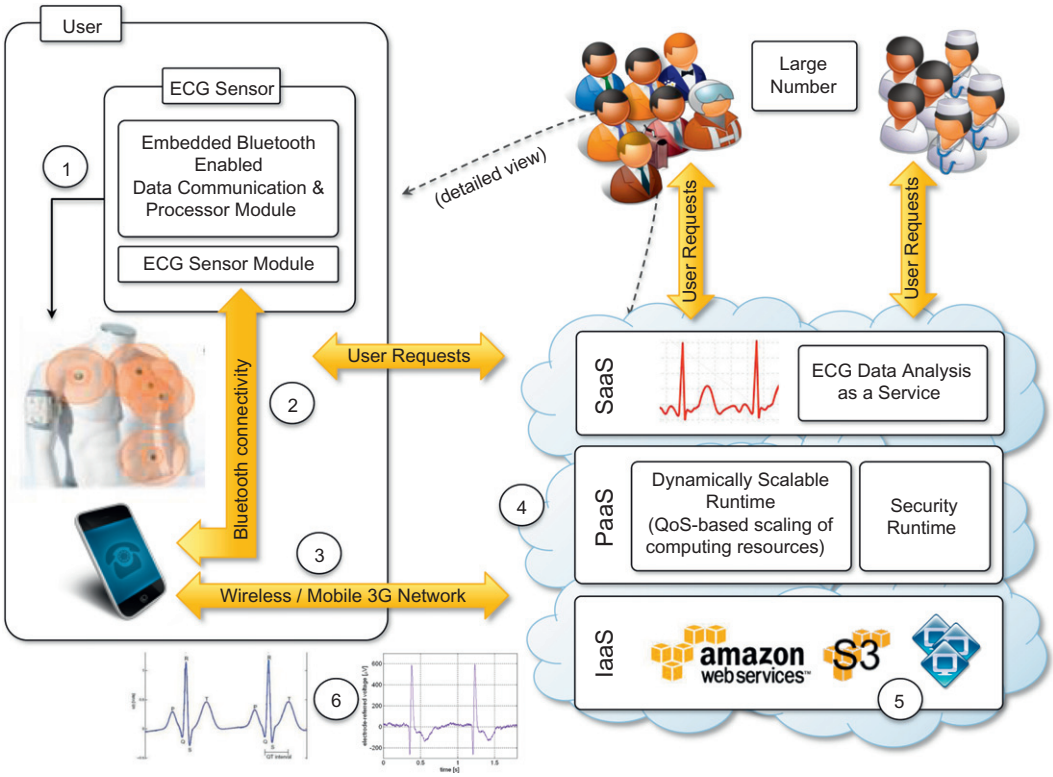

FIGURE 10.1

An online health monitoring system hosted in the cloud.

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, biology applications have often made extensive use of supercomputing and cluster computing infrastructures. Similar capabilities can be leveraged on demand using cloud computing technologies in a more dynamic fashion, thus opening new opportunities for bioinformatics applications.

Protein structure prediction is a computationally intensive task that is fundamental to different types of research in the life sciences. Among these is the design of new drugs for the treatment of diseases. The geometric 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, 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 navigating the bureaucracy to get access to

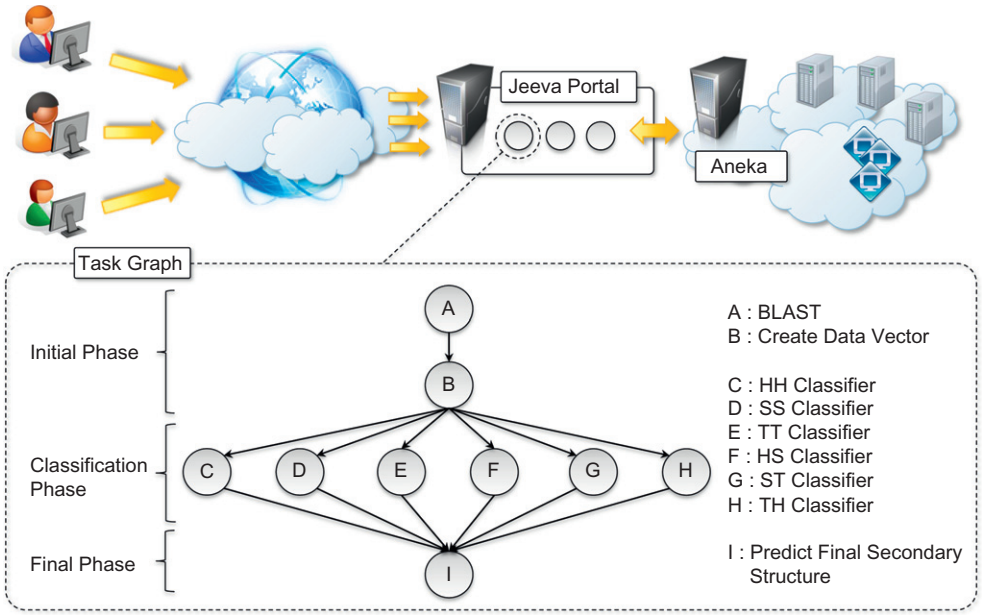

FIGURE 10.2

Architecture and overview of the Jeeva Portal.

parallel and distributed computing facilities. Cloud computing grants access to such capacity on a pay-per-use basis.

One 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 (see Figure 10.2). The prediction task uses machine learning techniques (support vector machines) for determining the secondary structure of proteins. These techniques translate the problem into one of pattern recognition, where a sequence has to be classified into one of 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 to sensibly reduce the computational time of the prediction. The prediction algorithm is then translated into a task graph that is submitted to Aneka. Once the task is completed, the middleware makes the results available for visualization through the portal.

The advantage of using cloud technologies (i.e., Aneka as scalable cloud middleware) versus conventional grid infrastructures is the capability to leverage 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.

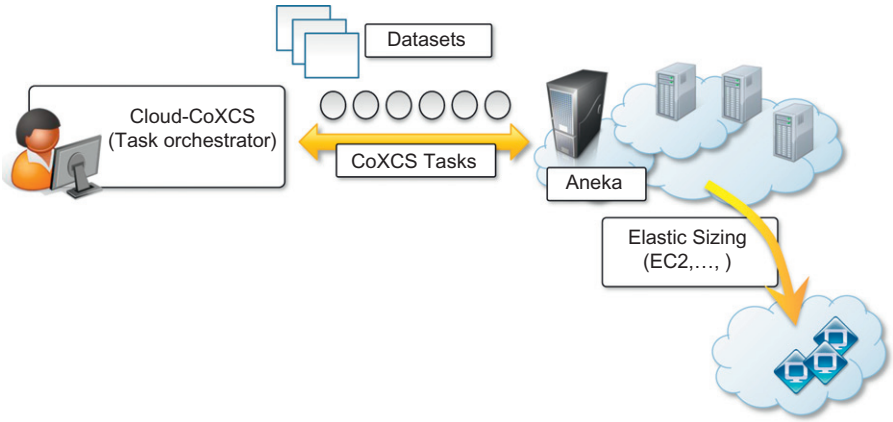

FIGURE 10.3

Cloud-CoXCS: An environment for microarray data processing on the cloud.

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 medical 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 genes regulating the cell growth mutate. 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 datasets ranges from several thousands to over tens of thousands of 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 rules 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 datasets (such as microarray gene expression data sets), has not been explored in detail. A variation of this 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 because the classifications problems on the subdomains can be solved concurrently. Cloud-CoXCS (see Figure 10.3) is a cloud-based implementation of CoXCS that leverages Aneka to solve the classification problems in parallel and compose their outcomes. The algorithm is controlled by strategies, which define the way the outcomes are composed together and whether the process needs to be iterated.

Because of the dynamic nature of XCS, the number of required compute resources to execute it can vary over time. Therefore, the use of scalable middleware such as Aneka offers a distinctive advantage.

10.1.4 Geoscience: satellite image processing

Geoscience applications collect, produce, and analyze massive amounts of geospatial and nonspatial 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 needs 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 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 to support 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 images need to be moved from a 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 scenarios. A cloud-based implementation of such a workflow has been developed by the Department of Space, Government of India [163]. The system shown in Figure 10.4 integrates several technologies across the entire computing stack. A SaaS application provides a collection of services for such tasks as geocode generation and data visualization. At the PaaS level, Aneka controls the importing 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.

10.2 Business and consumer applications

The business and consumer sector is the one that probably benefits the most from cloud computing technologies. On one hand, the opportunity to transform capital costs into operational costs makes clouds an attractive option for all enterprises that are IT-centric. On the other hand, the sense of ubiquity that the 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 up-front 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

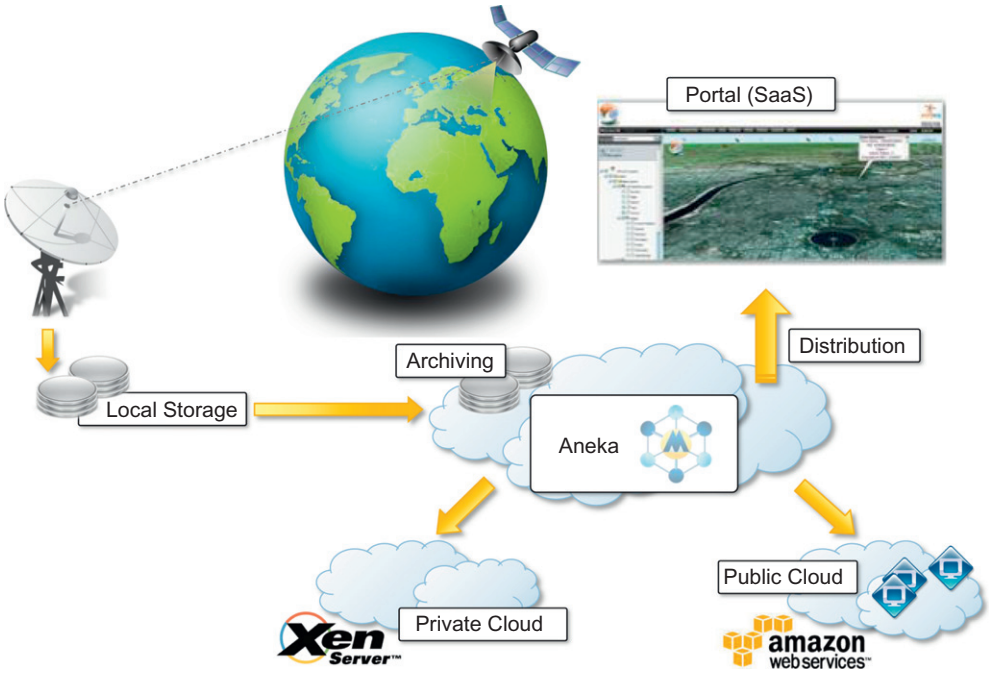

FIGURE 10.4

A cloud environment for satellite data processing.

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 the more mature of the two. Cloud CRM applications constitute a great opportunity for small enterprises and start-ups to have fully functional CRM software without large up-front costs and by paying subscriptions. Moreover, CRM 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 from 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 CRM. 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 cloud-based models is more difficult: he cost advantage over the long term might not be clear, and the switch to

the cloud could be difficult if organizations already have large ERP installations. For this reason cloud ERP solutions are less popular than CRM solutions at this time.

10.2.1.1 Salesforce.com

Salesforce.com is probably the most popular and developed CRM solution available today. As of today more than 100,000 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 scalable and high-performance middleware executing all the operations of all Salesforce.com applications.

The architecture of the Force.com platform is shown in Figure 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, which 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

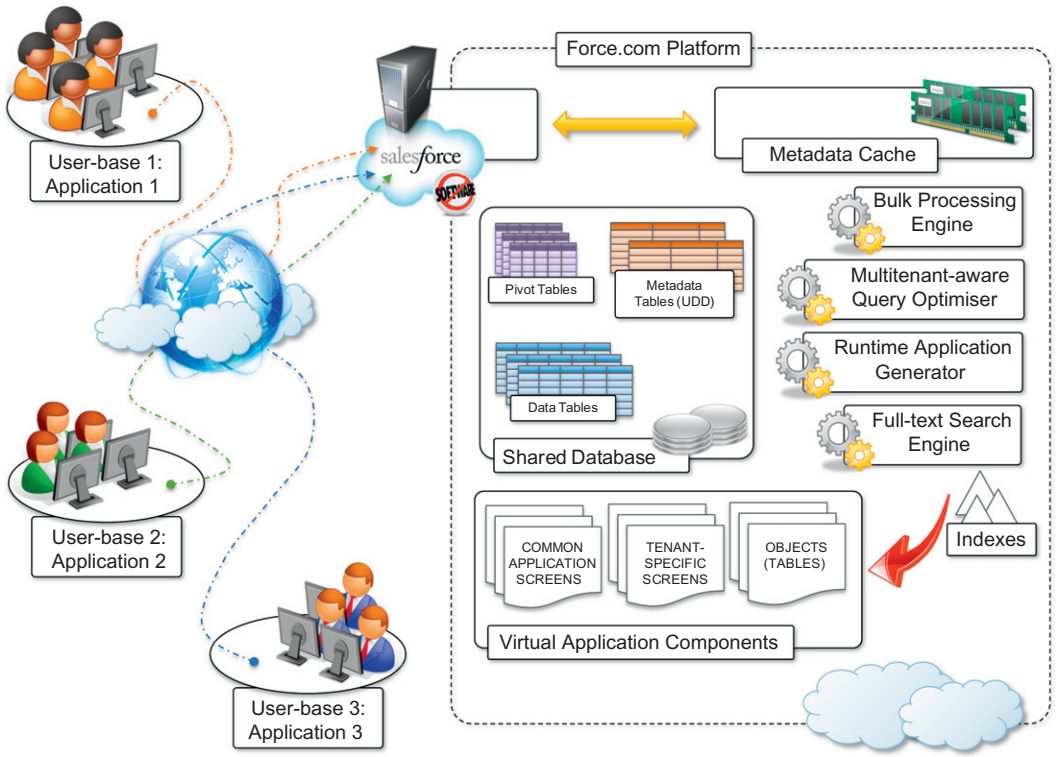

FIGURE 10.5

Salesforce.com and Force.com architecture.

application data are stored in 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 amounts of data that need to be crawled. The search engine maintains its indexing data in a separate store and is 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 that provides 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.

10.2.1.2 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 as a monthly per-user subscription.

The system is completely hosted in Microsoft's datacenters across the world and offers to customers a 99.9% SLA, with bonus credits if 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.

10.2.1.3 NetSuite

NetSuite provides a collection of applications that help customers manage every aspect of the business enterprise. Its offering is divided into three major products: *NetSuite Global ERP*, *NetSuite Global CRM+*, and *NetSuite Global Ecommerce*. Moreover, an all-in-one solution: *NetSuite One World*, integrates all three products together.

The services NetSuite delivers are powered by two large datacenters on the East and West coasts of the United States, connected by redundant links. This allows NetSuite to guarantee 99.5% uptime to its customers. Besides the prepackaged 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 SaaS 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 ecommerce 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 warranties regarding application uptime and availability.

10.2.2 Productivity

Productivity applications replicate in the cloud some of the most common tasks that we are used to performing on our desktop: from document storage to office automation and complete desktop environments hosted in the cloud.

10.2.2.1 Dropbox and iCloud

One of the core features of cloud computing is availability anywhere, at any time, and from any Internet-connected device. Therefore, document storage constitutes a natural application for such technology. Online storage solutions preceded cloud computing, but they never became popular. With the development of cloud technologies, online storage solutions have turned into SaaS applications and become more usable as well as more advanced and accessible.

Perhaps the most popular solution for online document storage is *Dropbox*, an online application that allows users to synchronize any file across any platform and any device in a seamless manner (see Figure 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

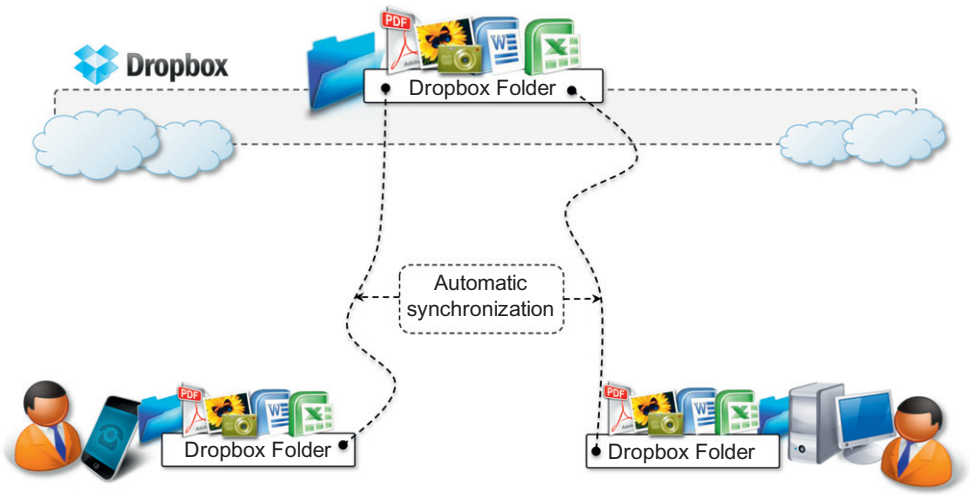

FIGURE 10.6

Dropbox usage scenario.

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*, a cloud-based document-sharing application provided by Apple to synchronize iOS-based devices in a completely transparent manner. Unlike 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 the system to efficiently automate common operations without any human intervention: taking a picture with your iPhone and having it automatically available in iPhoto on your Mac at home; editing a document on the iMac at home and having the changes updated in your iPad. Unfortunately, this capability is limited to iOS devices, and currently there are no plans to provide iCloud with a Web-based interface that would make user content accessible from even unsupported platforms.

There are other solutions for online document sharing, such as *Windows Live*, *Amazon Cloud Drive*, and *CloudMe*, that are popular and that we did not cover. These solutions offer more or less the same capabilities of those we've discussed, with different levels of integration between platform and devices.

10.2.2.2 Google docs

Google Docs is a SaaS application that delivers the basic office automation capabilities with support for collaborative editing over the Web. The application is executed on top of the Google distributed computing infrastructure, which allows the system to dynamically scale according to the number of users using the service.

Google Docs allows users to create and edit text documents, spreadsheets, presentations, forms, and drawings. It aims to replace 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 emailing 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 from any device that is connected to the Internet. Moreover, the suite allows users to work offline if Internet connectivity is not available. Support for various formats such as those that are produced by the most popular desktop office solutions allows users to easily import and move documents in and out of Google Docs, thus eliminating barriers to 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.

10.2.2.3 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, which delivers a considerable amount of its services through the Web browser. Together with the opportunity to leverage 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 in popularity.

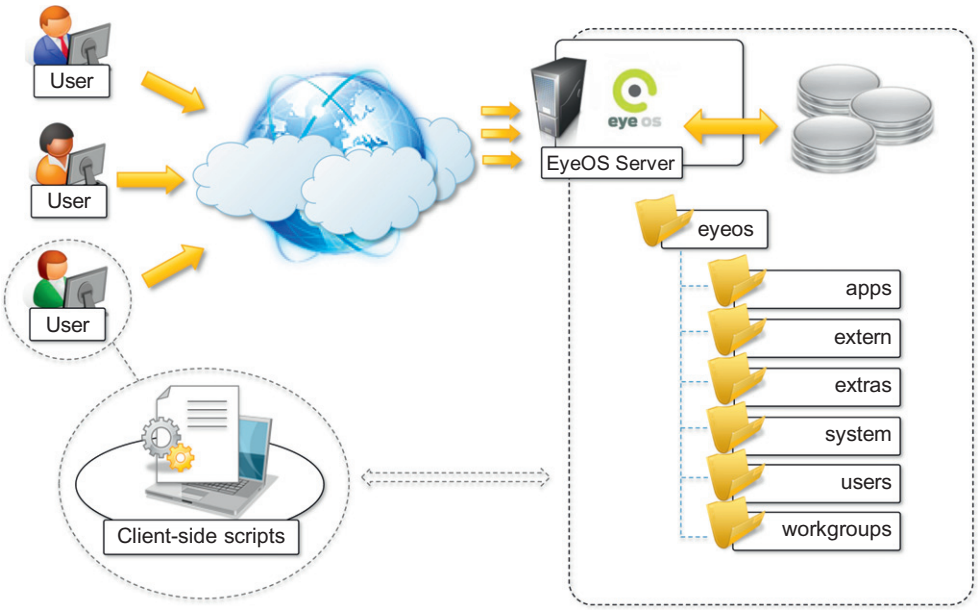

FIGURE 10.7

EyeOS architecture.

EyeOS[1] is one of the most popular Web desktop solutions based on cloud technologies. It replicates the functionalities of a classic desktop environment and comes with pre-installed applications for the most common file and document management tasks (see Figure 10.7). Single users can access the EyeOS desktop environment from anywhere and through any Internet-connected device, whereas organizations can create a private EyeOS Cloud on their premises to virtualize the desktop environment of their employees and centralize their management.

The EyeOS architecture 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 on the server file system. Once the user has logged in by providing credentials, the desktop environment is rendered in the 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; this communication model is used to access user data as well as to perform application operations: editing documents, visualizing images, copying and saving files, sending emails, and chatting.

EyeOS also provides APIs for developing new applications and integrating new capabilities into the system. EyeOS applications are server-side components that are defined by at least two files (stored in the *eyeos/apps/appname* directory): *appname.php* and *appname.js*. The first file defines

[1] www.eyeos.org.

and implements all the operations that the application exposes; 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 differentiator of XIOS/3 is its strong leverage of 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 applications into the environment. XIOS/3 also provides an environment for developing applications (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 open-source software and implements a marketplace 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 an 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 that simplifies collaboration with peers.

10.2.3 Social networking

Social networking applications have grown considerably in the last few years to become the most active sites on the Web. To sustain their traffic and serve millions of users seamlessly, services such as Twitter and 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.

10.2.3.1 Facebook

Facebook is probably the most evident and interesting environment in social networking. With more than 800 million users, it has become one of the largest Websites in the world. To sustain this incredible growth, it has been fundamental that Facebook be capable of continuously adding capacity and developing new scalable technologies and software systems while maintaining high performance to ensure a smooth user experience.

Currently, 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 opportunely modified and refined open-source technologies constitutes the back-end 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 *LAMP* (*Linux*, *Apache*, *MySQL*, and *PHP*). 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, news feeds, notifications, and others. While serving page requests, the *social graph* of the user is composed. The social graph identifies a collection of interlinked information that is of relevance for a given user. Most of the user data are served by querying a distributed cluster of MySQL instances, which mostly contain key-value pairs. These data are then cached for faster retrieval. The rest of the relevant information is then composed together using the services mentioned before. These services are located closer to the data and developed in languages that provide better performance than PHP.

The development of services is facilitated by a set of internally developed tools. 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, who can quickly prototype services and leverage existing ones. 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.

10.2.4.1 Animoto

Animoto[2] 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 a 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 email once the video is rendered.

The core value of Animoto is the ability to quickly create videos with stunning effects without user intervention. A proprietary artificial intelligence (AI) engine, which 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 users don't like the result, the video can be rendered again and the engine will select a different composition, thus producing a different outcome every time. The service allows users to create 30-second videos

[2] www.animoto.com.

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 of different systems that all need to scale (see Figure 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 system's auto-scaling capabilities are managed by Rightscale, which monitors the load and controls the creation of new worker instances as well as their reclaim. Front-end nodes collect the components required to make the video and store them in 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 are 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 4,000 servers on EC2 in peak times without dropping requests but simply causing acceptable temporary delays for the rendering process.

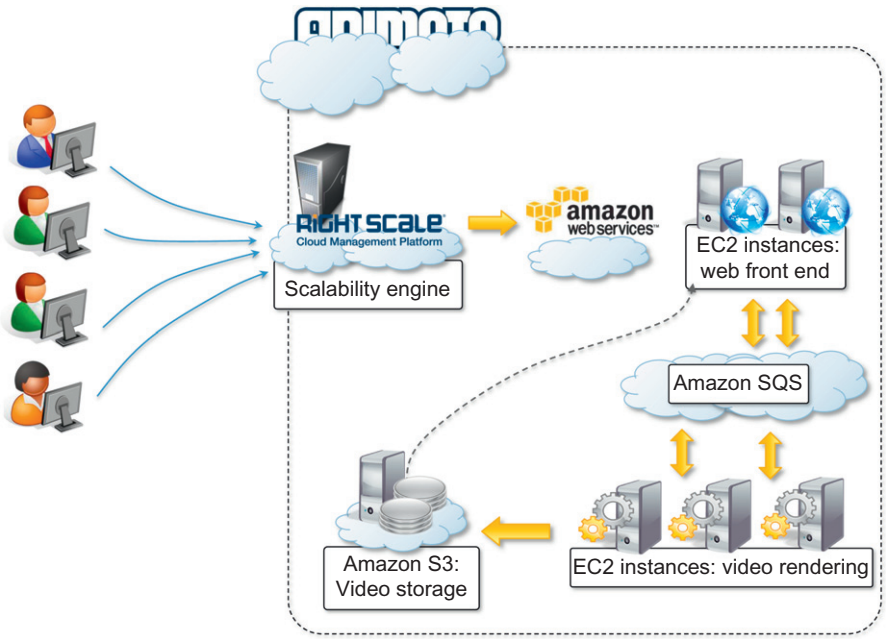

FIGURE 10.8

Animoto reference architecture.

10.2.4.2 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, it is iteratively used to improve the design. It is then fundamental to perform such tasks as fast as possible. Cloud computing provides engineers with the necessary computing power to make this happen.

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 Figure 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 for prototypes requires high-quality, three-dimensional (3D) images. The analysis of these images can help engineers identify problems and correct their design. Three-dimensional rendering tasks take considerable amounts of time, especially in the case of huge numbers of frames, but it is critical for the department to reduce the time spent in these iterations. This goal has been achieved by leveraging cloud computing technologies, which turned the network of desktops in the department into a desktop cloud managed by Aneka.

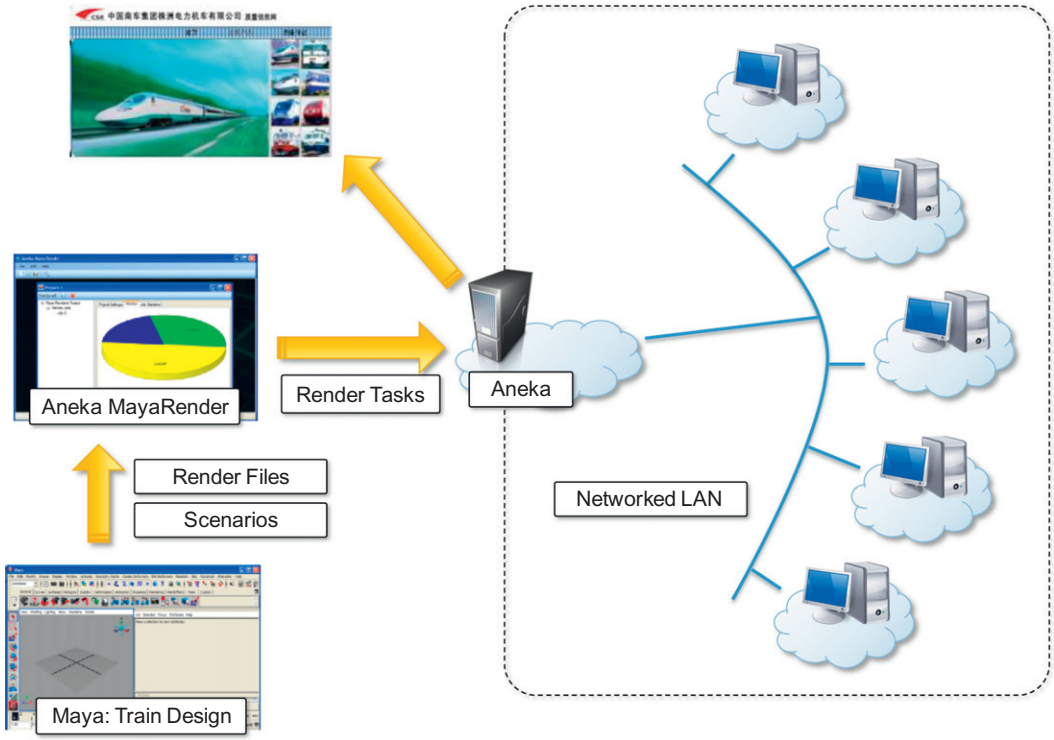

FIGURE 10.9

3D rendering on private clouds.

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, which 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 visualization.

By turning the local network into a private cloud, the resources of which 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.

10.2.4.3 Video encoding on the cloud: Encoding.com

Video encoding and transcoding are operations that can greatly benefit from using cloud technologies: They are computationally intensive and potentially require considerable amounts of storage. Moreover, with the continuous improvement of mobile devices as well as the diffusion of the Internet, requests for video content have significantly increased. The variety of devices with video playback capabilities has led to an explosion of 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 a variety of workflows or made available to everyone according to their needs.

Encoding.com is a software solution that offers video-transcoding services on demand and leverages cloud technology to provide both the horsepower required for video conversion and the storage for staging videos. The service integrates with both 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: the Encoding.com Website, Web service XML APIs, desktop applications, and watched folders. 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 as the insertion of thumbnails, watermarks, or logos. Moreover, it extends its capabilities to audio and image conversion.

The service provides various pricing options: monthly fee, pay-as-you-go (by batches), and special prices for high volumes. Encoding.com now has more than 2,000 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 who share a common experience by playing together in a virtual environment that extends beyond the boundaries of a normal LAN. Online games support hundreds of players in the same session, made possible by the specific architecture used to forward interactions, which 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 interface with the actions of other players.

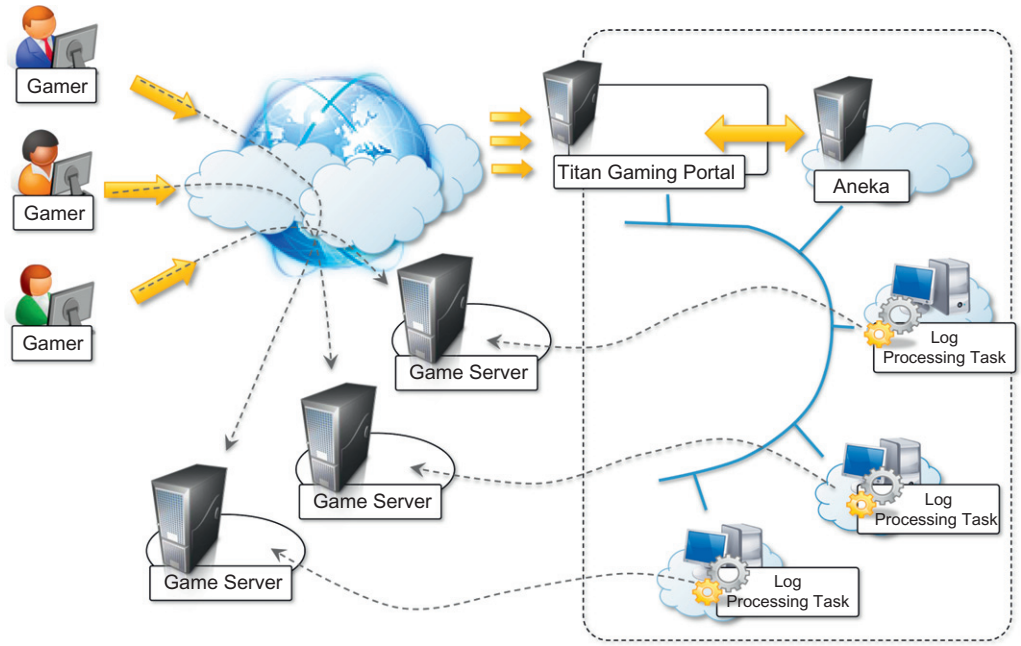

FIGURE 10.10

Scalable processing of logs for network games.

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 amounts 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 for offload game log processing to an Aneka Cloud. The prototype (shown in Figure 10.10) uses a private cloud deployment that allowed Titan Inc. to process concurrently multiple logs and sustain a larger number of users.

SUMMARY

This chapter presented a brief overview of applications developed for the cloud or that leverage cloud technologies in some form. Different application domains, from scientific to business and consumer applications, can take advantage of cloud computing.

Scientific applications take great benefit from the elastic scalability of cloud environments, which also provide the required degree of customization to allow 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 to copy 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 transformed the way we use these applications on a daily basis, but they also introduced new challenges for developers, who have to rethink their designs 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 fundamental advantages does cloud technology bring 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 the field of biology.
5. What are the advantages cloud computing brings to the field of 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 kinds 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 relationship to 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.

This page intentionally left blank

CHAPTER

Advanced Topics in Cloud Computing

11

Cloud computing is a rapidly moving target. New technological advances and application services are regularly introduced. There are many open challenges, especially in the context of energy-efficient management of datacenters and the marketplace for cloud computing.

This chapter presents an overview of various open issues in cloud computing that need long-term investigation. It discusses issues involved in energy-efficient cloud computing and presents a "green" cloud computing architecture. It discusses market models needed for realizing an open market for cloud computing systems from the perspective of federations of clouds and agreements between clouds. A general overview of some of the existing standards that enable interoperation between clouds and a brief look at third-party cloud services are presented.

11.1 Energy efficiency in clouds

Modern datacenters that operate under the cloud computing model are hosting a variety of applications ranging from those that run for a few seconds (e.g., serving requests of Web applications such as ecommerce and social network portals) to those that run for longer periods of time (e.g., simulations or large dataset processing) on shared hardware platforms. The need to manage multiple applications in a datacenter creates the challenge of on-demand resource provisioning and allocation in response to time-varying workloads. Normally, datacenter resources are statically allocated to applications based on peak load characteristics in order to maintain isolation and provide performance guarantees. Until recently, high performance has been the sole concern in datacenter deployments, and this demand has been fulfilled without paying much attention to energy consumption. According to the McKinsey report on "Revolutionizing Data Center Energy Efficiency" [118], a typical datacenter consumes as much energy as 25,000 households. Energy costs of powering a typical data center doubles every five years. Because energy costs are increasing while availability dwindles, there is a need to shift focus from optimizing datacenter resource management for pure performance alone to optimizing for energy efficiency while maintaining high service-level performance (see Figure 11.1).

Datacenters are not only expensive to maintain, they are also unfriendly to the environment. Carbon emissions due to datacenters worldwide are now more than the emissions of both Argentina and the Netherlands [118]. High energy costs and huge carbon footprints are incurred due to the massive amount of electricity needed to power and cool the numerous servers hosted in these

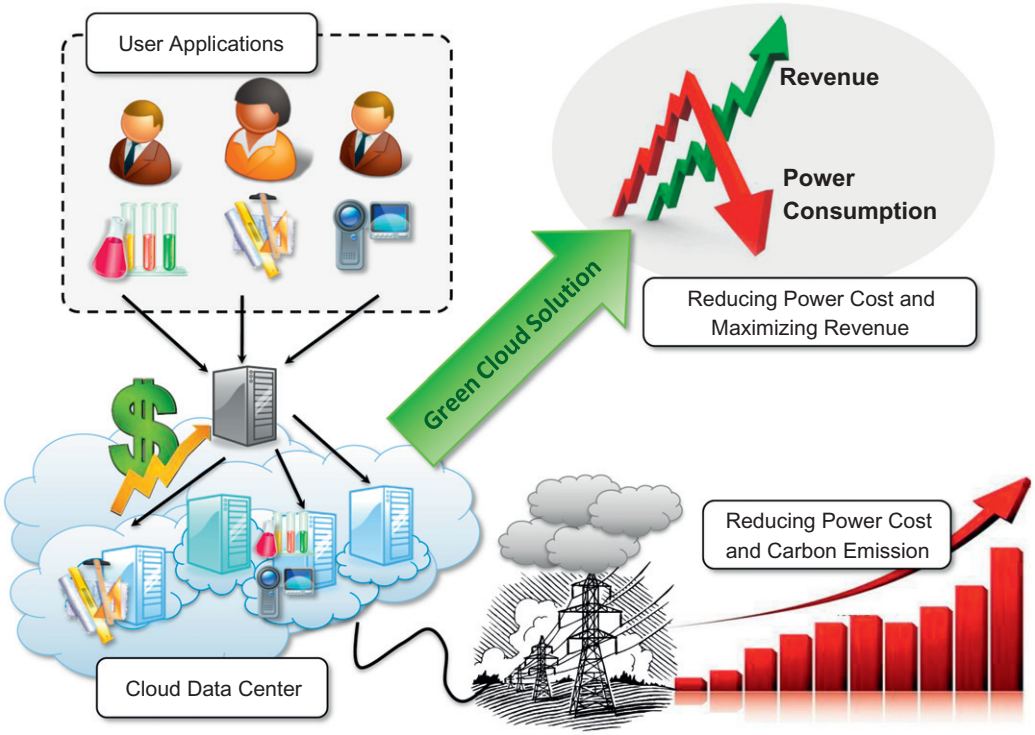

FIGURE 11.1

A "green" cloud computing scenario.

datacenters. Cloud service providers need to adopt measures to ensure that their profit margins are not dramatically reduced due to high energy costs. According to Amazon's estimate, the energy-related costs of its datacenters amount to 42% of the total budget, which includes both direct power consumption and the cooling infrastructure amortized over a 15-year period. As a result, companies such as Google, Microsoft, and Yahoo! are building large datacenters in barren desert land surrounding the Columbia River in the United States to exploit cheap hydroelectric power. There is also increasing pressure from governments worldwide to reduce carbon footprints, which have a significant impact on climate change. To address these concerns, leading IT vendors have recently formed a global consortium, called The Green Grid, to promote energy efficiency for datacenters and minimize their impact on the environment. Pike Research forecasts that datacenter energy expenditures worldwide will reduce from $23.3 billion in 2010 to $16.0 billion in 2020, as well as causing a 28% reduction in greenhouse gas (GHG) emissions from 2010 levels as a result of the adoption of the cloud computing model for delivering IT services.

Lowering the energy usage of datacenters is a challenging and complex issue because computing applications and data are growing so quickly that larger servers and disks are needed to process them fast enough within the required time period. *Green cloud computing* is envisioned to achieve

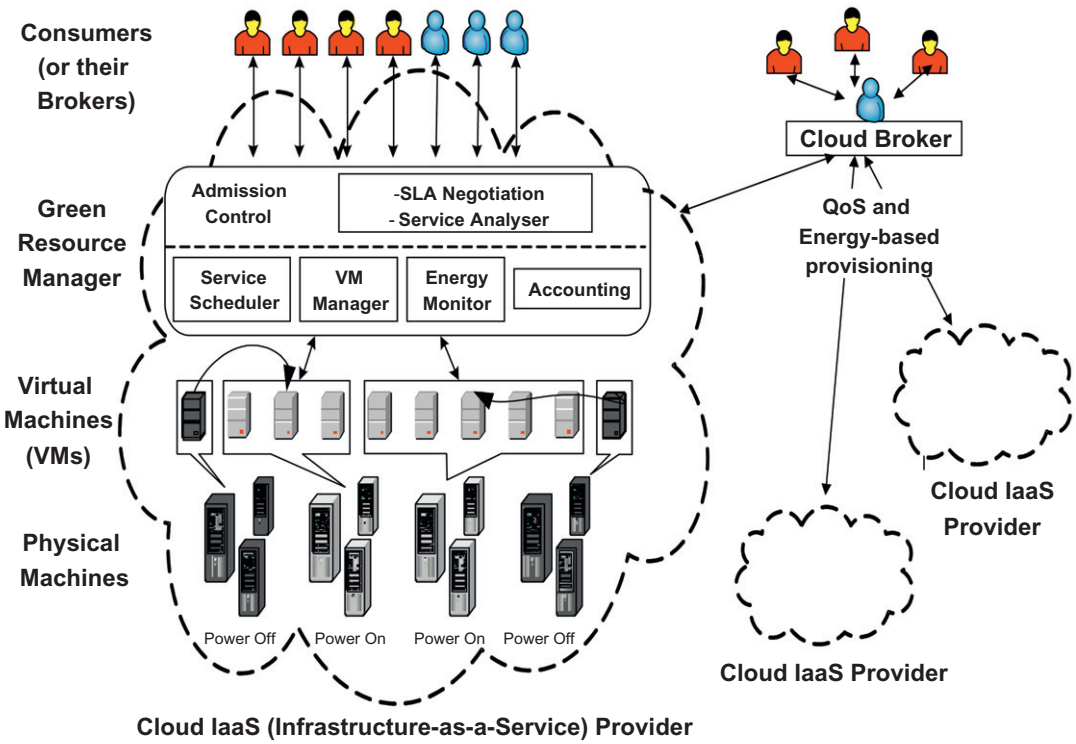

FIGURE 11.2

High-level system architectural framework for green cloud computing.

not only efficient processing and utilization of computing infrastructure but also minimize energy consumption. This is essential for ensuring that the future growth of cloud computing is sustainable. Cloud computing, with increasingly pervasive front-end client devices such as iPhones interacting with back-end datacenters, will cause an enormous escalation in energy usage. To address this problem, datacenter resources need to be managed in an energy-efficient manner to drive green cloud computing. In particular, cloud resources need to be allocated not only to satisfy QoS requirements specified by users via service-level agreements (SLAs) but also to reduce energy usage. This can be achieved by applying market-based utility models to accept user requests that can be fulfilled to enhance revenue along with energy-efficient utilization of cloud infrastructure.

11.1.1 Energy-efficient and green cloud computing architecture

A high-level architecture for supporting energy-efficient resource allocation in a green cloud computing infrastructure is shown in Figure 11.2. It consists of four main components:

- *Consumers/brokers.* Cloud consumers or their brokers submit service requests from anywhere in the world to the cloud. It is important to note that there can be a difference between cloud

consumers and users of deployed services. For instance, a consumer can be a company deploying a Web application, which presents varying workloads according to the number of "users" accessing it.

- *Green Resource Allocator.* Acts as the interface between the cloud infrastructure and consumers. It requires the interaction of the following components to support energy-efficient resource management:
  - *Green Negotiator*. Negotiates with the consumers/brokers to finalize the SLAs with specified prices and penalties (for violations of SLAs) between the cloud provider and the consumer, depending on the consumer's QoS requirements and energy-saving schemes. In Web applications, for instance, the QoS metric can be 95% of requests being served in less than 3 seconds.
  - *Service Analyzer*. Interprets and analyzes the service requirements of a submitted request before deciding whether to accept or reject it. Hence, it needs the latest load and energy information from VM Manager and Energy Monitor, respectively.
  - *Consumer Profiler.* Gathers specific characteristics of consumers so that important consumers can be granted special privileges and prioritized over other consumers.
  - *Pricing*. Decides how service requests are charged to manage the supply and demand of computing resources and facilitate prioritizing service allocations effectively.
  - *Energy Monitor*. Observes and determines which physical machines to power on or off.
  - *Service Scheduler*. Assigns requests to VMs and determines resource entitlements for allocated VMs. It also decides when VMs are to be added or removed to meet demand.
  - *VM Manager.* Keeps track of the availability of VMs and their resource entitlements. It is also in charge of migrating VMs across physical machines.
  - *Accounting*. Maintains the actual usage of resources by requests to compute usage costs. Historical usage information can also be used to improve service allocation decisions.
- *VMs.* Multiple VMs can be dynamically started and stopped on a single physical machine to meet accepted requests, hence providing maximum flexibility to configure various partitions of resources on the same physical machine to different specific requirements of service requests. Multiple VMs can also run concurrently applications based on different operating system environments on a single physical machine. In addition, by dynamically migrating VMs across physical machines, workloads can be consolidated and unused resources can be put on a low-power state, turned off, or configured to operate at low performance levels (e.g., using *Dynamic Voltage and Frequency Scaling*, or DVFS) to save energy.
- *Physical machines.* The underlying physical computing servers provide hardware infrastructure for creating virtualized resources to meet service demands.

11.1.1.1 Energy-aware dynamic resource allocation

Recent developments in virtualization have resulted in its use across datacenters. Virtualization enables dynamic migration of VMs across physical nodes according to QoS requirements. Unused VMs can be logically resized and consolidated on a minimal number of physical nodes, while idle nodes can be turned off (or hibernated). Through consolidation of VMs, large numbers of users can share a single physical server, which increases utilization and in turn reduces the total number of servers required. Moreover, VM consolidation can be applied dynamically by capturing the workload variability and adapting the VM placement at runtime using migration.

Currently, resource allocation in a cloud datacenter aims at providing high performance while meeting SLAs, with limited or no consideration for energy consumption during VM allocations. However, to explore both performance and energy efficiency, two crucial issues must be addressed. First, turning off resources in a dynamic environment puts QoS at risk; aggressive consolidation may cause some VMs to obtain insufficient resources to serve a spike in load. Second, agreed SLAs bring challenges to application performance management in virtualized environments. These issues require effective consolidation policies that can minimize energy use without compromising user QoS requirements. The current approaches to dynamic VM consolidation are weak in terms of providing performance guarantees. One of the ways to prove performance bounds is to divide the problem of energy-efficient dynamic VM consolidation into a few subproblems that can be analyzed individually. It is important to analytically model the problem and derive optimal and near-optimal approximation algorithms that provide provable efficiency. To achieve this goal, clouds need novel analytical models and QoS-based resource allocation algorithms that optimize VM placements with the objective of minimizing energy consumption under performance constraints.

11.1.1.2 InterClouds and integrated allocation of resources

Cloud providers have been deploying datacenters in multiple locations throughout the globe. For example, Amazon EC2 Cloud services are available via Amazon datacenters located in the United States, Europe, and Singapore. This disbursement is leading to the emergence of a notion, called the *InterCloud*, supporting scalable delivery of application services by harnessing multiple datacenters from one or more providers. In addition to enhancing performance and reliability, these InterClouds provide a powerful means of reducing energy-related costs. One reason is that the local demand for electricity varies with time of day and weather. This causes time-varying differences in the price of electricity at each location.Moreover, each site has a different source of energy (such as coal, hydroelectric, or wind), with different environmental costs.This gives scope to adjust the load sent to each location, and the number of servers powered on at each location, to improve efficiency.

In such environments, algorithms that make routing decisions by considering the location of the user, the energy-efficiency of the hardware at each site, the energy mix, and the number of servers currently on at each location are needed. A particularly promising approach is to use this routing to make work "follow the renewables."A major problem with renewable energy is that most sources are intermittent and uncontrollable.Dynamically routing requests to locations with available renewable energy can greatly reduce the nonrenewable energy used and facilitate the widespread use of clean energy.

Sending loads to remote datacenters incurs both delay costs and energy costs due to the increased amounts of data that are transferred over the Internet. Improvements in energy-efficient transport technology should lead to significant reductions in the power consumption of the cloud software services [120].

11.2 Market-based management of clouds

Cloud computing is still in its infancy, and its prominent use is twofold: (1) complete replacement of in-house IT infrastructure and services with the same capabilities rented by service providers;

and (2) elastic scaling of existing computing systems in order to address peak workloads. The efforts in research and industry have been mostly oriented to design and implement systems that actually enable business vendors and enterprises to achieve these goals. The real potential of cloud computing resides in the fact that it actually facilitates the establishment of a market for trading IT utilities. This opportunity until now has been mildly explored and falls in the domain of what it is called *market-oriented cloud computing* [30].

11.2.1 Market-oriented cloud computing

Cloud computing already embodies the concept of providing IT assets as utilities. Then, what makes cloud computing different from *market-oriented* cloud computing? First, it is important to understand what we intend by the term *market*. The *Oxford English Dictionary* (OED)[1] defines a *market* as a "place where a trade is conducted" (Def. I). More precisely, *market* refers to a meeting or a gathering together of people for the purchase and sale of goods. A broader characterization defines the term *market* as the action of buying and selling, a commercial transaction, a purchase, or a bargain. Therefore, essentially the word *market* is the act of trading mostly performed in an environment—either physical or virtual—that is specifically dedicated to such activity.

If we consider the way IT assets and services are consumed as utilities, it is evident that there is a trade-off between the service provider and the consumer; this enables the use of the service by the user under a given SLA. Therefore, cloud computing already expresses the concept of trade, even though the interaction between consumer and provider is not as sophisticated as happens in real markets: Users generally select one cloud computing vendor from among a group of competing providers and leverage its services as long as they need them. Moreover, at present, most service providers have *inflexible pricing*, generally limited to flat rates or tariffs based on usage thresholds. In addition, many providers have proprietary interfaces to their services, thus restricting the ability of consumers to quickly move—and with minimal conversion costs—from one vendor to another. This rigidity, known as *vendor lock-in*, undermines the potential of cloud computing to be an open market where services are freely traded. Therefore, to remove such restrictions, it is required that vendors expose services through standard interfaces. This enables full commoditization and thus would pave the way for the creation of a market infrastructure for trading services.

What differentiates *market-oriented cloud computing (MOCC)* from cloud computing is the presence of a virtual marketplace where IT services are traded and brokered dynamically. This is something that still has to be achieved and that will significantly evolve the way cloud computing services are eventually delivered to the consumer. More precisely, what is missing is the availability of a market where desired services are published and then automatically bid on by matching the requirements of customers and providers. At present, some cloud computing vendors are already moving in this direction[2]; this phenomenon is happening in the IaaS domain—which is the market

[1]The definition of *market* according to the OED can be found at www.oed.com/view/Entry/114178?rskey=13s2aI&result=1#eid (retrieved July 5, 2011).

[2]Amazon introduced the concept of *spot instances* that are dynamically offered by the provider according to their availability and bid on by customers. Their effective usage and consumption is then determined by the spot price established by Amazon and the maximum price provided by the customers.

sector that is more consolidated and mature for cloud computing—but it has not taken off broadly yet. We can clearly characterize the relationship between cloud computing and MOCC as follows:

> *Market Oriented Computing has the same characteristics as Cloud Computing; therefore it is a dynamically provisioned unified computing resource allowing you to manage software and data storage as on aggregate capacity resulting in "real-time" infrastructure across public and private infrastructures. Market Oriented Cloud Computing goes one step further by allowing spread into multiple public and hybrid environments dynamically composed by trading service. [122]*

The realization of this vision is technically possible today but is not probable, given the lack of standards and overall immaturity of the market. Nonetheless, it is expected that in the near future, with the introduction of standards, concerns about security and trust will begin to disappear and enterprises will feel more comfortable leveraging a market-oriented model for integrating IT infrastructure and services from the cloud. Moreover, the presence of a demand-based marketplace represents an opportunity for enterprises to shape their infrastructure for dynamically reacting to workload spikes and for cutting maintenance costs. It also allows the possibility to temporarily lease some in-house capacity during low usage periods, thus giving a better return on investment. These developments will lead to the complete realization of market-oriented cloud computing.

11.2.2 A reference model for MOCC

Market-oriented cloud computing originated from the coordination of several components: service consumers, service providers, and other entities that make trading between these two groups possible. Market orientation not only influences the organization on the global scale of the cloud computing market. It also shapes the internal architecture of cloud computing providers that need to support a more flexible allocation of their resources, which is driven by additional parameters such as those defining the quality of service.

11.2.2.1 A global view of market-oriented cloud computing

A reference scenario that realizes MOCC at a global scale is given in Figure 11.3. It provides guidance on how MOCC can be implemented in practice.

Several components and entities contribute to the definition of a global market-oriented architecture. The fundamental component is the virtual marketplace—represented by the *Cloud Exchange (CEx)*—which acts as a market maker, bringing service producers and consumers together. The principal players in the virtual marketplace are the *cloud coordinators* and the *cloud brokers*. The cloud coordinators represent the cloud vendors and publish the services that vendors offer. The cloud brokers operate on behalf of the consumers and identify the subset of services that match customers' requirements in terms of service profiles and quality of service. Brokers perform the same function as they would in the real world: They mediate between coordinators and consumers by acquiring services from the first and subleasing them to the latter. Brokers can accept requests from many users. At the same time, users can leverage different brokers. A similar relationship can be considered between coordinators and cloud computing services vendors. Coordinators take responsibility for publishing and advertising services on behalf of vendors and can gain benefits from reselling services to brokers. Every single participant has its own utility function, that they all want to optimize rewards. Negotiations and trades are carried out in a secure

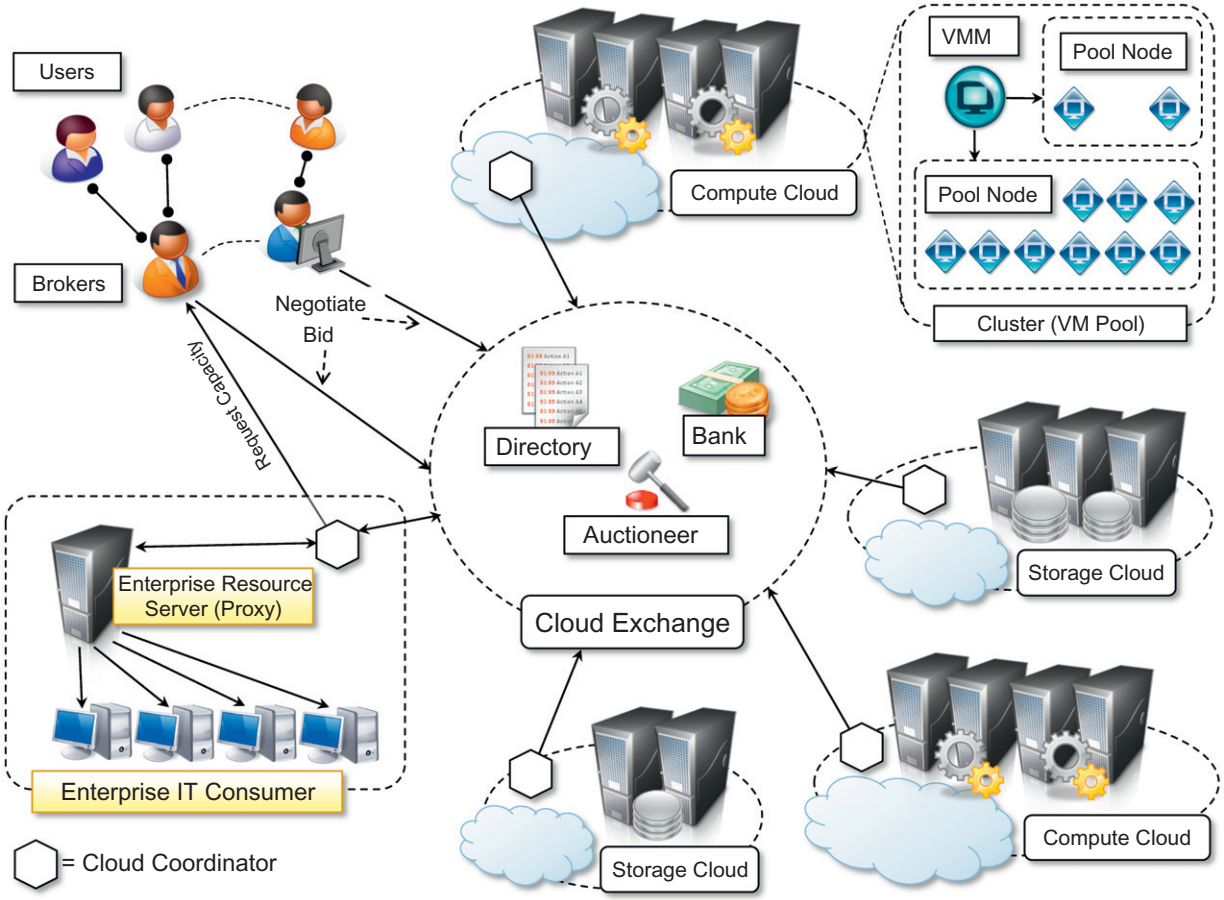

FIGURE 11.3

Market-oriented cloud computing scenario.

and dependable environment and are mostly driven by SLAs, which each party has to fulfill. There might be different models for negotiation among entities, even though the auction model seems to be the more appropriate in the current scenario. The same consideration can be made for the pricing models: Prices can be fixed, but it is expected that they will most likely change according to market conditions.

Several components contribute to the realization of the Cloud Exchange and implement its features. In the reference model depicted in Figure 11.3, it is possible to identify three major components:

- *Directory.* The market directory contains a listing of all the published services that are available in the cloud marketplace. The directory not only contains a simple mapping between service names and the corresponding vendor (or cloud coordinators) offering them. It also provides additional metadata that can help the brokers or the end users in filtering from among the services of interest those that can really meet the expected quality of service. Moreover, several indexing methods can be provided to optimize the discovery of services according to various criteria. This component is modified in its content by service providers and queried by service consumers.
- *Auctioneer.* The auctioneer is in charge of keeping track of the running auctions in the marketplace and of verifying that the auctions for services are properly conducted and that malicious market players are prevented from performing illegal activities.
- *Bank.* The bank is the component that takes care of the financial aspect of all the operations happening in the virtual marketplace. It also ensures that all the financial transactions are carried out in a secure and dependable environment. Consumers and providers may register with the bank and have one or multiple accounts that can be used to perform the transactions in the virtual marketplace.

This organization, as described, constitutes only a reference model that is used to guide system architects and designers in laying out the foundations of a Cloud Exchange system. In reality, the architecture of such a system is more complex and articulated since other elements have to be taken into account. For instance, since the cloud marketplace supports trading, which ultimately involves financial transactions between different parties, security becomes of fundamental importance. It is then important to put in place all the mechanisms that enable secure electronic transactions. These and other aspects are not unique to the design and implementation of MOCC systems but are of concern for any distributed computing system; therefore, they have been only mentioned here.

11.2.2.2 Market-oriented architecture for datacenters

Datacenters are the building blocks of the computing infrastructure that backs the services offered by a cloud computing vendor, no matter its specific category (IaaS, PaaS, or SaaS). In this section, we present these systems by taking into account the elements that are fundamental for realizing computing infrastructures that support MOCC. These criteria govern the logical organization of these systems—rather than their physical layout and hardware characteristics—and provide guidance for designing architectures that are market oriented. In other words, we describe a reference architecture for MOCC datacenters.

Figure 11.4 provides an overall view of the components that can support a cloud computing provider in making available its services on a market-oriented basis [123]. More specifically, the

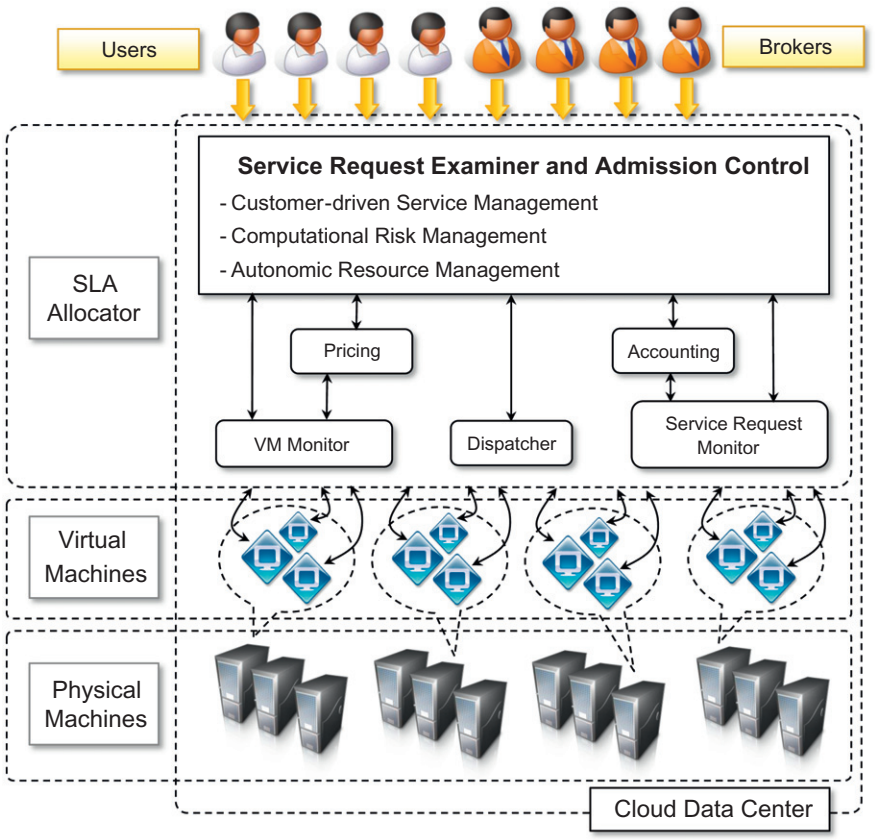

FIGURE 11.4

Reference architecture for a cloud datacenter.

model applies to PaaS and IaaS providers that explicitly leverage virtualization technologies to serve customers' needs. There are four major components of the architecture:

- *Users and brokers.* They originate the workload that is managed in the cloud datacenter. Users either require virtual machine instances to which to deploy their systems (IaaS scenario) or deploy applications in the virtual environment made available to them by the provider (PaaS scenario). These service requests are issued by service brokers that act on behalf of users and look for the best deal for them.
- *SLA resource allocator.* The allocator represents the interface between the datacenter and the cloud service provider and the external world. Its main responsibility is ensuring that service requests are satisfied according to the SLA agreed to with the user. Several components coordinate allocator activities in order to realize this goal:
  - *Service Request Examiner and Admission Control Module.* This module operates in the front-end and filters user and broker requests in order to accept those that are feasible given

the current status of the system and the workload that is already processing. Accepted requests are allocated and scheduled for execution. IaaS service providers allocate one or more virtual machine instances and make them available to users. PaaS providers identify a suitable collection of computing nodes to which to deploy the users' applications.

- *Pricing Module.* This module is responsible for charging users according to the SLA they signed. Different parameters can be considered in charging users; for instance, the most common case for IaaS providers is to charge according to the characteristics of the virtual machines requested in terms of memory, disk size, computing capacity, and the time they are used. It is very common to calculate the usage in time blocks of one hour, but several other pricing schemes exist. PaaS providers can charge users based on the number of requests served by their application or the usage of internal services made available by the development platform to the application while running.
- *Accounting Module.* This module maintains the actual information on usage of resources and stores the billing information for each user. These data are made available to the Service Request Examiner and Admission Control module when assessing users' requests. In addition, they constitute a rich source of information that can be mined to identify usage trends and improve the vendor's service offering.
- *Dispatcher.* This component is responsible for the low-level operations that are required to realize admitted service requests. In an IaaS scenario, this module instructs the infrastructure to deploy as many virtual machines as are needed to satisfy a user's request. In a PaaS scenario, this module activates and deploys the user's application on a selected set of nodes; deployment can happen either within a virtual machine instance or within an appropriate sandboxed environment.
- *Resource Monitor.* This component monitors the status of the computing resources, either physical or virtual. IaaS providers mostly focus on keeping track of the availability of VMs and their resource entitlements. PaaS providers monitor the status of the distributed middleware, enabling the elastic execution of applications and loading of each node.
- *Service Request Monitor.* This component keeps track of the execution progress of service requests. The information collected through the Service Request Monitor is helpful for analyzing system performance and for providing quality feedback about the provider's capability to satisfy requests. For instance, elements of interest are the number of requests satisfied versus the number of incoming requests, the average processing time of a request, or its time to execution. These data are important sources of information for tuning the system.

  The SLA allocator executes the main logic that governs the operations of a single datacenter or a collection of datacenters. Features such as failure management are most likely to be addressed by other software modules, which can either be a separate layer or can be integrated within the SLA resource allocator.

- *Virtual machines (VMs).* Virtual machines constitute the basic building blocks of a cloud computing infrastructure, especially for IaaS providers. VMs represent the unit of deployment for addressing users' requests. Infrastructure management software is in charge of keeping operational the computing infrastructure backing the provider's commercial service offering. As we discussed, VMs play a fundamental role in providing an appropriate hosting environment for users' applications and, at the same time, isolate application execution from the infrastructure, thus preventing applications from harming the hosting environment. Moreover,

VMs are among the most important components influencing the QoS with which a user request is served. VMs can be tuned in terms of their emulated hardware characteristics so that the amount of computing resource of the physical hardware allocated to a user can be finely controlled. PaaS providers do not directly expose VMs to the final user, but they may internally leverage virtualization technology in order to fully and securely utilize their own infrastructure. As previously discussed, PaaS providers often leverage given middleware for executing user applications and might use different QoS parameters to charge application execution rather than the emulated hardware profile.

- *Physical machines.* At the lowest level of the reference architecture resides the physical infrastructure that can comprise one or more datacenters. This is the layer that provides the resources to meet service demands.

This architecture provides cloud services vendors with a reference model suitable to enabling their infrastructure for MOC. As mentioned, these observations mostly apply to PaaS and IaaS providers, whereas SaaS vendors operate at a higher abstraction level. Still, it is possible to identify some of the elements of the SLA resource allocator, which will be modified to deal with the services offered by the provider. For instance, rather than linking user requests to virtual machine instances and platform nodes, the allocator will be mostly concerned with scheduling the execution of requests within the provider's SaaS framework, and lower layers in the technology stack will be in charge of controlling the computing infrastructure. Accounting, pricing, and service request monitoring will still perform their roles.

Regardless of the specific service offering category, the reference architecture discussed here aims to support cloud computing vendors in delivering commercial solutions that are able to [123]:

- Support customer-driven service management based on customer profiles and requested service requirements
- Define computational risk management tactics to identify, assess, and manage risks involved in the execution of applications with regard to service requirements and customer needs
- Derive appropriate market-based resource management strategies encompassing both customer-driven service management and computational risk management in order to sustain SLA-oriented resource allocation
- Incorporate autonomic resource management models that effectively self-manage changes in service requirements to satisfy both new service demands and existing service obligations
- Leverage—when appropriate—VM technology to dynamically assign resource shares according to service requirements

These capabilities are of fundamental importance for competitiveness in the cloud computing market and for addressing a scenario characterized by dynamic SLA negotiations as envisioned by MOCC. Currently, there is no or limited support for such a dynamic negotiation model, which constitutes the next step toward the full realization of cloud computing.

11.2.3 Technologies and initiatives supporting MOCC

Existing cloud computing solutions have very limited support for market-oriented strategies to deliver services to customers. Most current solutions mainly focused on enabling cloud computing

concern the delivery of infrastructure, distributed runtime environments, and services. Since cloud computing has been recently adopted, the consolidation of the technology constitutes the first step toward the full realization of its promise. Until now, a good deal of interest has been directed toward IaaS solutions, which represent a well-consolidated sector in the cloud computing market, with several different players and competitive offers. New PaaS solutions are gaining momentum, but it is harder for them to penetrate the market dominated by giants such as Google, Microsoft, and Force.com.

11.2.3.1 Framework for trading computing utilities

From an academic point of view, a considerable amount of research has been carried out in defining models that enable the trading of computing utilities, with a specific focus on the design of market-oriented schedulers for grid computing systems. As discussed in the introduction, computing grids aggregate a heterogeneous set of resources that are geographically distributed and might belong to different organizations. Such resources are often leased for long-term use by means of agreements among these organizations. Within this context, market-oriented schedulers, which are aware of the price of a given computing resource and schedule user's applications according to their budgets, have been investigated and implemented. The research in this area is of relevance to MOCC, since cloud computing leverages preexisting distributed computing technologies, including grid computing.

Garg and Buyya [124] have provided a complete taxonomy and analysis of such schedulers, which is reported in Figure 11.5. A major classification categorizes these schedulers according to *allocation decision*, *objective*, *market model*, *application model*, and *participant focus*. Of particular interest is the classification according to the market model, which is the mechanism used for trading between users and providers. Along this dimension, it is possible to classify the schedulers into the following categories:

- *Game theory.* In market models that are based on game theory, participants interact in the form of an allocation game, with different payoffs as a result of specific actions that employ various strategies.
- *Proportional share*. This market model originates from proportional share scheduling, which aims to allocate jobs fairly over a set of resources. This original concept has been contextualized within a market-oriented scenario in which the shares of the cluster are directly proportional to the user's bid.
- *Commodity market.* In this model the resource provider specifies the price of resources and charges users according to the amount of resources they consume. The provider's determination of the price is the result of a decision process involving investment and management costs, current demand, and supply. Moreover, prices might be subject to vary over time.
- *Posted price.* This model is similar to the commodity market, but the provider may make special offers and discounts to new clients. Furthermore, with respect to the commodity market, prices are fixed over time.
- *Contract-Net.* In market models based on the Contract-Net [125] protocol, users advertise their demand and invite resource owners to submit bids. Resource owners check these advertisements with respect to their requirements. If the advertisement is favorable to them, the providers will respond with a bid. The user will then consolidate all the bids and compare them to select those

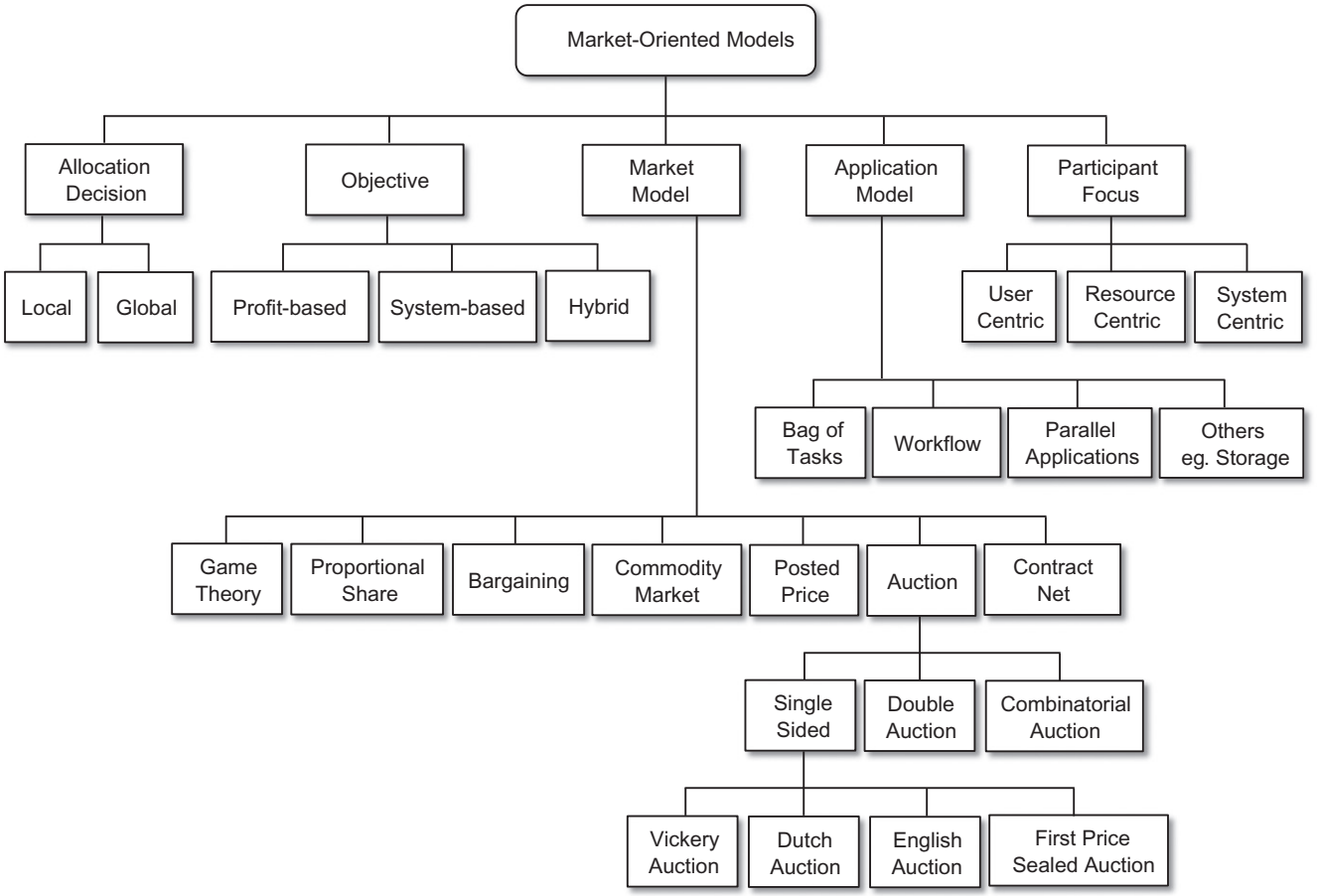

FIGURE 11.5

Market-oriented scheduler taxonomy.

most favorable to him. The providers are then informed about the outcome of their bids, which can be acceptance or rejection.

- *Bargaining*. In market models based on bargaining, the negotiation among resource consumers and providers is carried out until a mutual agreement is reached or it is stopped when either of the parties is no longer interested.
- *Auction.* In market models based on auctions, the price of resources is unknown, and competitive bids regulated by a third party—the auctioneer—contribute to determining the final price of a resource. The bid that ultimately sets the price of a resource is the winning bid, and the corresponding user gains access to the resource.

The most popular and interesting market models for trading computing utilities are the *commodity market*, *posted price*, and *auction* models. Commodity market and posted price models, or variations/combinations of them, are driving the majority of cloud computing services offerings today. Auction-based models can instead potentially constitute the reference market-models for MOCC, since they are able to seamlessly support dynamic negotiations.

Academic research has led to the development of a considerable number of software projects that implement either user resource brokers or resource management systems able to trade computing utilities according to one, or some, of the market models described previously. Some of them—namely *SHARP* [126], *Tycoon* [127], *Bellagio* [128], and *Shirako* [129]—have focused on trading VM-based resource slices and are more likely to be applicable within the MOCC scenario. In addition, it is worth noting some relevant research projects focusing on resource brokering, such as *Nimrod-G* [164] and *Gridbus Broker* [15], which have evolved to integrate capabilities for leasing cloud computing resources.

11.2.3.2 Industrial implementations

Even though market-oriented models have been mostly developed in the academic domain, industrial implementations of some aspects of MOCC are becoming available and gaining popularity. In particular, some interesting initiatives show how different aspects of MOCC, such as flexible pricing models, virtual market place, and market directories, have been made available to the wider public.

Flexible pricing models: amazon spot instances

Amazon Web Services (AWS), one of the biggest players in the IaaS market, recently introduced the concept of spot instances,[3] which allows EC2 customers to bid on unused Amazon EC2 capacity and run those instances for as long as their bid exceeds the current spot price. The spot price varies periodically according to the supply of and demand for EC2 instances and is kept constant within a single hour block. Spot instances can be terminated at any time, and they are usually priced at a lower price with respect to the traditional (on-demand and reserved) instances, since they rely on exceeding capacity available in the EC2 infrastructure. Therefore, it is the responsibility of the user to periodically persist the state of applications executing within spot instances. Spot instances represent an interesting opportunity for both Amazon and EC2 users to benefit from the current condition of the market: The provider can make revenue from a capacity that would have

[3]Full details about the spot instances service offering can be found in Amazon Web Services at the following link: http://aws.amazon.com/ec2/spot-instances/ (retrieved May 9, 2011).

been wasted if priced at the normal level, and the consumer has the opportunity to pay less by taking major risks.

Despite their volatile nature, spot instances have been demonstrated to be reasonably reliable and usable for performing tasks that have a lower priority and are not critical. In other words, they are suitable for applications that can tolerate QoS limitations. Moreover, they are profitably used to extend the capacity of an existing infrastructure at lower costs [130].

Virtual market place: SpotCloud

SpotCloud[4] is an online portal that implements a virtual marketplace, where sellers and buyers can register and trade cloud computing services. The platform is a market place operating in the IaaS sector. Buyers are looking for compute capacity that can meet the requirements of their applications, while sellers can make available their infrastructure to serve buyers' needs and earn revenue. *SpotCloud* provides a comprehensive set of features that are expected for a virtual marketplace. Some of them include:

- Detailed logging of all the buyers' transactions
- Full metering, billing for any capacity
- Full control over pricing and availability of capacity in the market
- Management of quotas and utilization levels for providers
- Federation management (many providers, many customers, but one platform)
- Hybrid cloud support (internal and external resource management)
- Full market administration and reporting
- Applications and pre-build appliances directories

Besides being an online portal, the virtual market realized by SpotCloud can also be replicated in the private premises. Transactions are carried out with real money and based on credit that buyers and sellers must top up once they create an account.

SpotCloud is the most representative implementation of a platform that enables MOCC, even though with some limitations. SpotCloud's working principle is a common and unique platform that sellers need to share in order to join the portal and make available computing capacity. SpotCloud currently supports Enomaly ECP[5] and OpenStack.[6]

Market directories: AppSpot, the cloud market

SpotCloud is implemented as an application hosted on *AppSpot*. This is a huge portal serving applications built on top of the Google *AppEngine* infrastructure: *appspot.com* is a namespace under which all the scalable Web applications developed with the Google AppEngine technology are made available to the community of Internet users. A solution that is more oriented toward listing available cloud building blocks is *The Cloud Market*,[7] which features a comprehensive listing of Amazon EC2 images. Even though these solutions do not provide a complete implementation of a

[4] www.spotCloud.com.
[5] www.enomaly.com.
[6] www.openstack.org.
[7] www.theCloudmarket.com.

market directory, they constitute a step toward the realization of MOCC, since they provide a means to easily locate components useful in building cloud computing systems.

11.2.3.3 A case study: the cloudbus toolkit

An interesting case study coming from research is the *Cloudbus* toolkit [131]. This is a collection of technologies and components that comprehensively try to address the challenges involved in realizing the vision of market-oriented cloud computing.

Figure 11.6 provides a comprehensive view of the Cloudbus components and how they interact together to provide a platform for MOCC. Real-life applications belonging to various scenarios, such as finance, science, education, engineering, multimedia, and others, require cloud computing facilities for processing their workloads and storing data.

The Cloudbus toolkit can act as a general front-end by providing mediated access to cloud computing services that best serve applications' needs. It does so by making available tools and technologies to implement a service-brokering infrastructure and middleware for deploying applications in the cloud. Brokering services are implemented by the Market Maker, which allows users to take full advantage of the cloud marketplace. It relies on different middleware implementations to fulfill the requests of users; these can be either Cloudbus technologies or third-party implementations. Technologies such as Aneka or Workflow Engine provide services for executing applications in the cloud. These can be public clouds, private intranets, or datacenters that can all be uniformly managed within an *InterCloud* [114] realm, which federates computing clouds that belong to different organizations into a unique domain characterizing agreements among parties.

Table 11.1 provides an overview of all the components that constitute the *Cloudbus* Toolkit, together with a brief description of their function. The toolkit also includes facilities for developing algorithms and deployments in a simulated environment by means of *CloudSim* [132]. This is a toolkit that enables users to simulate many aspects of cloud computing environments, from the basic building blocks of a cloud (datacenters, computing nodes, cores, network, and virtual machines) to the resource allocation algorithms and policies. As an example of its versatility, the toolkit has been used to simulate power-aware scheduling policies [133] for reducing energy consumption in large datacenters.

11.2.4 Observations

In this section we have briefly reviewed the fundamental concepts of market-oriented cloud computing. As with any technology, there exists a consolidation phase during which the concepts laying out its foundations are investigated, experimented, absorbed, and leveraged. Cloud computing is a relative evolution of distributed computing and has provided new ways to deliver IT infrastructure and services in a more efficient way to both enterprise and single users. Initially, considerable development interest and effort have been oriented toward the consolidation of the IaaS sector and its integration within the existing computing systems and workflow processes. Currently, PaaS solutions are gaining popularity and being used more consistently to deliver commercial and mainstream software applications and systems. This is the appropriate context where technologies and systems supporting MOCC can be implemented and made available. Most of the work done in this sense still belongs to the domain of academic research, even though the first commercial and industrial implementations are reaching the wider audience.

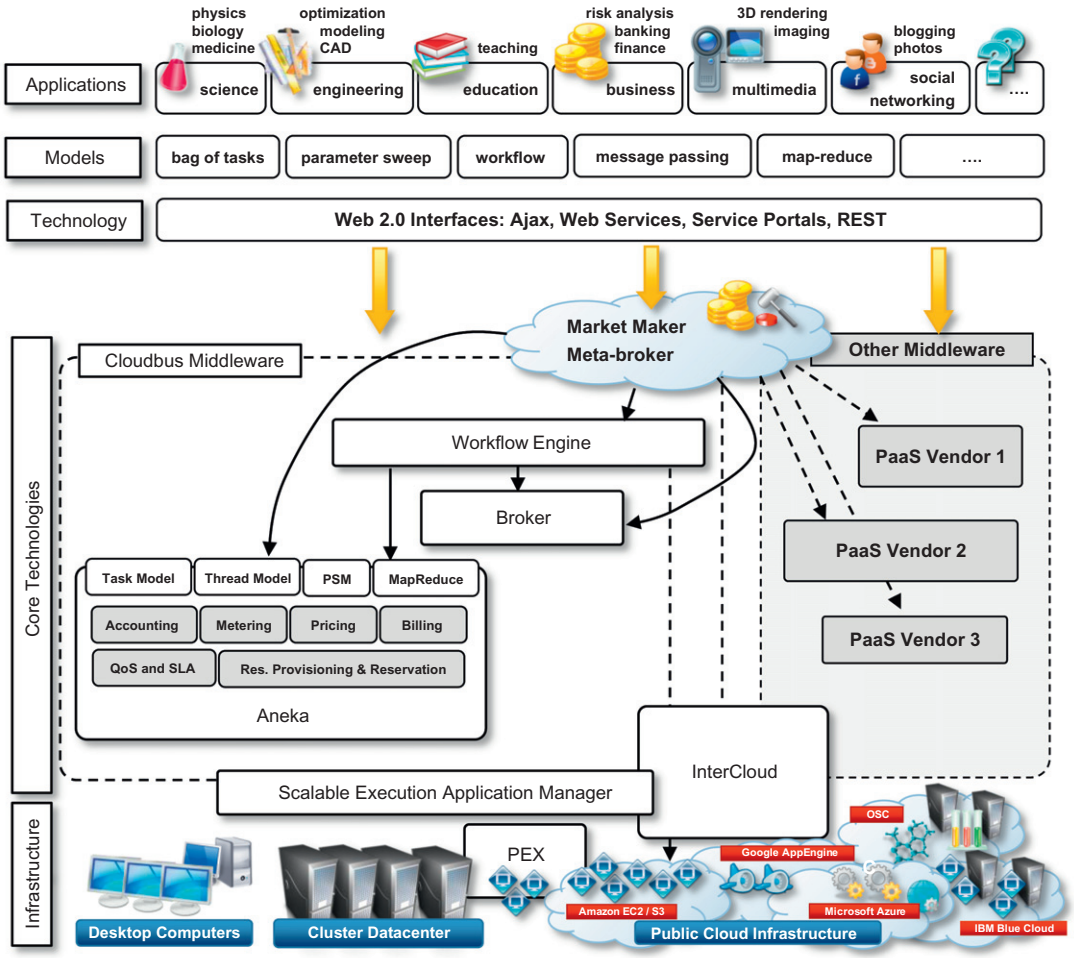

FIGURE 11.6

Cloudbus Toolkit.

11.3 Federated clouds/InterCloud

In the previous section we discussed how to define models and systems for trading computing resources as utilities and how cloud computing systems can interact with each other to make this trading happen. In this section, we address the same problem from the administrative and organizational points of view and introduce the concept of *cloud federation* and the *InterCloud*. These are enablers for MOCC since they provide means for interoperation among different cloud providers.

Cloud computing strongly implies the presence of financial agreements between parties, since services are available on demand on a pay-per-use basis. Nonetheless, the concepts characterizing

Table 11.1 Cloudbus Toolkit Components and Technologies

Technology	Description
Aneka	Middleware for cloud applications development and deployment.
Broker	Middleware for scheduling distributed applications across heterogeneous systems based on the bag-of-tasks model.
Workflow management system	Middleware for the execution, composition, management, and monitoring of workflows across heterogeneous systems.
Market Maker/Meta-Broker	A matchmaker that matches the user's requirements with service providers' capabilities within the context of a marketplace.
InterCloud	A framework for the federation of independent computing clouds.
MetaCDN	Middleware that leverages storage clouds for intelligently delivering users' content based on their QoS and budget preferences.
Energy-efficient computing	Ongoing research on developing techniques and technologies for addressing scalability and energy efficiency.

cloud federation and the InterCloud are applicable, with some limitations, to building aggregations of clouds that belong to different administrative domains.

11.3.1 Characterization and definition

The terms *cloud federation* and *InterCloud*, often used interchangeably, convey the general meaning of an aggregation of cloud computing providers that have separate administrative domains. It is important to clarify what these two terms mean and how they apply to cloud computing.

The term *federation* implies the creation of an organization that supersedes the decisional and administrative power of the single entities and that acts as a whole.[8] Within a cloud computing context, the word *federation* does not have such a strong connotation but implies that there are agreements between the various cloud providers, allowing them to leverage each other's services in a privileged manner. A definition of the term *cloud federation* was given by Reuven Cohen, founder and CTO of Enomaly Inc.[9] :

> *Cloud federation manages consistency and access controls when two or more independent geographically distinct Clouds share either authentication, files, computing resources, command and control or access to storage resources.*

This definition is broad enough to include all the different expressions of cloud services aggregations that are governed by agreements between cloud providers, rather than composed by the user.

InterCloud is a term that is often used interchangeably to express the concept of Cloud federation. It was introduced by Cisco for expressing a composition of clouds that are interconnected by

[8]This is the definition given by the Merriam-Webster online dictionary: www.merriam-webster.com/dictionary/federation (retrieved May 10, 2011).
[9]www.elasticvapor.com/2008/08/standardized-Cloud.html (retrieved May 10, 2011).

means of open standards to provide a universal environment that leverages cloud computing services. By mimicking the Internet term, often referred as the "network of networks," InterCloud represents a "Cloud of Clouds"[10] and therefore expresses the same concept of federating together clouds that belong to different administrative organizations. Whereas this is in many cases acceptable, some practitioners and experts—like Ellen Rubin, founder and VP of Products at CloudSwitch[11] —prefer to give different connotations to the two terms:

> *The primary difference between the InterCloud and federation is that the InterCloud is based on future standards and open interfaces, while federation uses a vendor version of the control plane. With the InterCloud vision, all Clouds will have a common understanding of how applications should be deployed. Eventually workloads submitted to a Cloud will include enough of a definition (resources, security, service level, geo-location, etc.) that the Cloud is able to process the request and deploy the application. This will create the true utility model, where all the requirements are met by the definition and the application can execute "as is" in any Cloud with the resources to support it.*

Therefore, the term *InterCloud* refers mostly to a global vision in which interoperability among different cloud providers is governed by standards, thus creating an open platform where applications can shift workloads and freely compose services from different sources. On the other hand, the concept of a *cloud federation* is more general and includes *ad hoc* aggregations between cloud providers on the basis of private agreements and proprietary interfaces.

11.3.2 Cloud federation stack

Creating a cloud federation involves research and development at different levels: conceptual, logical and operational, and infrastructural. Figure 11.7 provides a comprehensive view of the challenges faced in designing and implementing an organizational structure that coordinates together cloud services that belong to different administrative domains and makes them operate within a context of a single unified service middleware.

Each cloud federation level presents different challenges and operates at a different layer of the IT stack. It then requires the use of different approaches and technologies. Taken together, the solutions to the challenges faced at each of these levels constitute a reference model for a cloud federation.

11.3.2.1 Conceptual level

The conceptual level addresses the challenges in presenting a cloud federation as a favorable solution with respect to the use of services leased by single cloud providers. In this level it is important to clearly identify the advantages for either service providers or service consumers in joining a federation and to delineate the new opportunities that a federated environment creates with respect to the single-provider solution. Elements of concern at this level are:

- Motivations for cloud providers to join a federation

[10] http://www.samj.net/2009/06/interCloud-is-global-Cloud-of-Clouds.html.

[11] CloudSwitch is a Cloud company that focuses on delivering an enterprise gateway to several cloud computing systems; www.Cloudswitch.com/page/Cloud-federation-and-the-interCloud (retrieved May 10, 2011).

- Motivations for service consumers to leverage a federation
- Advantages for providers in leasing their services to other providers
- Obligations of providers once they have joined the federation
- Trust agreements between providers
- Transparency versus consumers

Among these aspects, the most relevant are the motivations of both service providers and consumers in joining a federation.

From the perspective of cloud service providers, being part of federation is favorable if it helps increase their revenue and if it provides new opportunities to increase their business. Moreover, the option of joining a federation can also be considered convenient if it helps sustain the QoS ensured to customers in periods of peak load, which put extreme demand on the infrastructure of the single provider. More precisely, it is possible to identify functional and nonfunctional requirements that cloud service providers have behind these motivations. The functional requirements include:

- *Supplying low-latency access to customers, regardless of their location.* It is very unlikely that single cloud providers have a capillary distribution of their datacenters. Therefore, services that require low latency might provide poor performance because of unfortunate geo-location. Within this scenario the federation might help the single providers deliver the same service and meet the expected QoS.

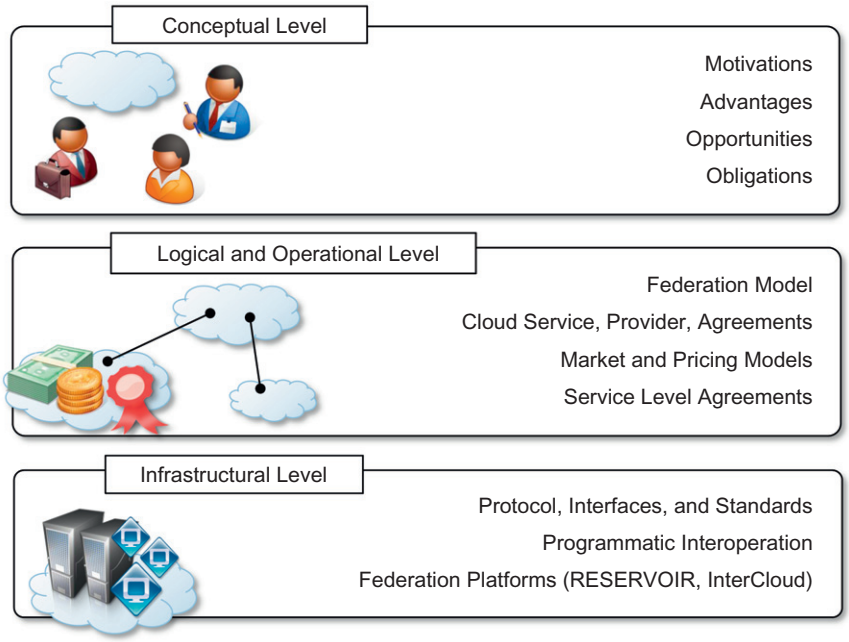

FIGURE 11.7

Cloud federation reference stack.

- *Handling bursts in demand.* Even though cloud computing gives the illusion of infinite capacity and continuous availability, service providers rely on a finite IT infrastructure that eventually will be fully utilized. A natural solution to this problem is increasing the infrastructure by adding more capacity. For example, to keep up with the increasing demand for storage and computation, Google has increased its number of servers from 8,000 to more than 450,000 in five years and moved from four server farms to more than 60 datacenters; Facebook has recently doubled its datacenter capacity. Such huge provisions are affordable for large IT companies that can make appropriate forecasts about increasing demand. Irregular demand can be better addressed by renting capacity from other providers, since not every cloud provider is in the position of being an IT giant. Cloud federation facilitates such activity by providing a context within which the lease of resources or services is encouraged.
- *Scaling existing applications and services beyond the capabilities of the owned infrastructure.* The need for additional capacity can also originate from the growth in scale of existing applications that are temporarily hosted and do not constitute a vital part of the service provider core business. Again, the opportunities for leasing additional services from a federated provider can constitute a potential advantage for a cloud federation.
- *Make revenue from unused capacity.* To provide the illusion of continuous availability and infinite capacity, cloud service providers generally own large computing systems, which generate costs in terms of maintenance and power consumption despite their real use. Energy-efficient computing solutions can help reduce costs and the impact of IT on the environment. A different opportunity is given by the cloud federation, whereby providers can lease their services to other providers for a limited period of time and thus make revenue, even without direct customers.

The motivations for joining a cloud federation also include nonfunctional requirements. The most relevant are the following:

- *Meeting compulsory regulations about the location of data.* Geo-location might become an in issue that limits a provider's capability to serve consumers. In this particular scenario it is not lack of capacity on the provider side that is the reason for leveraging the federation but, instead, the opportunity for identifying a provider that is in a position to deliver the service to the customer because of the location of its datacenter. Geo-location of data becomes an important matter when cloud services deal with confidential data that require specific levels of secrecy. Different countries have different regulations with respect to, for instance, the level of access to confidential data that government institutions may have.
- *Containing transient spikes in operational costs.* Operational costs can experience temporary spikes when there is a sudden change in electrical power due to natural disasters. This situation makes it inconvenient to fully exploit a given datacenter and provides an opportunity for leveraging federation resources to deliver services a cheaper price.
- *Disaster recovery.* Natural disasters happen, and if datacenters are co-located a disaster can put an entire datacenter or more out of service for an undefined period of time. In this scenario agreements between providers to handle disaster conditions are more likely to be settled in a federated context than in a competitive market.

For all these cases, cloud federation helps provide not only conceptual solutions but also practical means to realize these goals.

Cloud federation is an overlay that mostly benefits cloud service providers and that is supposed to be transparent to service consumers. Besides the indirect benefits to end users, there are indeed some potential direct benefits originating specifically from the concept of federation. Indirect benefits are mostly related to the QoS perceived by the end users. Real QoS is possible by enforcing admission control, which ensures that if a request is accepted, it will be served in compliance with the QoS profile defined in the SLA signed with the customer. Currently, the major cloud service providers engage QoS agreements that are mostly based on availability rather than other quality factors. For instance, in an IaaS scenario the published hardware features of a VM instance might not mirror its real performance. Since there is no SLA enforcement of such features, the provider will always try to serve requests, even when risking delivery of poor performance. In a federated scenario, requests may be served by leveraging other providers, thus ensuring that the expected performance profile is met. Therefore, as indirect benefit for users, cloud federation can help increase the overall QoS the user experiences when requesting a service. Direct benefits instead constitute something that is an advantage to end users, and they are perceivable because of the existence of federated clouds.

Cloud providers that offer different services can support each other since they are not competitors. A good example can be taken from the cooperation between the airline and accommodation market segments. Airline companies provide you with selected options for accommodation to be paired with a flight booking. This is generally the result of an agreement between the hotel and airline companies that might support each other, thus providing better service to the customer. Since companies operating in the two sectors are not competing with each other, they can both gain advantage if they provide customers with a complete solution. It is possible to replicate this type of collaboration in a federated cloud computing environment. For instance, providers that reside in different market segments (IaaS, PaaS, SaaS) might advertise each other to provide better service to the user. Enterprises that have legacy systems will be primarily looking at IaaS solutions to deploy and scale their systems. IaaS vendors can complement their offerings with advantageous access to some PaaS services by selecting those that might be complementary, of interest to the user, and offered by federated providers.

In the future, Amazon AWS might provide discounted access to AppEngine or simply provide a better interaction with the services exposed by Google in terms of data transfer, network connection, and bandwidth. How does this help the customer? The same company that is already hosting its Web application on Amazon EC2 might in the future want to integrate new features and develop them with a scalable technology. Due to performance advantages gained in leveraging AppEngine from an EC2 deployment, this could be the solution of choice. This is more likely to be possible within the context of a cloud federation. Moreover, federated clouds can provide better service to users, even when they reside in the same market segment but provide different services. For instance, in an IaaS scenario a specific provider might not be able to serve VM templates for hosting a specific operating system, but it can suggest or point the customer to another provider that's able to supply that capability. This scenario is applicable if the two providers have mutual agreements that are facilitated by belonging to a federation.

Being part of a federation also implies providers' obligations to avoid parasitic behaviors. For instance, each provider is expected to be an active member of the federation by contributing its

resources. This makes an organization such as the federation dependable and increases the trust that each provider puts in it. Obligations, such as always making available a fraction of resources and services to the federation, might be considered disadvantages, but they may also constitute potential benefits. For instance, large companies such as Google are charged for energy usage according to the peak requests rather than detailed actual usage over a month [135]. This means that if in one month a datacenter reaches 90% of peak capacity and on average works at 60%, it will pay power bills for the cost of operating at 90% capacity for the entire month. This has led companies to put a lot of effort into optimizing the utilization of datacenters. A cloud federation might be an alternative to frenetic optimization, since it might make internal resources available for usage by other member of the federation. The revenue obtained from leasing these resources is an opportunity to compensate the energy costs for peak request.

All these aspects provide a rationale for the existence of federated clouds. Obstacles at the conceptual level are the implications for security and trust. For instance, in a federated context a provider might offload a portion of the service consumers' requests to another provider with which it has agreements. This is done transparently to the user, who might not desire such behavior. These are challenges that have to be properly addressed in order to make the concept of cloud federation a viable way to efficiently exploit cloud computing as a technology.

11.3.2.2 Logical and operational level

The logical and operational level of a federated cloud identifies and addresses the challenges in devising a framework that enables the aggregation of providers that belong to different administrative domains within a context of a single overlay infrastructure, which is the cloud federation. At this level, policies and rules for interoperation are defined. Moreover, this is the layer at which decisions are made as to how and when to lease a service to—or to leverage a service from—another provider. The logical component defines a context in which agreements among providers are settled and services are negotiated, whereas the operational component characterizes and shapes the dynamic behavior of the federation as a result of the single providers' choices. This is the level where MOCC is implemented and realized.

It is important at this level to address the following challenges:

- How should a federation be represented?
- How should we model and represent a cloud service, a cloud provider, or an agreement?
- How should we define the rules and policies that allow providers to join a federation?
- What are the mechanisms in place for settling agreements among providers?
- What are providers responsibilities with respect to each other?
- When should providers and consumers take advantage of the federation?
- Which kinds of services are more likely to be leased or bought?
- How should we price resources that are leased, and which fraction of resources should we lease?

The logical and operational level provides opportunities for both academia and industry. Whereas the need for a federation—or more generally, some sort of interoperation—has now been assessed, there is no common and clear guideline for defining a model for cloud federation and addressing these challenges. Indeed, several initiatives are developing. On the research side, several works are investigating organizational models for cloud interoperability and studying economical models for describing the behavior of cloud service providers and service consumers in a federated

environment. Within industry, the major IT giants are working on promoting standards and drafting proposals for interoperable clouds.

Particular attention on this level has been put on the necessity for SLAs and their definition [150]. The need for SLAs is an accepted fact in both academy and industry, since SLAs define more clearly what is leased or bought between different providers. Moreover, SLAs allow us to assess whether the services traded are delivered according to the expected quality profile. It is then possible to specify policies that regulate the transactions among providers and establish penalties in case of degraded service delivery. This is particularly important because it increases the level of trust that each party puts in cloud federation.

SLAs have been in use since 1980 and originated in the telecommunications domain [152] to define the QoS attached to a contract between a consumer and a network provider. From there on, they have been used in several fields, including Web services, grid computing, and cloud computing. The specific nature of SLA varies from domain to domain, but it can be generally defined as "an explicit statement of expectations and obligations that exist in a business relationship between two organizations: the service provider and the service consumer.[12]" SLAs define the provider's performance delivery ability, the consumer's performance profile, and the means to monitor and measure the delivered performance. An implementation of an SLA should specify [153]:

- *Purpose*. Objectives to achieve by using a SLA.
- *Restrictions*. Necessary steps or actions that need to be taken to ensure that the requested level of service is delivered.
- *Validity period*. Period of time during which the SLA is valid.
- *Scope*. Services that will be delivered to the consumer and services that are outside the SLA.
- *Parties*. Any involved organizations or individual and their roles (e.g. provider, consumer).
- *Service-level objectives (SLOs)*. Levels of services on which both parties agree. These are expressed by means of service-level indicators such as availability, performance, and reliability.
- *Penalties*. The penalties that will occur if the delivered service does not achieve the defined SLOs.
- *Optional services*. Services that are not mandatory but might be required.
- *Administration*. Processes that are used to guarantee that SLOs are achieved and the related organization responsibilities for controlling these processes.

Parties that are willing to operate under an SLA engage a multistep process that generally involves the discovery or creation of the SLA, its operational phase, and its termination once its validity is over or there has been a violation of the contract [154]. A more articulated process has been detailed by the Sun Internet Data Center Group, which includes six steps for the SLA life cycle:

1. Discover service provider.
2. Define SLA.
3. Establish agreement.
4. Monitor SLA violation.
5. Terminate SLA.
6. Enforce penalties for SLA violation.

[12]Dinesh V, Supporting Service Level Agreements on IP Networks, Proceedings of IEEE/IFIP Network Operations and Management Symposium, 92(9):1382–1388, NY, USA, 2004.

Currently, a very rudimentary level of SLA management is present and the interoperability among different cloud providers is mostly characterized by *ad hoc* aggregation. The proposed process constitutes a reference model that still needs to be realized rather than a common implementation. Considerable work has been done in the academic domain regarding each of the steps defined here [150], but within industry, SLAs are still unilateral arrangements that are imposed by service providers and that the user can only accept rather than negotiate.

11.3.2.3 Infrastructural level

The infrastructural level addresses the technical challenges involved in enabling heterogeneous cloud computing systems to interoperate seamlessly. It deals with the technology barriers that keep separate cloud computing systems belonging to different administrative domains. By having standardized protocols and interfaces, these barriers can be overcome. In other words, this level for the federation is what the TCP/IP stack is for the Internet: a model and a reference implementation of the technologies enabling the interoperation of systems.

The infrastructural level lays its foundations in the IaaS and PaaS layers of the Cloud Computing Reference Model (discussed in Section 4.2). Services for interoperation and interface may also find implementation at the SaaS level, especially for the realization of negotiations and of federated clouds. At this level it is important to address the following issues:

- What kind of standards should be used?
- How should design interfaces and protocols be designed for interoperation?
- Which are the technologies to use for interoperation?
- How can we realize a software system, design platform components, and services enabling interoperability?

Interoperation and composition among different cloud computing vendors is possible only by means of open standards and interfaces. Moreover, interfaces and protocols change considerably at each layer of the Cloud Computing Reference Model. As the more mature layer, the IaaS layer has evolved more in this sense. Almost every IaaS provider exposes Web interfaces for packaging virtual machine templates, launching, monitoring, and terminating virtual instances. Even though not standardized, these interfaces leverage the Web services and are quite similar to each other. The use of a common technology simplifies the interoperation among vendors, since a minimum amount of code is required to enable such interoperation. These APIs allow for defining an abstraction layer that uniformly accesses the services of several IaaS vendors. There are already tools—both open-source and commercial—and specifications that provide interoperability by implementing such a layer.

To fully support the vision of a cloud federation, more sophisticated capabilities need to be implemented. For instance, the possibility of dynamically moving virtual machine instances among different providers is essential to supporting dynamic load balancing among different IaaS vendors. In this direction, the *Open Virtualization Format (OVF)* [51] aims to be a solution for this problem and may eventually be successful, since it has been endorsed by several cloud computing vendors. If we consider the PaaS layer, the interoperations become even harder since each cloud computing vendor provides its own runtime environment, which might differ from others in terms of implementation language, abstractions used to develop applications, and purpose. Currently, there is no

sign of interoperation at this level and no proposed standards simplifying the interoperation among vendors. Regarding the SaaS layer, the variety of services offered makes the operation less crucial.

An interesting case for interoperability at the SaaS layer is provided by online office automation solutions such as Google Documents, Zoho Office, and others; several of them provide the capability to export and import documents to and from different formats, thus simplifying the exchange of data. Alternatively, composition seems to be a more attractive opportunity; different SaaS services can be glued together to provide users with more complex applications. Composition is also important in considering interoperability across cloud computing platforms operating at different layers. Even in this case it is important to note that, currently, cloud computing providers operating at one layer often implement on their own infrastructure any lower layer that is required to provide the service to the end user, and they are not willing to open their stack of technologies to support interoperation.

The vision proposed by a federated environment of cloud service vendors still poses a lot of challenges at each level, especially the logical and infrastructural level, where appropriate system organizations need to be designed and effective technologies need to be deployed. Considerable research effort has been carried out on the logical and operational level, and initial implementations and drafts of interoperable technologies are now developed especially for the IaaS market segment.

11.3.3 Aspects of interest

Several aspects contribute to the successful realization of a cloud federation. Besides motivation and technical enablers, other elements should be considered. In particular, standards for interoperability, security, and legal issues have to been taken into consideration while defining a platform for interoperability among cloud vendors.

11.3.3.1 Standards

Standards play a fundamental role in building a federation. Their main role is to organize a platform for interoperation that goes beyond *ad hoc* aggregations and private settlements between providers. Standardized interfaces and protocols facilitate the realization of an open organization where providers can easily join. The advantages are primarily technical; standards facilitate the development of software and services that interconnect systems. Furthermore, they help in defining clear paths for new providers to join, thus contributing to the realization of open systems.

Interoperation between vendors has always been an element of concern for enterprises and one of the reasons that initially prevented them from fully embracing the cloud computing model. More specifically, the absence of common standards for developing applications and systems in a portable way initially developed the fear of *vendor lock-in*. Applications and systems developed to be deployed on a specific cloud computing vendor's infrastructure could not be easily moved to another vendor. IaaS solutions still leverage proprietary formats for virtual machine instances and templates. This prevents instances from being moved from one vendor's platform to another. The technical barriers are even more considerable in the case of PaaS solutions, where even the development technology might differ. These technical barriers led to the development of solutions that tried to overcome these obstacles, at least for a limited number of providers. *Rightscale*,[13] for instance, is a

[13] www.rightscale.com.

solution that provides customers with a development platform that can be transparently deployed over different IaaS vendors. On the PaaS segment, *Aneka* provides middleware that can harness heterogeneous hardware and support cross-platform deployment across IaaS vendors. The *jClouds* project[14] defines a set of libraries and components that allows uniform use of different IaaS providers to develop applications on the cloud in Java.

These initiatives constitute technical improvements that help system designers and developers develop systems that are less subject to the vendor lock-in problem. They are far from providing the appropriate support that is required for building a federation, which can only be achieved by means of standards, protocols, and formats designed for interoperation. The efforts to create standards are carried out within the contexts of open organizations and consortia of major industry partners. Some minor efforts can be found in the academic world.

Open cloud manifesto

The Open Cloud Manifesto[15] constitutes the first step toward the realization of a cloud interoperability platform. The manifesto was drafted in 2009 as a result of the coordinated activity of different cloud vendors and currently lists more than 400 cloud computing services providers that support the vision it embodies. More than proposing standards, the manifesto is a declaration of intent, endorsed by commercial players in the field of cloud computing, to realize an interoperable and open cloud computing platform. The document is intended to be of guidance to the CIOs, governments, IT users, and business leaders who want to use cloud computing and to establish a set of core principles for cloud computing.

The document enumerates the advantages of cloud computing and discusses the challenges and barriers to its adoption. It introduces the goals of an open cloud platform, which can be summarized as follows:

- *Choice*. With the use of open technology it will be possible for IT consumers to select the best provider, architecture, or usage model as the business environment changes. Furthermore, the use of open technologies simplifies the integration of cloud computing solutions provided by different vendors or with the existing infrastructure, thus facilitating its adoption.
- *Flexibility*. The change between one provider and another becomes easier if the different vendors do not use a closed proprietary technology, which implies considerable conversion costs.
- *Speed* and *Agility*. As noted before, open technologies facilitate the integration of cloud computing solutions within existing software systems, thus realizing the promise of scaling demand with speed and agility.
- *Skills*. Open technologies simplify the learning process and contribute to developing a common knowledge that can be used to design, develop, and deploy systems across multiple providers. This simplifies the chances for organizations to find someone with the appropriate skills for their needs.

The manifesto ends with a set of recommendations for cloud computing vendors to pursue an open collaboration and an appropriate use of standards. Existing standards rather than proprietary

[14] http://code.google.com/p/jClouds/.
[15] www.opencloudmanifesto.org.

solutions are encouraged to be leveraged and, where appropriate, new standards should be drafted as a result of a community-based effort.

By evidencing the advantages of an open cloud platform, the manifesto lays the conceptual foundations for a cloud federation scenario. In fact, the use of open technologies will create a more flexible environment where IT consumers will more comfortably choose cloud computing technologies, without feeling the menace of vendor lock-in. The concept of cloud federation constitutes an evolution of this initial vision, which implies a more structured and explicit collaboration.

Distributed management task force

The *Distributed Management Task Force (DMTF)* is an organization involving more than 4,000 active members, 44 countries, and nearly 200 organizations. It is the industry organization leading the development, adoption, and promotion of interoperable management standards and initiatives. With specific reference to cloud computing, the DMTF introduced the *Open Virtualization Format (OVF)* and supported several initiatives for interoperable cloud technologies, such as the *Open Cloud Standards Incubator*, the *Cloud Management Working Group (CMWG)*, and the *Cloud Audit Data Federation Working Group (CADFWG)*.

The *Open Virtualization Format (OVF)* [51] is a vendor-independent format for packaging standards designed to facilitate the portability and deployment of virtual appliances across different virtualization platforms. OVF can be used by independent software vendors (ISVs) to package and securely distribute applications, and system images can be imported and deployed on multiple platforms, thus enabling cross-platform portability. The specification is the result of the collaborative effort of Dell, HP, IBM, Microsoft, VMWare, and XenSource in defining a platform-independent virtualization format for packaging software appliances. An initial draft was submitted to the DMTF in 2008, and currently the DMTF has released version 1.1.0 of the specification, which has also been ratified as American National Standards Institute (ANSI) standard INCITS 469-2010. The main purpose of the specification is to provide a platform-neutral format for distributing packaged software systems. Therefore, the key features of the format are the following:

- *Optimized for distribution*. The format supports verification and integrity checks based on industry-standard public key infrastructure and provides a basic license management scheme.
- *Optimized for a simple, automated user experience*. Supports validation of the entire package as well as each virtual machine or metadata component.
- *Supports both single VM and multi-VM configurations*. Provides the ability to package either single-image applications or complex multitier systems.
- *Portable vendor- and platform-independent VM packaging*. OVF is platform-neutral by design but also allows the embedding of platform-specific enhancements.
- *Extensible*. Even though the current specification is already capable of capturing all the required data for packaging a virtual appliance, it provides ways to extend and define new features for future needs.
- *Localizable*. The format allows embedding user-visible descriptions in multiple locales, thus being immediately ready to be processed over a wide range of systems and markets.
- *Open standard*. The format has arisen from the collaboration of key vendors in the industry and its evolution will be driven in an accepted industry forum in order to guarantee its future diffusion and use.

From a technical point of view, the OVF defines a transport mechanism for virtual machine templates. One single OVF package can contain one or more virtual machine images, and once deployed to the host system, it adds a self-containing, self-consistent software solution for achieving a particular goal. Examples are a Linux, Apache, MySQL, and PHP (LAMP) stack or any other combination of components divided into one or more virtual images. OVF's focus is to deliver a packaged, portable, ready-to-use, and verifiable software appliance. To achieve this goal, it allows the packaging of multiple virtual instances together with all the necessary descriptors for their deployment on virtual hardware. Currently, it is quite common to implement a software application or system by using a multitier architecture, which may spread across several computing nodes the components of the system, each of them requiring different operating systems and hardware characteristics. By supporting multiple virtual images within a single package, ISVs can package and certify entire systems as a whole and distribute them as a single component.

Portability is a fundamental property of the OVF. To fully address the potential issues arising in deploying a software appliance over a wide range of virtual hardware, different levels of portability are identified:

- *Level 1*. The packaged appliance runs on a particular product and/or CPU architecture and/or virtual hardware selection.
- *Level 2*. The packaged appliance runs on a specific family of virtual hardware.
- *Level 3*. The packaged appliance runs on multiple families of virtual hardware.

What determines the level of portability of an OVF package is the information that is packaged in the appliance and its specific requirements. Level 1 portability is mostly due to virtual hardware-specific information preventing the OVF's deployment even on a similar family of hypervisors. This information might be, for instance, a virtual machine suspended state or snapshot. Level 2 portability is acceptable within a single administrative domain where the decisions about the virtualization technology to adopt are most likely to be centralized. Level 3 portability is the one expected and desired by ISVs that want to package appliances capable of deployment on any virtual platform. In the context of a cloud federation, this is the expected level of portability for a seamless collaboration among different vendors. The OVF specification has been accepted quite positively since its inception, and several open-source initiatives and commercial products can import software appliances distributed as OVF packages.

The *Open Cloud Standards Incubator* is another important initiative that is driven by the DMTF to promote interoperability among cloud technologies. The focus of the incubator was the standardization of the interactions between cloud computing environments by developing cloud management use cases, architectures, and defining interactions. The activity of the incubator ended in July 2010 and produced a collection of white papers ([137, [138], and [139]) that are of guidance for developing interoperable cloud systems. These white papers were the starting point of the *Cloud Management Working Group (CMWG)*, the aim of which is to deliver a set of prescriptive specifications that confer architectural semantics and define implementation details to achieve interoperable management of clouds between service requestors/developers and providers. Another relevant initiative of the DMTF is the *Cloud Audit Data Federation (CADF)* working group, which will focus its interests on defining interoperable APIs for producing and sharing specific audit events, log information, and reports on a per-tenant basis, thus simplifying monitoring heterogeneous cloud computing environments.

The DMTF has made a significant contribution toward the realization of an interoperable cloud environment. The only concrete activity has been the release of the OVF specification. However, activities of the working groups and associated documents pave the way for realization of open cloud federation.

Open cloud computing interface

The *Open Cloud Computing Interface (OCCI)*[16] is an open organization that comprises a set of specifications driven by the community and delivered through the Open Grid Forum. These specifications define protocols and APIs for several kinds of management tasks. Initially conceived to create a remote management API for IaaS-type services, the OCCI has evolved into a wider set of APIs focusing on integration, portability, and interoperability. The current set of specifications covers all three market segments of cloud computing: IaaS, PaaS, and SaaS. OCCI has currently delivered three documents that define:

- The formal definition of the OCCI core model
- The definition of the OCCI infrastructure extensions for the IaaS domain
- The OCCI rendering model, which defines how to interact with the OCCI core model through REST over HTTP

Besides these, the OCCI is also working on other specifications that cover the following topics: billing, monitoring, advanced reservation, negotiation, and agreement features.

Figure 11.8 provides a reference scenario for the Open Cloud Computing Interface, which defines a single access point to the resources managed by a given cloud resource provider.

The most relevant specification for defining an interoperable platform is the OCCI Core Model (*GFD.183−OCCI Core*). This document defines the boundary protocol and API that acts as a service front-end to the provider's internal management framework. This interface was designed to serve both service consumers and other systems. This specification is not focused on defining any particular domain, but its main scope is to provide a built-in type classification system allowing for safe extensions toward domain-specific usage. Therefore, it provides a reference model for building a type system that is as flexible as possible and that supports extensibility. This allows clients to dynamically discover the resource model of any provider supporting this specification. *Resources*, which are the fundamental element of the model, can be of any type and vary according to the context in which the resource provider operates: a virtual machine or a template in case of IaaS providers, a job or a task in the case of a job management system, and so on. These resources are connected through *links*, which allow the clients to see how they are connected together. Resources and links are *entities* that can be used to define a specific type. Besides the entities, a type (also called a *kind*) is defined by *actions*, which represent the operations that can be done on instances of that type. To support flexibility and extensions, *mixins* are used to aggregate preexisting types and augment their capabilities.

This reference model is then specialized by the OCCI Infrastructure document (*GFD.184−OCCI Infrastructure*) that defines how an OCCI implementation can model and implement an IaaS API according to the previous reference model. Common resources in this scenario are *Compute*, *Storage*, and *Network*. Links might be of type *NetworkInterface* and *StorageLink*.

[16] http://occi-wg.org/.

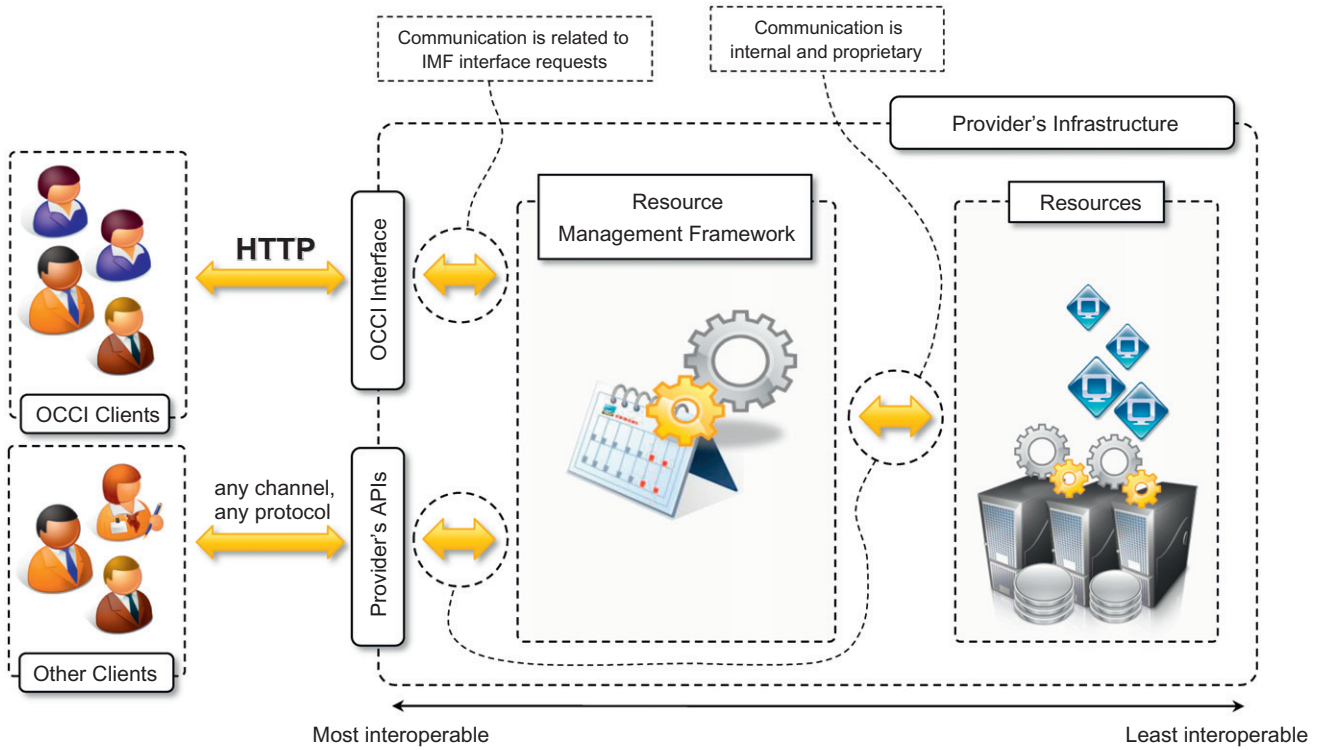

FIGURE 11.8

OCCI reference scenario.

These components can be used to represent any services offered by any IaaS provider and can describe their behavior and properties.

A practical implementation of an OCCI model is then made accessible through RESTful interfaces over HTTP. The OCCI rendering model specification (*GFD.185−OCCI HTTP Rendering*[17]) defines how the model can be transmitted and serialized through HTTP. Therefore, a cloud resource provider has to implement the following two steps in order to be OCCI compliant:

1. Define the services and resources it offers, according to the OCCI core model.
2. Provide a RESTful interface that allows clients to discover the set of resources it exposes according to the OCCI HTTP rendering model.

If the cloud vendor is an IaaS provider, the OCCI Infrastructure document constitutes a starting point from which the vendor develops its own representation.

Currently, the Open Cloud Computing Interface provides linking with several technologies for building cloud computing services, principally within the IaaS market segment. Various open-source initiatives (jCloud, libvirt, OpenNebula, and OpenStack), research projects (RESERVOIR, Big Grid, Morfeo Claudia, Emotive), and consortia (SLA@SOI) are offering OCCI interfaces to their services. Initially developed as a joint effort of Sun Microsystems, Rabbit MQ, and the Univesidad Complutense de Madrid, now OCCI involves more than 250 organizations in both academia and industry, such as Rackspace, Oracle, GoGrid, and Flexiscale. Because of such strong support, OCCI is a promising step toward the definition of cloud interoperability standards for a cloud federation scenario.

Cloud data management interface

The *Cloud Data Management Interface (CDMI)* [136] is a specification for a functional interface that applications will use to create, retrieve, update, and delete data elements from the cloud. This interface also provides facilities for discovering the properties of a given cloud storage offering. CDMI has been proposed by the *Cloud Storage Technical Working Group* of the *Storage Network Industry Association (SNIA)*, an association promoting standards in the management of IT information with a particular focus on data storage. SNIA has also produced a reference implementation of the CDMI, thus facilitating the process of quickly producing a working standard by means of feedback from the community.

The specification introduces and defines the concept of cloud storage as a "delivery of virtualized storage on demand," also known as *Data storage-as-a-Service (DaaS)*. The main concept of DaaS is to abstract data storage behind a set of interfaces and to make it available on demand. This definition encloses a considerably wide range of storage architectures. Figure 11.9 provides the overall context in which cloud storage interfaces will operate. A cloud data management service provides a CDMI made available to clients through RESTful interfaces. Such an interface provides access to information, data, and storage services that can be leveraged to access storage clouds. These can eventually rely on several different technologies to implement data and storage services.

[17]The document is reported to be published but was not available at the time of writing.

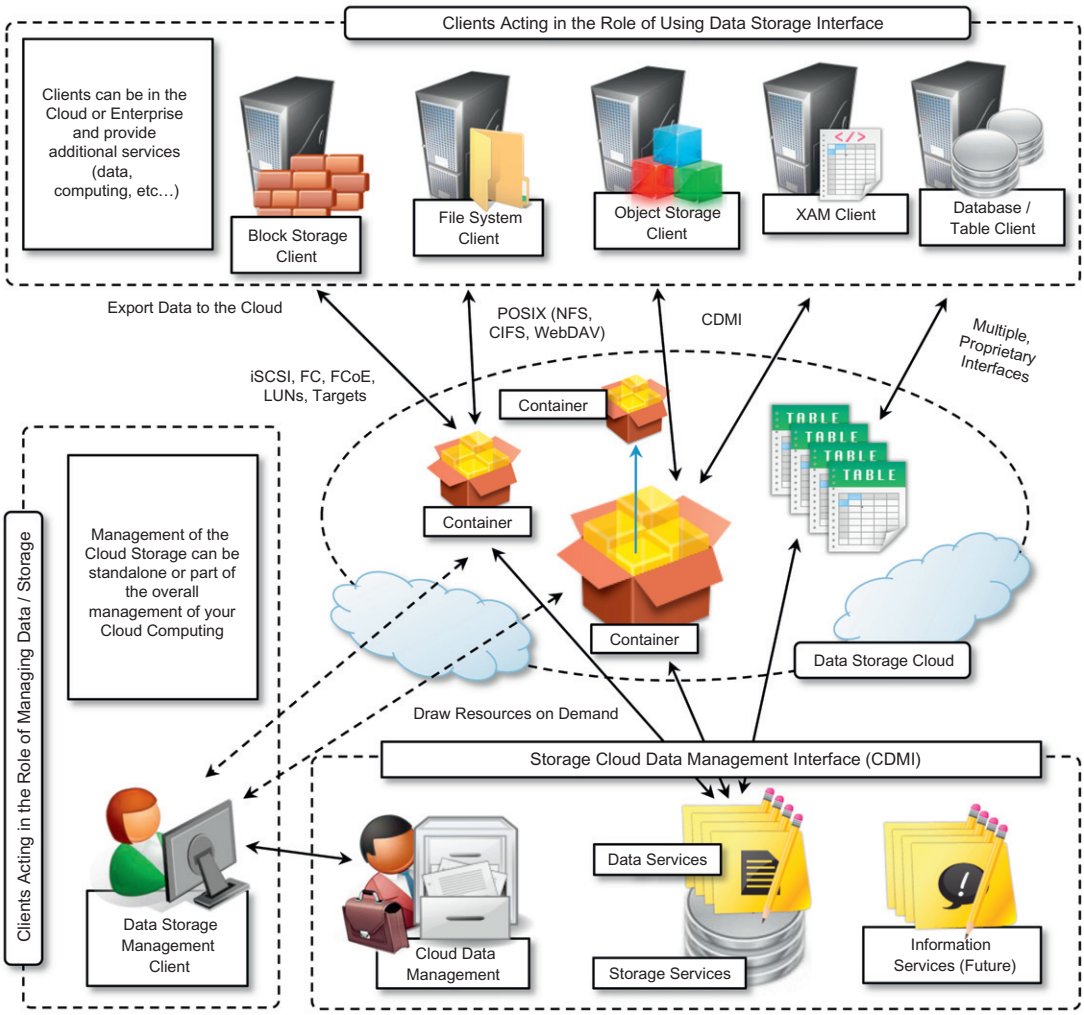

FIGURE 11.9

Cloud storage reference model.

The interface exposes an object model that allows clients to manipulate and discover data components. The object model contains the following components:

- *Data objects*. These are the fundamental storage component in CDMI and are analogous to file in a file system. Data objects can have a set of well-defined single-value fields. In addition, they can support metadata used to describe the object, which can be provided by either the storage system or the client.
- *Container objects*. Container objects are the fundamental abstraction used to group stored data. A container may have zero or more child objects and a set of well-defined fields. As happens

for data objects, containers support metadata as well. Containers support nesting, and a child container inherits all the data system metadata from its parent.

- *Domain objects*. Domain objects are quite similar to container objects and they are used to represent administrative ownership stored within a CDMI storage system. As containers, they support nesting and facilitate the flow of information upward since an aggregate view of data is useful for administrative purposes.
- *Queue objects*. Queues are a special class of containers that are used to provide first-in, first-out (FIFO) access when storing and retrieving data. Queues are useful to support writer-reader and producer-consumer patterns for storage data management. A queue always has a parent object from which it inherits the system metadata.
- *Capability objects.* Capability objects are a special class of container object that allow a CDMI client to discover what subset of the CDMI standard is implemented by a CDMI provider. Capabilities are descriptors of the set of interactions that the system is capable of performing against the URI to which they are attached. Each entity defined in the object model is expected to have a field that represents the URI from which the capabilities for that object can be retrieved. Every CDMI-compliant interface must be able to list the capabilities for each given object, but support for all the capabilities listed in the standard is optional.

Using the simple operation defined by REST, clients can discover and manipulate these objects by creating, retrieving, updating, and deleting objects (*CRUD: Create, Retrieve, Update, and Delete*). The set of operations that are supported is defined by the capabilities attached to each entity. Besides the fundamental operations allowed by REST, the CDMI also provides support for snapshots, serialization and deserialization, logging, and interoperation compared to other protocols and standards.

The CDMI was initially proposed in 2010 and collected consensus from several other bodies involved in the standardization of cloud computing technologies and was included in roadmaps and studies. Currently, SNIA is moving toward transforming CDMI into a *de jure* standard by interoperating with the major standardization organizations, such as ISO/IEC and INCITS.[18]

Cloud security alliance

The *Cloud Security Alliance (CSA)* is nonprofit organization with the mission of promoting the use of best practices for providing security assurance in cloud computing and education on the use of cloud computing to help secure all other forms of computing. Rather than acting as a standardizing body, CSA offers a context in which to discuss security practices and provide guidance for developing reliable and secure cloud computing systems.

The most relevant initiative of the CSA has been the *Cloud Controls Matrix (CCM)*. The matrix is specifically designed to provide fundamental security principles for guiding cloud vendors and for assisting prospective cloud service consumers in assessing the overall risks implied in leveraging a cloud service provider. This document was prepared by taking into account the most important security standards, regulations, and control frameworks, such as ISO 27001/27002, ISACA COBIT, PCI, and NIST. The CCM strengthens existing information security control environments

[18] *InterNational Committee for Information Technology Standards (INCITS)* is the primary U.S. focus of standardization in the field of information and telecommunication technologies (ICT).

within a cloud computing context and provides a way to align the security practices that should be adopted in the cloud with those that already exist in other domains.

The CCM's relevance in a cloud federation scenario is quite evident. It provides a standardized way to assess the security measures put in place by each cloud service provider and helps define a minimum-security profile within a cloud federated scenario, thus increasing the trust in the concept of federation.

Other initiatives

Besides the major standardization efforts, many initiatives have obtained minor popularity, are endorsed by specific organizations, or are related to the use of cloud computing in a very specific context. Among the most relevant are these:

- *National Institute of Standards and Technologies (NIST).* As discussed in Chapter 4, NIST proposed a working definition of cloud computing that is now widely accepted. Another important initiative is the *Standards Acceleration to Jumpstart Adoption of Cloud Computing (SAJACC).* This initiative's activities are mostly related to the assessment of existing standards in cloud computing, to actively contributing to the creation of open standards, and to the identification of gaps within existing standards.
- *Cloud Computing Interoperability Forum (CCIF).* This industry forum was initially supported by several players in the field of cloud computing and was formed to enable a global cloud computing ecosystem whereby organizations can seamlessly operate together to foster wider adoption of the technology. The activity of the forum is limited to one single event in 2009 and led to the proposal of the *Unified Cloud Interface (UCI),*[19] which is an attempt to provide a unified interface to the various APIs exposed by different vendors. Functional implementations exist for Amazon EC2 and Enomaly ECP.
- *Open Cloud Consortium (OCC).* This not-for-profit organization manages cloud computing infrastructures to support scientific research. It also develops reference implementations, benchmarks, and standards for cloud technologies.
- *Global Inter-Cloud Technology Forum (GICTF).* The GICTF is a Japanese organization whose aim is to bring together the knowledge developed by industry, academia, and government and to support research and development feasibility tests on the technologies related to cloud interoperability. More precisely, the forum focuses on the standardization of the network protocols that enable interoperation among clouds and of the interfaces through which this interoperation happens.
- *Cloud Work Group.* This initiative of the Open Group is aimed at creating a common understanding between buyers and suppliers as to how enterprises of all sizes and scales can leverage cloud computing technologies. The group has established several activities to enhance business understanding, analysis, and uptake of cloud computing technologies.

Several other initiatives exist. They cover the creation of standards and specifications for areas of cloud computing, which are not strictly related to supporting the interoperation among different cloud service providers but still constitute a step toward the development of supporting

[19]Available at http://code.google.com/p/unifiedCloud/ (retrieved May 17, 2011).

technologies for a cloud federation scenario that is based on the definition of the rules, policies, and a model for a cloud federation.

11.3.3.2 Security

Cloud computing supports the development of elastically scalable systems by leveraging large computing infrastructures that eventually host applications, services, and data. In this scenario, security arrangements constitute a fundamental requirement that cannot be overlooked. Security management becomes even more complex in the case of a cloud federation, where confidential information is dynamically moved across several cloud computing vendors.

One of the first questions that should be asked is whether the cloud vendors offer enough transparency to implement the security management process required to protect your data and applications. Implementing and deploying the required security infrastructure is important to understanding the responsibilities of those who offer cloud computing services and of those who use them. Cloud computing is a new territory for security; compared to managed services providers, where the management of security infrastructure is completely handled by the service provider, in a cloud deployment security management is divided between the provider and the customer. The responsibilities of the two parties change according to the type of service offering. IaaS vendors are required to provide basic security in terms of auditing and logging of access to virtual machine instances or cloud storage. PaaS vendors are expected to offer a secure development platform and a runtime environment for applications. SaaS vendors have the major responsibilities in terms of security, since they have to build a secure computing stack (infrastructure, platform, and applications) that users customize for their needs.

Despite the novelty introduced by cloud computing, it is possible to consider existing security frameworks and practices as a starting point for securely running applications and systems. In particular, the standard *ISO/IEC 27001/27002* and the *Information Technology Infrastructure Library (ITIL)* service management framework are industry standards that provide guidance in terms of security management processes and practices. ITIL is a set of guidelines that define a security framework in terms of *policies*, *processes*, *procedures*, and *work instructions*. ITIL is not a standard, and therefore organizations and management systems cannot be certified "ITIL-compliant." Despite this fact, these guidelines constitute a reference model for implementing security measures. ISO/IEC 27001/27002 formally define the mandatory requirements for an information security management system and provide guidelines for the security controls to implement within a security system. In short, these standards help organizations ensure that the current security levels are appropriate to their needs and to implement security.

Key elements in management of security in a cloud scenario have been identified as the following [156]:

- Availability management (ITIL)
- Access control (ISO/IEC 27002, ITIL)
- Vulnerability management (ISO/IEC 27002)
- Patch management (ITIL)
- Configuration management (ITIL)
- Incident response (ISO/IEC 27002)
- System use and access monitoring (ISO/IEC 27002)

Put together, these elements can compose a reference security framework for cloud computing systems. They invest all the layers of the cloud computing reference model and are differently supported across the market segments characterizing cloud computing.

Table 11.2 Customer Responsibilities for Security Management in the Cloud [156]

Activities	IaaS	PaaS	SaaS
Availability management	Manage VM availability with fault-tolerant architecture	Manage this activity for applications deployed in the PaaS platform (the provider is responsible for runtime engine and services)	Provider responsibility
Patch and configuration management	Manage VM image hardening Harden your VMs, applications, and database using your established security hardening process Manage activities for your VMs, database, and applications using your established security management process	Manage this activity for applications deployed in the PaaS platform Test your applications for OWASP Top 10 vulnerabilities[20]	Provider responsibility
Vulnerability management Access control management	Manage OS, applications, and database vulnerabilities leveraging your established vulnerability management process Manage network and user access control to VMs, secure privileged access to management consoles, install host Intrusion Detection System (IDS), and manage host firewall policies	Manage this activity for applications deployed in the PaaS platform (the provider is responsible for their runtime engine and service) Manage developer access to provisioning Restrict access using authentication methods (user- and network-based controls) Federate identity and enable SSO if SAML[21] is supported	Provider responsibility Manage user provisioning Restrict user access using authentication methods (user- and network-based controls) Federate identity and enable SSO if SAML is supported

An overall view of the security aspects and how they are addressed by cloud vendors at three different levels (IaaS, PaaS, and SaaS) is shown in Table 11.2. A good starting point in identifying a customer's security responsibilities and planning the appropriate measures is the service contract made with the provider. This contract clearly defines the scope of action of the provider and the warranties given to the customers. Moreover, cloud providers' APIs and services help in managing security and accountability. Cloud providers offer service dashboards that could be usefully integrated into the organization's internal management processes and tools. Currently, the missing bit

[20]The *Open Web Application Security Project (OWASP)* is an open community dedicated to enabling organizations to conceive, develop, maintain, acquire, and operate applications that can be trusted. In 2010 the organization published the OWASP Top Ten Project, which lists the top 10 vulnerabilities for applications; www.owasp.org/index.php/CategoryOWASP_Top_Ten_Project.

[21]*SSO*, an acronym that stands for *single sign-on*, identifies the capability offered by a collection of services that allows users to authenticate only once rather than explicitly providing each of the services with their credentials. SAML, an acronym that stands for *Security Assertion Markup Language*, is an XML dialect that is used to provide a portable framework to define security in a federated environment.

from a security point of view is the lack of enterprise-grade access for managing the security of cloud computing systems where system administrators can have a holistic view of the security defined for cloud computing applications.

Cloud federation security

Cloud federation introduces additional issues that have to be addressed in order to provide a secure environment in which to move applications and services among a collection of federated providers. Baseline security needs to be guaranteed across all cloud vendors that are part of the federation.

An interesting aspect is represented by the management of the digital identity across diverse organizations, security domains, and application platforms. In particular, the term *federated identity management* refers to standards-based approaches for handling authentication, single sign-on (SSO), role-based access control, and session management in a federated environment [157]. This enables users to utilize services more effectively in a federated context by providing their authentication details only once to log into a network composed of several entities involved in a transaction. This capability is realized by either relying on open industry standards or openly published specifications (*Liberty Alliance Identity Federation*, *OASIS Security Assertion Markup Language*, and *WS-Federation*) such that interoperation can be achieved. No matter the specific protocol and framework, two main approaches can be considered:

- *Centralized federation model.* This is the approach taken by several identity federation standards. It distinguishes two operational roles in an SSO transaction: the identity provider and the service provider.
- *Claim-based model*. This approach addresses the problem of user authentication from a different perspective and requires users to provide claims answering who they are and what they can do in order to access content or complete a transaction.

The first model is currently used today; the second constitutes a future vision for identity management in the cloud.

Digital identity management constitutes a fundamental aspect of security management in a cloud federation. To transparently perform operations across different administrative domains, it is of mandatory importance to have a robust framework for authentication and authorization, and federated identity management addresses this issue. Our previous considerations of security contribute to design and implement a secure system comprising the cloud vendor stack and the user application; federated identity management allows us to tie together the computing stacks of different vendors and present them as a single environment to users from a security point of view.

11.3.3.3 Legal issues

Besides the technical complexities required to make cloud computing happen, legal issues related to access rights, privacy, and control are peculiar to cloud computing. This section provides an overview of this topic and discusses the potential legal implications of a cloud federation scenario.

Enterprises and end users leveraging cloud computing relegate the control of systems, applications, or personal data to third parties. This is not much different from normal outsourcing practices, which have been widely in precedence. Also, a considerable similarity to application service

providers (ASPs) can be found. If we compare cloud computing with traditional outsourcing and ASP service delivery, we can observe that [140]:

- Outsourcing arrangements generally cover an entire business or IT process of a business organization, and they are geared toward running the business for the benefit of the customer. The software generally belongs to the customer, and it is deployed and managed on the customer's equipment. Moreover, arrangements are highly negotiable and defined by long and complex contracts.
- The ASP model constitutes an early implementation of the SaaS delivery model, which does not imply any sort of scalability on demand. In this scenario the provider owns the software and the hardware, which might be located in several locations but are statically known *a priori*. Arrangements are negotiable but not as complex as those regulating outsourcing.
- Cloud computing covers multiple service models: software, infrastructure, and runtime platforms. Arrangements between consumers and providers are quite limited and mostly based on a pay-per-use model, which is metered according to appropriate units (time, bandwidth, storage size, or transactions). Economics of performance drive the activity of the provider, which may choose to locate, or relocate, applications and systems in order to minimize costs and improve efficiency. Moreover, providers may have multiple datacenters located sparsely, either to better serve a wide variety of customers or to exploit advantageous environment conditions.

What makes cloud computing different from traditional approaches is the global scale at which providers operate and deliver services. The need to serve a potentially massive number of customers makes providers look for solutions that guarantee optimal service delivery and low operational costs. Therefore, having a sparse infrastructure, normally spanning several countries, is quite common. Second, because cloud computing delivers IT services as utilities, the dynamics of service lease and consumption make the common arrangements unsuitable for commercial use. As a result, multiple service-contracting models have been introduced: licensing, service agreements, and online agreements.

This new scenario makes more complicated the legal management of existing issues and introduces new ones. For instance, a disperse geography of datacenters makes it harder to guarantee confidentiality and privacy of data, especially when data are located in different countries with different jurisdictions. There already are laws and regulations for management of sensitive information, but the heterogeneous context in which cloud computing services are delivered requires a different approach from a legal perspective. Moreover, the dynamics of contracts between service consumers and providers introduce new challenges; arrangements need to be more flexible and cannot be defined by lengthy contracts obtained by complex negotiations. As we have seen, vendor lock-in is a sensitive issue in cloud computing; what will then happen when a provider files for bankruptcy or gets acquired by another provider? Will the previous SLAs still be honored? What are the warranties for customers with respect to this situation? Furthermore, service providers might stipulate agreements between themselves, which leads to a definitely more complex management of customers' sensitive data. For instance, customers do not generally have control over where the service they are leasing operates; in a federated scenario, a service might offered by a different provider from the one the customer has stipulated in the SLA. In such a case, what are the warranties offered by the other provider in terms of confidentiality, secrecy, integrity, and data retention?

We can categorize the wide range of legal issues that are connected with cloud computing in the following three major areas: *privacy, security, and intellectual property*; *business and commercial*; and *legislation and jurisdiction*. As happens for any disruptive technology, there is always a lag between the regulations that govern the interaction within parties and the new opportunities and approaches that such technology brings. In the case of cloud computing, the law lags behind the technology innovation, even though several initiatives are now taking place.

Privacy, security, and intellectual property related issues

This category includes all the legal issues arising from the management of data, applications, and services by the cloud service provider on behalf of the customer as part of its service offering. As previously noted, cloud computing implies the relegation of control over data and applications to third parties. To provide customers with a reliable and commercially viable service, cloud computing providers are expected to implement appropriate procedures to protect the integrity of data, data secrecy, and the privacy of users. In this scenario, different pieces of legislation apply, which sometimes may also be conflicting.

The legislation concerning privacy and security of data is perhaps the most developed. This is because it does not only apply to cloud computing-based systems. Again, different countries, and different states within the same country, have different approaches and advancement stages. The United States has a considerable amount of legislation regulating privacy and security measures; these laws also apply to cloud computing. An important U.S. player is the *Federal Trade Commission (FTC)*, which is in charge of conducting investigations to assess the liability of any organization that provides financial services or that engages in a commercial activity aimed at selling products if that organization breaks security and privacy regulations. In particular, *data breach*, *protection*, and *accessibility* are objects of current legislation:

- *Data breach*, or loss of unencrypted electronically stored information, is a very sensitive matter. Avoiding data breach is important for both cloud services consumers and providers. The former have a direct loss caused by the potential misuse of sensitive information from third parties (stolen Social Security numbers, misuse of credit cards, and access to personal medical data or financial statements). The latter are damaged because of financial harm, potential lawsuits, damage to reputation, FTC investigations, and loss of customers. Therefore there is interest on both sides to embrace measures that prevent data breach. This is also supported by the legislation: Almost all 50 U.S. states require the affected person to be notified upon the occurrence of data breach. Therefore, cloud providers that have their systems compromised are required to notify the affected persons and coordinate with those who provided such data. Relevant legislation for data breach includes the *Health Insurance Portability and Accountability Act (HIPAA)* [145] and the *Health Information Technology for Economic and Clinical Health (HITECH) Act* [144].
- *Data protection*. Other interesting considerations can be made regarding the measures that should be put in place to ensure the confidentiality of personal information. The *Gramm-Leach-Bliley (GLB) Act* [141] requires that financial institutions implement appropriate procedures to protect personal information and prevent unauthorized access to data. Cloud providers managing sensitive information will then be required to (1) demonstrate that they have deployed appropriate measures for the protection of sensitive data; (2) contractually agree to prevent

unauthorized access; or (3) both. Under the GLB Act, the FTC requires that all businesses significantly involved in the provision of financial services and product have a written security plan to protect information related to customers (*Safeguards Rules* [142]). Another important protection measure is represented by the *Red Flag Rules* [143], which force financial institutions to monitor and block suspicious activities that potentially might represent evidence of identity theft. Organizations need to have documented policies describing the measures they take to prevent identity theft.

- *Data access*. Legal issues also arise when it comes to determining who, besides their respective owners, can have access to sensitive data. The U.S. legislation has a considerably aggressive approach with respect to this issue. The USA PATRIOT Act [146] confers the right to any organization operating on behalf of the U.S. government to gain access to personal financial information and student information stored in electronic systems without any suspicion of wrongdoing of the person whose information it seeks. The act requires the organization to prevent a government's certification that the information accessed can be relevant to an ongoing criminal investigation. The institution holding the financial information might be prevented from informing their clients about governmental access to their data. The power conferred by this act puts a cloud provider in the difficult position of being required to provide sensitive information without giving notice to the respective user.

Other countries and interstate organizations have different approaches. The European Union (EU) in 1995 passed the *European Union Directive on the Protection of Individuals with Regard to the Processing of Personal Data and Movement of Such Data Privacy Directive*. The directive states that countries belonging to the EU must pass a data protection law covering both government and private entities that process business and consumer data. Furthermore, the directive mandates that any geography to which EU personal data are sent must have a level of data protection as measured by EU standards. This directly impacts on the possibility for many cloud providers entering and operating lawfully in the EU. Therefore, the possibility of conducting business in the European Union must be carefully studied.

Besides the EU, many other countries have passed legislation protecting privacy and security of data. Argentina has an approach similar to that of the EU. Hong Kong has a *Personal Data Ordinance* [148] that covers both public and private data processors of either electronic or non-electronic documents. Other countries, like Brazil, express the constitutional right to privacy but do not have a comprehensive law on the subject. A different approach is taken by Canada, which has an organization-to-organization approach. The *Personal Information Protection and Electronic Document Act (PIPEDA)* [149] regulates the level of protection for personal data whenever interacting with a Canadian-based company, regardless of its geographical location. In brief, Canadian organizations are held accountable for the protection of personal information they transfer to third parties, whether such parties are inside or outside Canada. All these different regulations are not only a source of international conflicts of law but also identify the immature state of legal support with regard to the globalization of computing utilities trading.

Another element of concern that directly relates to the way sensitive information is handled in the cloud is the management of intellectual property (IP) rights. It is important in this case not to protect the data from entities external to the cloud service provider but to ensure the IP attached to the data, or generated by applications, that is hosted in the cloud still belongs to the user of the cloud. This is

generally the rule of thumb that applies for the protection of IP rights, which needs to be enforced by appropriate contractual terms between the service consumer and the service provider. This is particularly important in case cloud service providers file for bankruptcy and the hosted IP is used to augment the value of the company, thus facilitating the provider's exit from bankruptcy.

Business and commerce-related issues

Legal issues can also arise as a result of specific contractual terms and arrangements between the cloud provider and the user. Contractual terms need to be able to deal with issues such as:

- What is defined as a service that is agreed between the two parties?
- What is the liability of each of the parties?
- What risk are users running while relying on a given provider?
- What are measures that are put in place to guarantee users access to their data?

The first types of legal issues we can consider are contractual ones. These relate to the specific nature of the agreement between the cloud provider and the consumer. Licensing agreements, the most used form of contractual agreement for software before the spread of SOA, are no longer applicable. Service agreements are a more appropriate form for settling arrangements in a cloud computing scenario, since there is no real transmission of a software artifact from the contracting entities and since IT services are offered and priced as utilities. In any market segment of cloud computing (Iaas, Paas, or SaaS), the control and access points are offered by the cloud provider and the agreement covers all the basic terms and conditions of use of the service, with clear specification of provider liabilities in case of service interruption. Generally, users have very limited options and the contractual terms are in the form of online agreements that the user cannot negotiate. Since this is the only documents that binds the two entities, it is really important to have a clear specification of the privacy policy adopted by the provider and the compliance of such policies with the current laws on the subject. It is also important to verify what will be the provider's liability in case of data breach. In most cases, providers have a very limited liability and compensate users with service credit in the best cases. Since these terms are not negotiable, the service agreement is of use to the consumer in carefully evaluating whether to sign an agreement with the provider. As more enterprises and mission-critical applications move to the cloud, a more robust and balanced contractual form is needed.

Some business and commercial considerations may also influence the contractual arrangements between providers and consumers. For instance, measures aimed at minimizing the consumer's, considerations about the provider's viability, and the future availability of the data stored in the cloud are all element of concern. These issues must be referenced in the agreement. Elements that contribute to minimizing the risk are measures aimed at protecting the integrity of data, specific SLAs indicating the guaranteed percentage of availability of the service outside scheduled outages, and disaster-recovery measures. The provider's viability is another aspect that contributes to the final decision to enter an agreement with a given provider. Since vendor lock-in is still a reality, the ability of the provider's business to survive is fundamental also because source-code escrow[22] does not help in this scenario. The risk of bankruptcy is not the only menace that could prevent

[22] *Source-code escrow* is the deposit of software source code with a third-party escrow agent to ensure the maintenance of the software in case the party licensing the software files for bankruptcy.

durable and continuous access to data and applications in the future. Other conditions include termination of the provider's business activity or provider acquisition by another company. In these cases, the SLA with the customer should clearly specify which policies are in place to guarantee durable access to customer data.

Jurisdictional and procedural issues

Jurisdictional and procedural legal issues arise from two major aspects of cloud computing: the geo-location of data and applications served by cloud providers and the laws and regulations that are applied in case of litigation.

Jurisdictional issues are mostly related to location of data and the specific laws that apply in that location. Cloud service providers locate their datacenters in order to reduce their operational costs. The placement of datacenters is influenced by the desire to optimally serve customers on a global scale. For this reason it is quite common to distribute the infrastructure of a single cloud provider over the globe. Specific issues arise from the different laws that are applied for the protection of data. For instance, the EU directive states that any personal data generated within the European Union are subject to European law as well as concerning the export of these data to a third-party country. This limits the mobility of data among datacenters located in different countries, if an appropriate level of data protection is not guaranteed. Furthermore, SLAs are agreed to within a context defined by a specific governing law, but due to the mobility of data, such laws might not be effective and could fail in their purpose of protecting customer rights. The condition is even worse when there is no specific statement indicating the governing law under which the agreement was signed.

Jurisdictional issues may also arise in the case of subcontracting. This is a quite common scenario in the case of cloud federation: A cloud provider leverages other providers' services and facilities to provide services to customers. This is mostly done transparently to the user. In case of failure in service delivery, it will be difficult for the cloud user to identity the real causes. In this case, the scenario is complicated by the fact that, besides different geographies, different organizations are involved in delivering the service to the end user.

Different jurisdictions lead to what is also called the *conflict of laws*, which acknowledges the fact that laws of different countries may operate in opposition to each other, even if they relate to the same subject matter. The general rule of thumb is that, since each nation is sovereign within its own territory, the laws of a country will affect all the people and property within it, including contracts made and actions carried out within its borders. As already observed, the SLA should clearly specify the governing law as well as the other potential jurisdictions that may be involved in delivering the service to the end user.

Besides jurisdictional issues, there are other important aspects concerning the application of the law in case of *litigation*, which is the process of bringing a legal dispute to a court. In this context, one party might require the other party to provide evidence of data in terms of electronic documents (emails, reports, financial records, etc.). This process is also known as *ediscovery* and poses an interesting challenge in the case of cloud computing, where control over data and systems is relegated to third parties. Failing to comply with an ediscovery process because of lack electronically stored information is not an option and in most cases will result in several charges.

Traditionally, enterprises organized staff and internal procedures to respond to an ediscovery process with the required information. In a scenario where an enterprise leverages cloud providers, there is no infrastructure nor staff that directly manages such activities; instead, the maintenance of

data and its accessibility are all offloaded to the cloud provider. Therefore, the challenges concern the storage and retrieval of sensitive data over the long term. Since storage is normally billed and has an operational cost in the cloud, this may pose an additional burden to enterprises that want to leverage the cloud. Second, access to data despite the survival of the provider's business activity is another element of concern with regard to ediscovery.

Implications in a cloud federation scenario

In this section we have mostly discussed the nature of the legal issues that may arise in leveraging cloud computing solutions, from the perspective of the interaction between cloud service consumers and providers. How do these issues impact a cloud federation scenario? And are there other elements to consider from a legal perspective?

We can observe that interactions among different cloud providers can occur in the form of subcontracting, which has been discussed, or as a normal interaction between a consumer and a provider, where the role of the consumer is played by another cloud provider. Therefore, all the legal issues introduced previously apply. As noted, the case in which services offered by different providers are composed together to deliver the final service to the cloud user poses additional complications. This is because the contractual interaction may spawn over heterogeneous geographies and necessarily involve different organizations. Hopefully, the creation of a federation will help define a better body of laws that are compliant with all the legislation of the countries involved in possible transactions among cloud providers and users and uniform across all the federation.

11.3.4 Technologies for cloud federations

Even though the concept of cloud federation or the InterCloud is still immature, there are some supporting technologies that enable the deployment of interoperable clouds, at least from an operational point of view. These initiatives mostly originated within academia and principally focus on supporting interoperability among different IaaS implementations, with a limited implementation of the capabilities defined at the logical level.

11.3.4.1 Reservoir

Resources and Services Virtualization Without Barriers, or *RESERVOIR* [155], is a European research project focused on developing an architecture that supports providers of cloud infrastructures to dynamically partner with each other to extend their capabilities while preserving their administrative autonomy. RESERVOIR defines a software stack enabling interoperation at the IaaS layer and providing support for SLA-based execution of applications on top of the infrastructure overlay that results from the federation of infrastructure providers.

RESERVOIR is based on the concept of dynamic federation: Each infrastructure provider is an autonomous business with its own business goals and that might decide to partner with other businesses when needed. The federation is obtained by means of the RESERVOIR middleware that needs to be deployed at each site. The IT management at a specific site is fully autonomous and dictated to by policies that depend on the site's business goals. When needed, internal IT resources are leased to other providers within the context of a negotiated SLA. The role of RESERVOIR is to orchestrate this process and to minimize the barriers obstructing interoperation among different administrative domains.

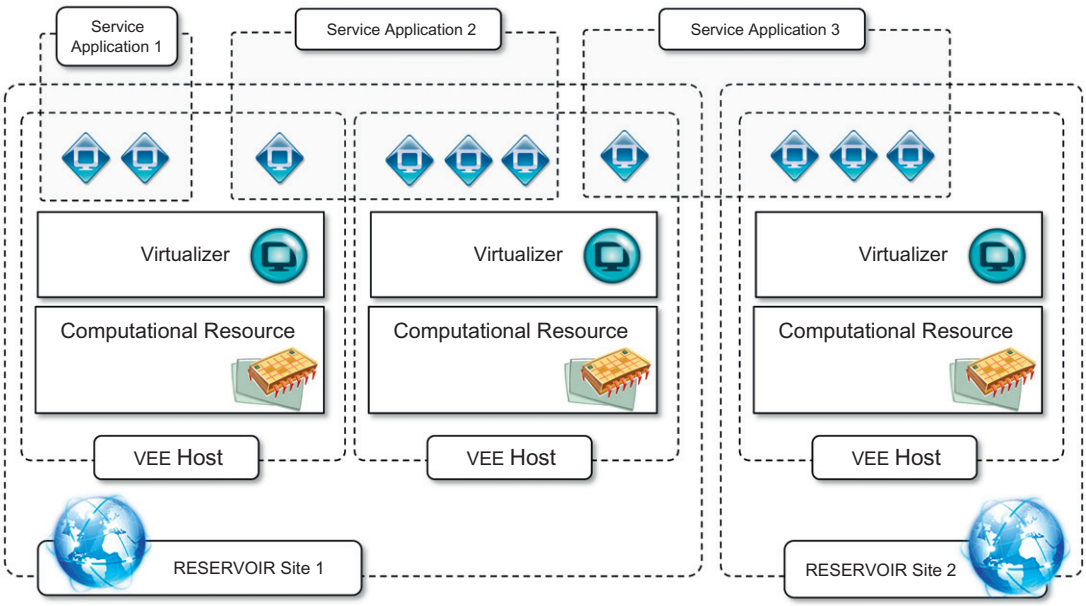

FIGURE 11.10

RESERVOIR cloud deployment.

Figure 11.10 provides a general overview of a RESERVOIR cloud. The framework defines an infrastructure overlay that spans multiple administrative domains and different geographic locations. Each site runs the RESERVOIR software stack and provides an on-demand execution environment in which components of a service application can be deployed and executed. The model introduces a clear separation between service providers and infrastructure providers. Service providers are the entities that understand the needs of a particular business and offer service applications to address those needs; they do not own infrastructure but rather lease it from infrastructure providers. Infrastructure providers operate RESERVOIR sites and offer a virtually infinite pool of computational, network, and storage resources. Service providers define service applications that are modeled as a collection of components that can be deployed over a distributed virtual infrastructure, which can be either explicitly or implicitly provisioned. In the first case, the service provider conducts sizing and capacity-planning studies to identify the appropriate number of components to be required for a given workload condition. The specification is obtained by means of minimal service configuration and a set of elasticity rules that are used by RESERVOIR to dynamically provision resources under varying workload conditions. In the second case, the service provider provides neither a minimal service configuration nor elasticity rules. The sizing is automatically made by the RESERVOIR middleware, which tries to minimize overprovisioning. Service providers are billed on a pay-as-you-go model and can ask for usage reports to verify the billing.

Service application components are represented as virtual packages, which also specify execution environment requirements. Infrastructure providers can leverage any virtualization technology

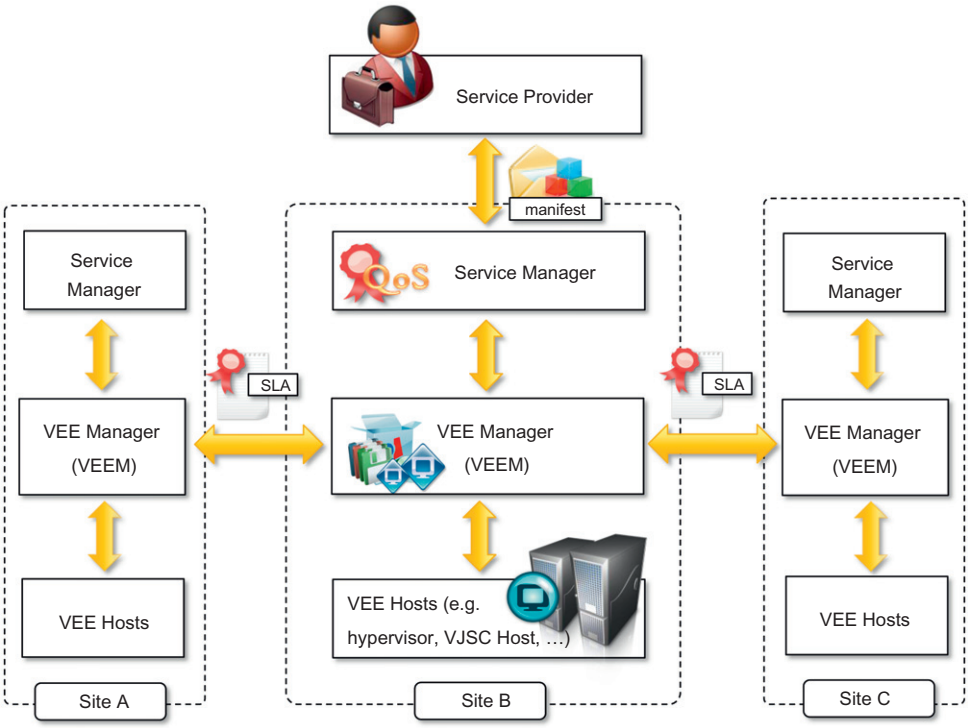

FIGURE 11.11

RESERVOIR architecture.

of choice and are in charge of providing the required environment for such applications. An important element for determining the virtual environment for service applications is the *service manifest*. This is one of the key elements of the RESERVOIR model and specifies the structure of the service applications in terms of component types that are to be deployed as *virtual execution environments (VEEs)*. The service manifest contains a reference to a master image, which is a self-contained software stack (OS, middleware, applications, data, and configuration) that fully captures the functionality of the component type. Additional information and rules specify how many instances to deploy and how to dynamically grow and shrink their number. Moreover, the manifest also specifies the grouping of components into virtual networks and/or tiers that form the service applications. The manifest is expressed by extending the Open Virtualization Format to simplify interoperability and leverage an already existing popular standard.

Figure 11.11 describes the internal architecture of a RESERVOIR site that enables the federated and SLA-based execution of applications. The RESERVOIR stack consists of three major components:

- *Service Manager*. The Service Manager is the highest level of abstraction and constitutes the front-end used by service providers to submit service manifests, negotiate pricing, and monitor

applications. This component deploys and provisions VEEs according to the service manifest and monitors and enforces SLA compliance by controlling the capacity of a service application.

- *Virtual Execution Environment (VEE) Manager*. This component is the core of the RESERVOIR middleware and is responsible for the optimal placement of VEEs into VEE hosts according to the constraints expressed by the Service Manager. Moreover, the VEE Manager also interacts with VEE Managers in other sites to provision additional instances for the execution of service applications or move VEEs to other sites in case of overload. This component realizes the cloud federation.
- *VEE Host (VEEH)*. This is the lowest level of abstraction and interacts with the VEE Manager to put into practice the IT management decisions regarding heterogeneous sets of virtualization platforms. This level is also in charge of ensuring appropriate and isolated networking among VEEs that belong to the same application. The VEEH encapsulates all platform-specific management that is required to expose the used virtualization technology through a standardized interface to the VEE Manager.

Each of these components adopts a standardized interface and protocol to interoperate with the adjacent layer. This design decision was made to promote a variety of innovative approaches at each layer. It offers infrastructure providers the freedom to integrate the RESERVOIR middleware with their own existing technology.

11.3.4.2 InterCloud

InterCloud is a service-oriented architectural framework for cloud federation that supports utility-driven interconnection of clouds. It is composed of a set of decoupled elements that interact via a market-oriented system to enable trading of cloud assets such as computing power, storage, and execution of applications. As depicted in Figure 11.12, the InterCloud model comprises two main elements: CloudExchange and CloudCoordinator:

- *CloudExchange*. This is the market-making component of the architecture. It offers services that allow providers to find each other in order to directly trade cloud assets, as well as allowing parties to register and run auctions. In the former case, CloudExchange acts as a directory service for the federation. In the latter case, it runs the auction. For offering such services to the federation, CloudExchange implements a Web service-based interface that allows datacenters to join and leave the federation; to publish resources they want to sell; to register their resource requirements so that parties interested in selling providers are able to locate potential buyers for their resources; to query resource offers that match specific requirements; to query requirements that match available resources from a party; to withdraw offers and requests from the coordinator; to offer resources in auctions; to register bids; and to consult the status of a running auction.
- *CloudCoordinator*. This component manages domain-specific issues related to the federation. This component is present on each party that wants join the federation. CloudCoordinator has front-end components (i.e., elements that interact with the federation) as well as back-end components (i.e., components that interact with the associated datacenter). Front-end components interact with the CloudExchange and with other coordinators. The former allows datacenters to announce their offers and requirements, whereas the latter allows the Coordinator to learn about the current state of the datacenter to decide whether actions from the federation

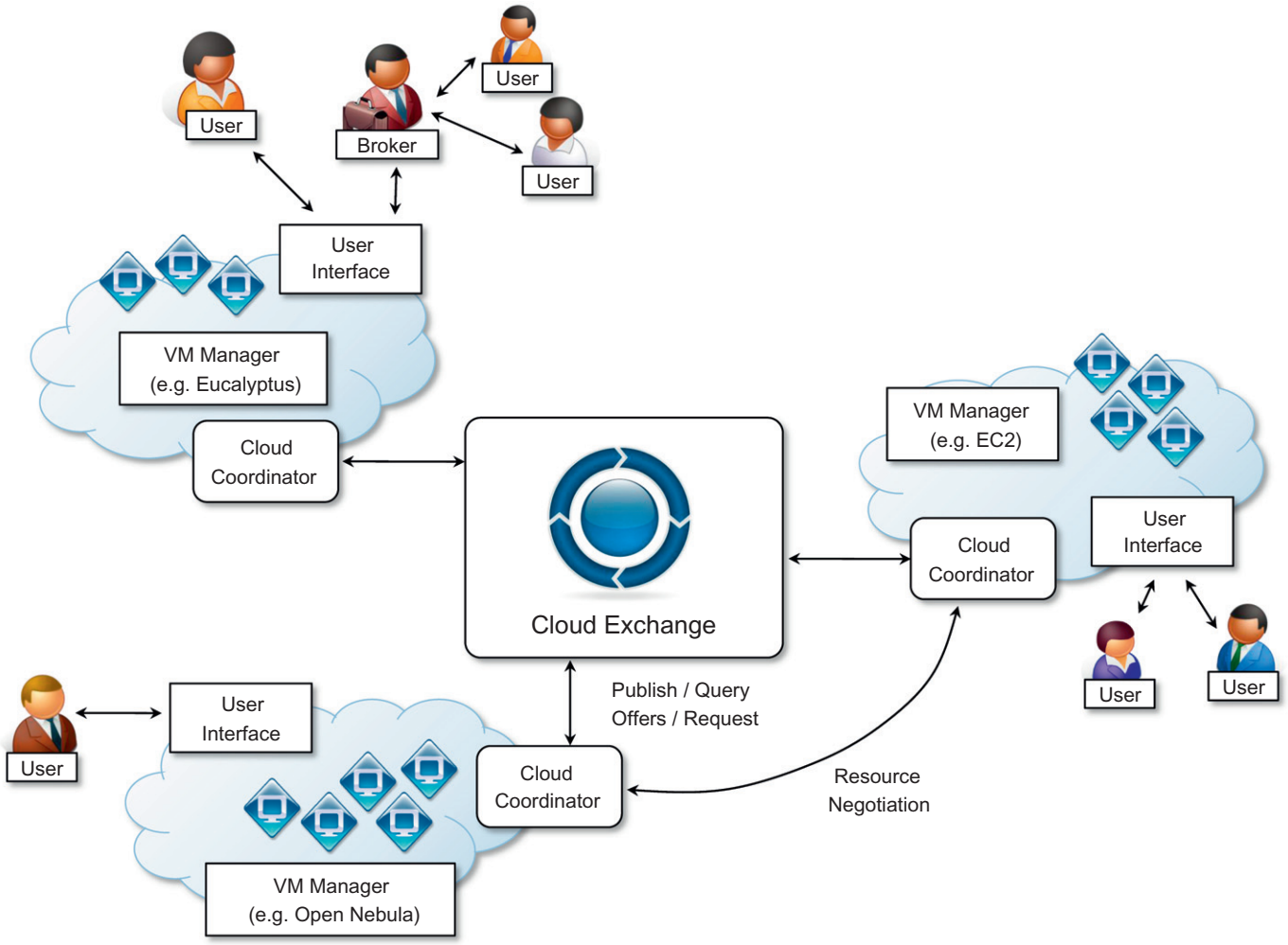

FIGURE 11.12

InterCloud architecture.

are required or not. Therefore, wherever the Coordinator detects that extra resources are required by the datacenter, it triggers the process of discovery of potential providers (by interacting with the cloud federation). Once potential providers are discovered and the preferred one is selected, the Coordinator contacts the remote Coordinator and negotiates. Similarly, when the Coordinator detects that local resources are underutilized, they can publish an offer for resources in the CloudExchange or they can look for matches among requirements registered in the Exchange service.

Negotiation between parties follows the Alternate Offers protocol. The value that each Coordinator is willing to pay for a resource, as well as the value for which each Coordinator wants to sell resources, is not defined by the federation; instead, each Coordinator is free to value resources according to utilization and profit estimation (or any other criteria that a federation member wants to consider), and they are also free to reject offers that are not attractive for them. This flexibility of providers in deciding how and when to buy or sell resources, how many of them, when, and for what price is a big motivator for providers to join an InterCloud federation. Providers are free to refuse to buy or sell resources if they disagree with the price; similarly, they may opt for using the method that seems more profitable to them (e.g., auction or fixed price).

Moreover, InterCloud acts primarily in the operational layer. That means that issues related to security, VM images, and networking are not handled by the framework. Similarly, providers' obligations and expectations for joining the federation are also not addressed by the framework. Therefore, existing approaches for these layers, or even new solutions for them, can be applied in the InterCloud without changes to its architecture.

11.3.5 Observations

Cloud service providers, which may be competing with each other, need to have clear objectives and incentives for establishing a federation. One primary reason can be to improve the overall QoS perceived by their customers. The second step toward the creation of a federated scenario is the definition of a reference model for the cloud federation, together with a set of rules and policies. This is the level where market-oriented models are adopted and opportunities for renting and leasing resources develop. The development and maturity at this stage are still very limited. At a very low level the federation of different cloud vendors is realized through interoperable technologies. This is an area that has seen considerable development, especially concerning the drafting and adoption of standards that support interoperability. We have discussed the most relevant initiatives in this direction and analyzed some reference and prototypal implementations of systems enabling cloud federation at the IaaS level.

11.4 Third-party cloud services

One of the key elements of cloud computing is the possibility of composing services that belong to different vendors or integrating them into existing software systems. The service-oriented model, which is the basis of cloud computing, facilitates such an approach and provides the opportunity for developing a new class of services that can be called *third-party cloud services*. These are the

result of adding value to preexisting cloud computing services, thus providing customers with a different and more sophisticated service. Added value can be either created by smartly coordinating existing services or implementing additional features on top of an existing basic service.

Besides this general definition, there is no specific feature that characterizes this class of service. Therefore, in this section, we describe some examples of third-party services.

11.4.1 MetaCDN

MetaCDN [158] provides users with a Content Delivery Network (CDN) [159] [service by leveraging and harnessing together heterogeneous storage clouds. It implements a software overlay that coordinates the service offerings of different cloud storage vendors and uses them as distributed elastic storage on which the user content is stored. MetaCDN provides users with the high-level services of a CDN for content distribution and interacts with the low-level interfaces of storage clouds to optimally place the user content in accordance with the expected geography of its demand. By leveraging the cloud as a storage back-end it makes a complex—and generally expensive—content delivery service available to small enterprises.

The architecture of MetaCDN is shown in Figure 11.13. The MetaCDN interface exposes its services through users and applications through the Web; users interact with a portal, while applications take advantage of the programmatic access provided by means of Web services. The main operations of MetaCDN are the creation of deployments over storage clouds and their management. The portal constitutes a more intuitive interface for users with basic requirements, while the Web service provides access to the full capabilities of MetaCDN and allows for more complex and sophisticated deployment.

In particular, four different deployment options can be selected:

- *Coverage and performance-optimized deployment*. In this case MetaCDN will deploy as many replicas as possible to all available locations.
- *Direct deployment.* In this case MetaCDN allows the selection of the deployment regions for the content and will match the selected regions with the supported providers serving those areas.
- *Cost-optimized deployment.* In this case MetaCDN deploys as many replicas in the locations identified by the deployment request. The available storage transfer allowance and budget will be used to deploy the replicas and keep them active for as long as possible.
- *QoS optimized deployment.* In this case MetaCDN selects the providers that can better match the QoS requirements attached to the deployment, such as average response time and throughput from a particular location.

A collection of components coordinate their activities in order to offer the services we described. These constitute the additional value that MetaCDN brings on top of the direct use of storage clouds by the users. Of particular importance are three components: the *MetaCDN Manager*, the *MetaCDN QoS Monitor*, and the *Load Redirector*. The Manager is responsible for ensuring that all the content deployments are meeting the expected QoS. It is supported in this activity by the Monitor, which constantly probes storage providers and monitors data transfers to assess the performance of each provider. Content serving is controlled by the Load Redirector, which is in charge of redirecting user content requests to the most suitable replica given the condition of the systems and the options specified during the deployment. Interactions with storage

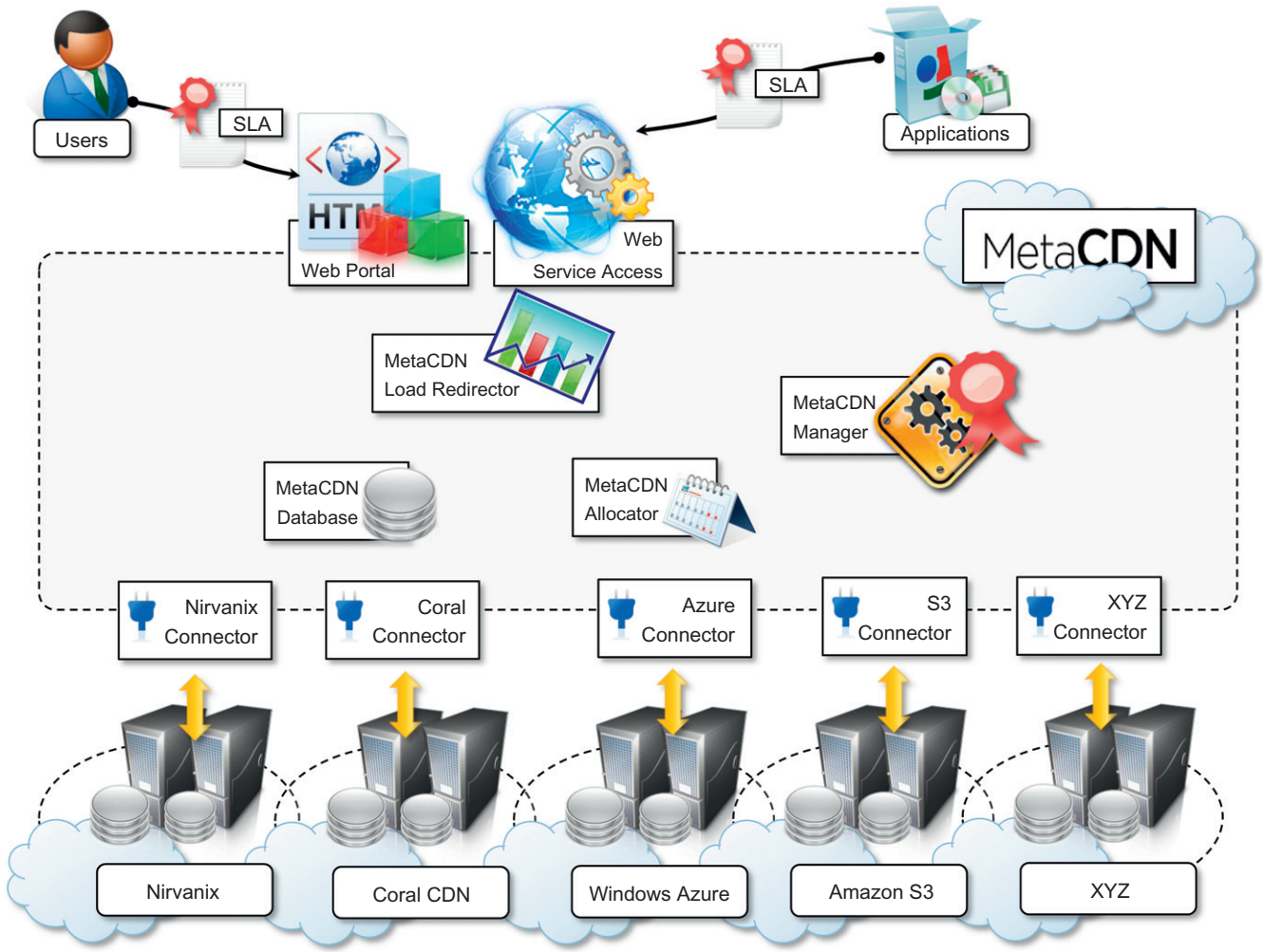

FIGURE 11.13

MetaCDN architecture.

clouds are managed by means of connectors, which abstract away the different interfaces exposed by the providers and present a uniform interface within the MetaCDN system.

As discussed, the core value of MetaCDN resides in the software overlay that enables the uniform use of heterogeneous storage clouds as a single, large, distributed content delivery network. The advantage is not only given by providing a CDN service at accessible costs but also in enriching the original service offering of existing cloud services with additional functionalities, thus creating a new and more sophisticated service that is of interest to a new market sector.

11.4.2 SpotCloud

SpotCloud has already been introduced as an example of a virtual marketplace. By acting as an intermediary for trading compute and storage between consumers and service providers, it provides the two parties with added value. For service consumers, it acts as a market directory where they can browse and compare different IaaS service offerings and select the most appropriate solution for them. For service providers it constitutes an opportunity for advertising their offerings. In addition, it allows users with available computing capacity to easily turn themselves into service providers by deploying the runtime environment required by SpotCloud on their infrastructure (see Figure 11.14).

SpotCloud is not only an enabler for IaaS providers and resellers, but its intermediary role also includes a complete bookkeeping of the transactions associated with the use of resources. Users deposit credit on their SpotCloud account and capacity sellers are paid following the usual pay-per-use model. SpotCloud retains a percentage of the amount billed to the user. Moreover, by leveraging a uniform runtime environment and virtual machine management layer, it provides users with a vendor lock-in-free solution, which might be strategic for specific applications.

The two previously presented examples give an idea of how different in nature third-party services can be: MetaCDN provides end users with a different service from the simple cloud storage offerings; SpotCloud does not change the type of service that is finally offered to end users, but it enriches it with additional features that result in more effective use of it. These are just two examples of the market segment that is now developing as a result of the consolidation of cloud computing as an approach to a more intelligent use of IT resources.

SUMMARY

This chapter introduced some advanced topics that will characterize and drive research and development in cloud computing. These topics become particularly important as cloud computing technologies become increasingly consolidated.

Given the huge size of the computing facilities backing the service offerings of cloud providers, energy-efficient solutions play a fundamental role. These involve smarter and greener datacenter designs, efficient placement of virtual machines, and energy-driven server management. Energy-efficient solutions not only help cut power bills, they also reduce the impact of computing facilities on the environment. Although the advantage cloud computing brings to this issue is still controversial, it has the potential to deliver greener technology.

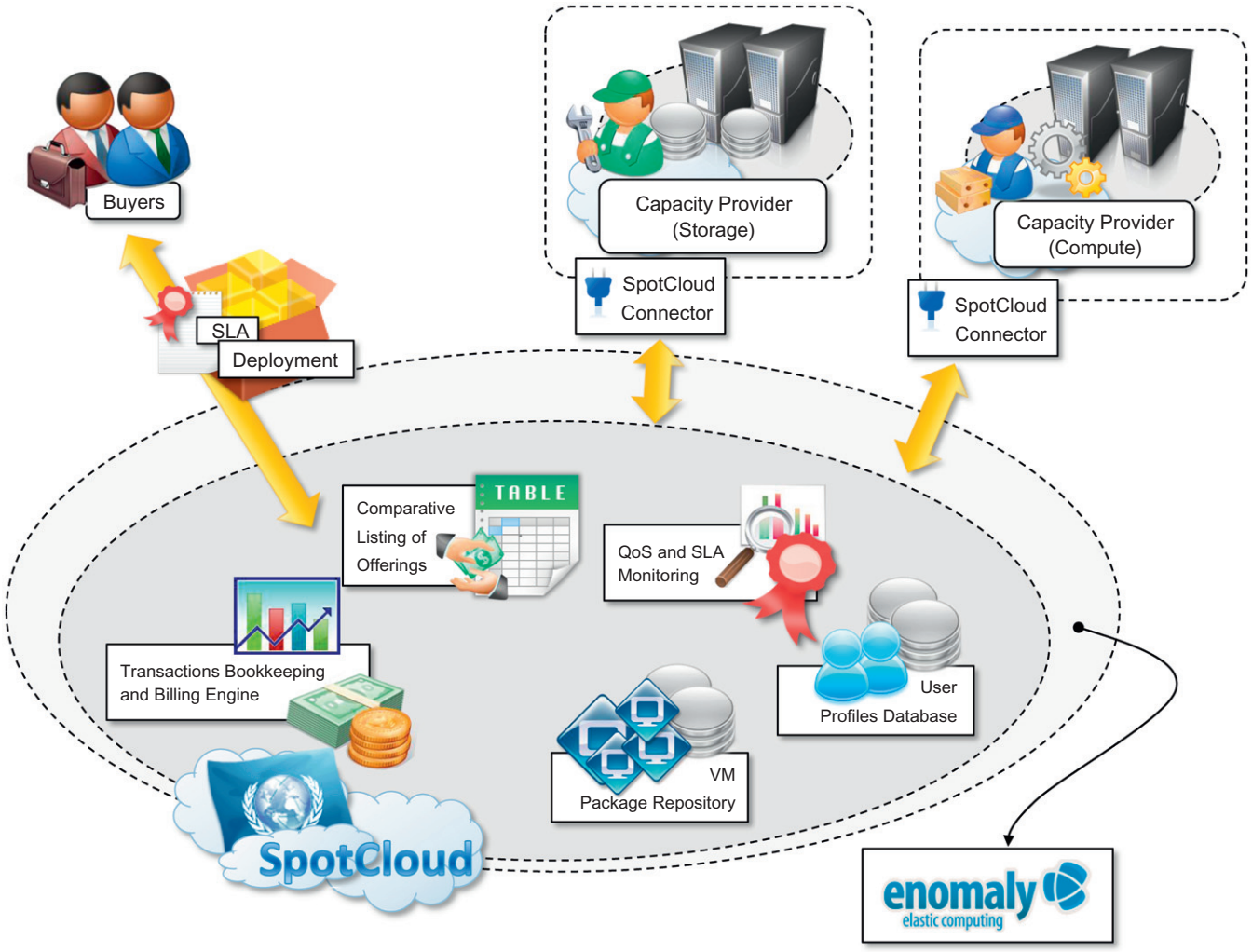

FIGURE 11.14

SpotCloud market architecture.

Besides energy efficiency, market-oriented cloud computing (MOCC) and InterCloud arrangements for optimal service delivery constitute important advancements. MOCC is the natural evolution of cloud computing once the technology enabling it is consolidated and commoditized. Maturity in the IaaS market segment facilitates the creation of more sophisticated market models. Recent advances have shown that cloud service brokering- and auction-based models are useful strategies for better resource management. MOCC's ultimate goal is to create a global virtual marketplace that facilitates computing utilities trading.

Moreover, cloud federations support the implementation of MOCC by deploying infrastructure and defining organizational models enabling interoperability among cloud providers. Currently, customers rely on a single provider or compose services from different cloud vendors by themselves. With the advent of cloud federation, users will be able to leverage multiple clouds.

Finally, we discussed third party services such as MetaCDN and SpotCloud, which are built by leveraging services offered by multiple cloud vendors.Several SaaS solutions might be classified as third-party services, which are essentially value-added resellers. This means that they compose and use other cloud services, adding value to and selling them.

Review questions

1. What is energy-efficient computing, and why it is fundamental to cloud computing?
2. What is the basic principle behind all the power management techniques?
3. What does DVFS stand for, and how will it help in cloud computing?
4. Compare hardware and software power management techniques.
5. What is server consolidation? How it can be considered a power management strategy?
6. What is market-oriented cloud computing?
7. What are the main components that implement a MOCC-based system?
8. What is a virtual marketplace?
9. What is a cloud federation?
10. What are the three different levels that express the concept of the InterCloud and cloud federation?
11. What are the motivations leading to the creation of a federation?
12. What kind of challenges must be addressed at the operational and logical level?
13. What is the domain of the infrastructural level, and which layers of the Cloud Computing Reference Model represent this level?
14. What kind of standards and protocols can be used to achieve interoperability in a cloud federation?
15. What is the InterCloud?
16. Describe the major characteristics of the RESERVOIR project.
17. Describe the role of standards in a federated environment.
18. What are the security implications in a federated environment?
19. What are the possible legal issues arising in the context of a cloud federation?
20. What is a third-party cloud service?
21. Give some examples of a third-party cloud service.

This page intentionally left blank

References

[1] Tanenbaum AS, Van Steen M. Distributed systems: principles and paradigm. Upper Saddle River, NJ, USA: Prentice Hall PTR; 2001.

[2] Coulouris G, Dellimore J, Kindberg T. Distributed systems: concepts and design. 4th ed. Boston, MA, USA: Addison Wesley; 2005.

[3] Pfister G. In search of clusters. 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR; 1998.

[4] Buyya R. High-performance cluster computing: architecture and systems. Upper Saddle River, NJ, USA: Prentice Hall PTR; 1999.

[5] Thain D, Tannenbaum T, Livny M. Distributed computing in practice: the condor experience. Concurrency Comput Pract Exper 2005;17(2−4):323−56.

[6] Sunderam VS. PVM: a framework for parallel distributed computing. Concurrency: Pract Exper 1990;2(4):315−39.

[7] Gropp W, Lusk E, Doss N, Skjellum A. A high-performance, portable implementation of the MPI message passing interface. Parallel Comput 1996;22(6):789−828.

[8] Foster I, Kesselman C. The grid: blueprint for a new computing infrastructure. San Francisco, CA, USA: Morgan Kaufmann Publishing; 1999.

[9] Foster I, Kesselman C, Tuecke S. The anatomy of the grid: enabling scalable virtual organizations. Int J High Perform Comput Appl 2001;15(3).

[10] Broberg J, Venugopal S, Buyya R. Market oriented grids and utility computing: the state of the art and future directions. J Grid Comput 2008;6(3):255−76 [Springer Netherlands].

[11] Buyya R, Bubendorfer K. Market-oriented grid computing and utility computing. Hoboken, NJ, USA: Wiley; 2009.

[12] Foster I. Globus toolkit version 4: software for service-oriented systems. IFIP International conference on network and parallel computing. Lecture notes in computer science (LNCS) 3779. Springer-Verlag; 2005. pp. 2−13.

[13] Gentzsch W. Sun grid engine: towards creating a compute power grid. Proceedings of the first IEEE/ACM international symposium on cluster computing and the grid (CCGrid 2001). Brisbane, Australia: IEEE Computer Society; 2001. pp. 35−36.

[14] Anderson DP. BOINC: a system for public-resource computing and storage, Proceedings of the fifth IEEE/ACM international workshop on grid computing. Pittsburgh, PA, USA: 2004.

[15] Buyya R, Venugopal S. The gridbus toolkit for service oriented grid and utility computing: an overview and status report. Proceedings of the first IEEE international workshop on grid economics and business models (GECON 2004). NJ, USA: IEEE Press; 2004.

[16] Popek GJ, Goldberg RP. Formal requirements for virtualizable third generation architectures. Commun ACM 1974;17(7):412−21.

[17] Whitaker A, Shaw M, Gribble D. Denali: a scalable isolation kernel. Proceedings of the tenth ACM SIGOPS European workshop. France: Saint-Emilion; 2002.

[18] Chaudhury A, Kuilboer JP. e-Business and e-Commerce infrastructure. NY, USA: McGraw Hill; 2002.

[19] Papazoglou MP, Traverso P, Dustdar S, Leymann. F. Service-oriented computing: state of the art and research challenges. IEEE Comput 2007;40(11):38−45 [IEEE Computing Society]

[20] Papazoglou MP, van den Heuvel. W-J. Service-Oriented Architecture: Approaches, Technologies, and Research Issues. Very Large Data Bases (VLDB) J 2007;16(3):389−415 VLDB Endowment Inc.

[21] Papazoglou MP. Web services: principles and technology. Hoboken, NJ, USA: Prentice Hall; 2007.

[22] W3C. Web service definition language (WSDL) 1.1. [Online Document] Available at <www.w3.org/TR/wsdl/>.
[23] W3C. Simple object access protocol (SOAP) specifications. [Online Document]. Available at <www.w3.org/TR/soap/>.
[24] Liberty J, Hurwitz D. Programming ASP.NET. 3rd ed. Sebastopol, CA, USA: O'Reilly Media; 2005.
[25] Ka lok Tong K. Developing web services with apache axis2. Birmingham, UK: TipTec Development; 2008.
[26] DiNucci D. Fragmented future. Design & New Media; 1999 [Online Document]. Available at <www.cdinucci.com/Darcy2/articles/Print/Printarticle7.html>
[27] O'Reilly T. What is web 2.0? Design patterns and business model for the next generation of software. O'Reilly Media; 2005 [Online Document]. Available at <http://oreilly.com/pub/a/web2/archive/what-is-web-20.html>
[28] Armbrust M, Fox A, Griffith R, Joseph A, Katz R, Konwinski A, et al. Technical Report No. UCB/EECS-2009-28 Above the clouds: a berkeley view of cloud computing. USA: University of California at Berkeley; 2009
[29] Reese G. Cloud application architectures: building applications and infrastructure in the cloud. Sebastopol, CA, USA: O'Reilly Media Inc.; 2009.
[30] Buyya R, Yeo CS, Venugopal S. Market oriented cloud computing: vision, hype, and reality for delivering IT services as computing utilities. Proceedings of the tenth conference on high performance computing and communications (HPCC 2008, IEEE Press, Los Alamitos, CA). Dalian, China: 2008.
[31] Bennett KH, Layzell PJ, Budgen D, Brereton P, Macaulay LA, Munro M. Service-based software: the future for flexible software, Proceedings of the seventh Asia-Pacific software engineering conference (ASPEC 2000). Singapore: IEEE Computer Society; December 2000.
[32] Strategic backgrounder: software as a service. Software & Information Industry Association; February 2001. [Online Document] Available at <www.siia.net/estore/pubs/SSB-01.pdf>.
[33] Koenig M, Guptill B, McNee B, Cassell J. SaaS 2.0: Software-as-a-Service as next gen business platform. Saugatuck Technologies; 2006. Available at <www.saugatech.com/239order.htm>
[34] Staten J, Yates S, Ryme J, Gillett F, Nelson LE. Deliver cloud benefits inside your walls: economic and self-service gains are within reach. Forrester Research Inc.; 2009.
[35] Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, et al. Xen and the art of virtualization. Proceedings of the 19th ACM symposium on operating systems principles (SOSP). Lake George, NY, USA: October 2003.
[36] Warnke R, Ritzau T. ISBN: 978-3-8370-0876-0 qemu-kvm& libvirt. 4th ed. Norderstedt: Books on Demand GmbH; 2010
[37] Nurmi D, Wolski R, Grzegorczyk C, Obertelli G, Soman S, Youseff L, et al. The eucalyptus open-source cloud computing system. In: Proceedings ninth IEEE/ACM international symposium on cluster computing and the grid (CCGrid 2009). Shanghai, China: May 2009.
[38] Llorente I, Moreno-Vozmediano R, Montero. R. Cloud computing for on-demand grid resource provisioning. Advances in parallel computing, 18. IOS Press; 2009.
[39] Sotomayor B, Keahey K, Foster I. Combining batch execution and leasing using virtual machines. HPDC '08: Proceedings of the 17th international symposium on high performance distributed computing. ACM; 2008. pp. 87–96.
[40] Venugopal S, Broberg J, Buyya R. OpenPEX: an open provisioning and execution system for virtual machines. Proceedings of the 17th international conference on advanced computing and communications (ADCOM 2009). Bengaluru, India: December. 14–18, 2009.
[41] Costanzo A, Assunção M, Buyya R. Harnessing cloud technologies for a virtualized distributed computing infrastructure. IEEE Internet Comput 2009;13(5):14–22.

[42] Vecchiola C, Chu X, Mattess M, Buyya R. Aneka: integration of public and private clouds. Cloud computing: principles and paradigms. Hoboken, NJ, USA: Wiley; 2011.
[43] Mell P, Grance T. NIST working definition on cloud computing. National Institute of Standard and Technology (NIST); [Online Document] Available at <http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf>.
[44] Briscoe G, De Wilde P. Digital ecosystems: evolving service oriented architectures. Proceedings of the first international conference of bio inspired models of network Information and Computing Systems (BIONETICS). Cavalese, Italy: IEEE Press; December 2006.
[45] G. Briscoe and M. Alexandros. Digital ecosystems in the clouds: towards community cloud computing. Proceedings of the third IEEE international conference on digital ecosystems and technologies (DEST 2009). New York, USA: IEEE; 2009. pp. 103−108.
[46]. Sriram I, Khajeh-Hosseini A. Research agenda in cloud technologies Technical Report. UK: University of Bristol; 2010. [Online Document] Available at <http://arxiv.org/ftp/arxiv/papers/1001/1001.3259.pdf>.
[47] Khajeh-Hosseini A, Sriram I, Sommerville I. Research challenges for enterprise cloud computing. UK: University of Bristol; 2010 [Online Document] Available at <http://arxiv.org/ftp/arxiv/papers/1001/1001.3257.pdf>.
[48] Vouk AM. Cloud computing issues, research, and implementations, Proceedings of the 30th international conference of information technologies and interfaces (ITI 2008). Cavtat/Dubrovnick, Croatia: June 2008. pp. 31−40.
[49] Birman K, Chockler G, van Renesse R. Toward a cloud computing research agenda. SIGACT News 2009;40(2):68−80.
[50] Youssef L, Butrico M, Da Silva D. Towards a unified ontology of cloud computing. Grid computing environments workshop (GCE08). Austin, Texas, USA: IEEE; November 2008.
[51] Open virtualization format specification. Distributed Management Task Force; February 2009. [Online Document] Available at <www.dmtf.org/standards/published_documents/DSP0243_1.0.pdf>.
[52] Standard ECMA-334. C# Language specification. Available at <www.ecma-international.org/publications/standards/Ecma-334.htm>.
[53] Standard ECMA-335. Common language infrastructure (CLI). Available at <www.ecma-international.org/publications/standards/Ecma-335.htm>.
[54] Ghemawat S, Gobioff H, Leung S-T. The google file system. Proceedings of the 19th ACM symposium of operating systems principles (SOSP'03). Lake George, NY, USA: ACM; October 2003. pp. 29−43.
[55] Dean J. Ghemawat S. MapReduce: simplified data processing on large clusters. Proceedings of the 6th symposium on operating system design and implementation (OSDI'04) USENIX. San Francisco, CA, USA: December 2004.
[56] Jin C, Buyya. R. Dataflow computations on enterprise grids. ISBN: 978-981-283-943-5. In: Misra S, Misra SC, Woungang I, editors. Selected topics in communication networks and distributed systems. Singapore: World Scientific; 2010.
[57] Agha. G. ISBN: 0-262-01092-5 Actors: a model of concurrent computation in distributed systems. Cambridge, MA, USA: MIT Press; 1986
[58] Buyya R, Ranjan R, Calheiros RN. InterCloud: utility-oriented federation of cloud computing environments for scaling of application services. Proceedings of the tenth international conference on algorithms and architectures for parallel processing (ICA3PP'10), LNCS 6081. Springer; 2010. pp. 13−31.
[59] Fox GC, Williams RD, Messina PC. Parallel computing works. San Francisco, CA, USA: Morgan Kaufmann; 1994.
[60] Gray J, Reuter A. Transaction processing: concepts and techniques. San Mateo, CA, USA: Morgan Kaufmann; 1992.

[61] Raicu I. Many-task computing: bridging the gap between high throughput computing and high performance computing. Saarbrücken, Germany: VDM Verlag; 2009.
[62] Wilde M, Foster I, Iskra K, Beckman P, Zhang Z, Espinosa A, et al. Parallel scripting for applications at the petascale and beyond. IEEE Comp 2009;42(11):50−60 IEEE Computing Society.
[63] Raicu I, Zhang Z, Wilde M, Foster I, Beckman P, Iskra K, Clifford B. Toward loosely coupled programming on petascale systems. Proceedings of the 2008 ACM/IEEE conference on supercomputing (SC08). Austin, TX, USA: November 15−21, 2008.
[64] Hollingsworth D. The workflow reference model. Workflow Management Coalition, Document nr. tc00-1003. 1993. [Online Document] Available at <www.wfmc.org/standards/docs/tc003v11.pdf>.
[65] The OASIS Committee. Web services business process execution language (WS-BPEL) Version 2.0. 2007. [Online Document] Available at http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.
[66] Barker A, van Hemert J. Scientific workflow: a survey and research directions. Proceedings of the seventh international conference parallel processing and applied mathematics (PPAM 2007). September 9−12, 2007.
[67] Yu J, Buyya. R. A taxonomy of scientific workflow systems for grid computing. SIGMOD Rec 2005;34 (3):44−9.
[68] Ludäscher B, Altintas I, Berkley C, Higgins D, Jaeger E, Jones M, et al. Scientific workflow management and the kepler system. Concurrency Comput Pract Exper 2005;18(10):1039−45.
[69] Couvares P, Kosar T, Roy A, Weber J, Wenger K. Workflow management in condor, Workflow for e-Science. London: Springer; 2007. pp. 357−375.
[70] Pandey S, Karunamoorthy D, Buyya R. Workflow engine for clouds ISBN-13: 978-0470887998 In: Buyya R, Broberg J, Goscinski A, editors. Cloud computing: principles and paradigms. New York, USA: Wiley Press; 2010.
[71] Vecchiola C, Kirley M, Buyya R. Multi-objective problem solving with offspring on enterprise clouds. Proceedings of the tenth international conference on high-performance computing in asia-pacific region (HPC Asia 2009). Kaoshiung, Taiwan: 2009.
[72] Buck JT, Ha S, Lee EA, Masserschmitt DG. Ptolemy: a framework for simulating and prototyping heterogeneous systems. J Comput Simul 2004;4:155−82.
[73] Moore R, Prince TA, Ellisman M. Data-intensive computing and digital libraries. Commun ACM 1998;41(11):56−62.
[74] Gorton I, Greenfield P, Szalay A, Williams. R. Data-intensive computing in the 21st century. IEEE Comput 2010.
[75] Johnston W. High-speed, Wide area, data-intensive computing: a ten years retrospective. Proceedings of the seventh symposium high-performance distributed computing. IEEE Press; 1998. pp. 280−291.
[76] Thompson M, Johnston W, Guojun J, Lee J, Tierney B, Terdiman JF. Distributed healthcare imaging information systems, PACS Design and Evaluation: Engineering and Clinical Issues. SPIE Medical Imaging; 1997.
[77] Johnston W, Jin G, Larsen C, Lee J, Hoo G, Thompson M, et al. Real-time generation and cataloguing of large object in widely distributed environments. Int J Digit Libr Spec Issue Digit Libr Med 1997; November.
[78] Lau S, Leclerc Y. Technical Note 540 TerraVision: a terrain visualization system. Menlo Park, CA: SRI International; 1994.
[79] Venugopal S, Buyya R, Kotagiri. R. A taxonomy of data grids for distributed data sharing, management, and processing. ACM Comput Surv 2006;38(1):1−53.
[80] Chervenak A, Foster I, Kesselman C, Salisbury C, Tuecke S. The data grid: towards an architecture for the distributed management and analysis of large scientific datasets. J Netw Comput Appl 2000;23(3):187−200.

[81] Stark C, Breitkreutz BJ, Chatr-Aryamontri A, Boucher L, Oughtred R, Livstone MS, et al. The BioGRID Interaction Database: 2011 Update. Nucleic Acids Res 2010.
[82] Jacobs. A. The pathologies of big data. NY, USA: ACMQueue, ACM Press; 2009.
[83] White T. Hadoop: the definitive guide. Sebastopol, CA, USA: O'Reilly & Associates, Inc.; 2009.
[84] Gu Y, Grossman RL. Sector and sphere: the design and implementation of a high performance data cloud. Phil Trans R Soc 2009;367(1897):2429−45 A 28.
[85] Ceri S, Pelagatti G. Distributed databases: principles and systems. New York, USA: McGraw-Hill; 1984.
[86] Codd. EF. A relational model for large shared data banks. Commun ACM 1970;13(6):377−87.
[87] Oram A. Peer-to-peer: harnessing the power of disruptive technologies. Sebastopol, CA, USA: O'Reilly & Associates, Inc.; 2001.
[88] Schmuck F Haskin R. GPFS: a shared-disk file system for large computing clusters. Proceedings of file and storage technologies 2002 (FAST 2002). Monterey, CA, USA: January 2002.
[89] Gu Y, Grossman RL. UDT: UDP-based data transfer for high-speed wide area networks. Comput Netw 2007;51(7) (Elsevier)
[90] Chodorow K, Dirolf M. ISBN: 978-1449381561 MongoDB: the definitive guide. Sebastopol, CA, USA: O'Reilly Media; 2010.
[91] Anderson JC, Lehnardt J, Slater. N. ISBN: 978-0596155896. CouchDB: the definitive guide: time to relax. Sebastopol, CA, USA: O'Reilly Media; 2010.
[92] DeCandia G, Hastorum D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, et al. Dynamo: amazon's highly available key-value store. Proceedings of the 21st symposium on operating system principles. Stevenson, WA, USA: October 14−17, 2007.
[93] Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, et al. Bigtable: a distributed storage system for structured data. Proceedings of the seventh USENIX symposium on operating system design and implementation. Seattle, USA: November 2006.
[94] Lakshman A Malik P. Cassandra: a decentralized structured storage system. Proceedings of the third ACM SIGOPS international workshop on large scale distributed systems and middleware (LADIS 2009). Big Sky, MT, USA: October 2009.
[95] Pike R, Dorward S, Griesemer R, Quinland S. Interpreting the data: parallel analysis with sawzall. Sci Program J 2005;13(4):227−98.
[96] M. Burrows. The chubby lock service for loosely coupled distributed systems. Proceedings of the seventh USENIX symposium on operating system design and implementation (OSDI '06). Seattle, WA, USA: November 2006.
[97] Chu CT, Kim SK, Lin YA, Yu YY, Bradski G, Ng AY, et al. Map-reduce for machine learning on multicore. In: Schölkopf B, Platt J, Hoffman T editors. Advances of neural information processing systems. 19, 2007.
[98] Yang HC, Dasdan A, Hsiao RL, Stott Parker D. Map-Reduce-Merge: simplified relational data processing on large clusters. Proceedings of the 2007 ACM SIGMOD international conference on management of data. Beijing, China: June 2007.
[99] Ekanayake J, Li H, Zhang B, Gunarathne T, Bae S-H, Qiu J, et al. Twister: a runtime for iterative MapReduce. The first international workshop on MapReduce and its applications (MAPREDUCE'10), HPDC2010. Chicago, Illinois, USA: June 2010.
[100] Moretti C, Bui H, Hollingsworth K, Rich B, Flynn P, Thain D. All-pairs: an abstraction for data-intensive computing on campus grids. IEEE T Parall Distr Syst 2010;21(1):33−46.
[101] Isard M, Budiu M, Yu Y, Birrell A, Fetterly D. Dryad: distributed data-parallel programs from sequential building blocks. European conference on computer systems (EuroSys). Lisbon, Portugal: March 21−23, 2007.
[102] Yu Y, Isard M, Fetterly D, Budiu M, Erlingsson U, Gunda PK, et al. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. Symposium on operating system design and implementation (OSDI). San Diego, CA: December 8−10, 2008.

[103] Calvert C, Kulkarni D. Essential LINQ. Boston, MA, USA: Addison-Wesley Professional; 2009.
[104] Perry DE, Wolf AL. Foundations for the study of software architecture. ACM SIGSOFT Softw Eng Notes 1992;17(4):40−52 ACM Press.
[105] Garlan D, Shaw M. Software architecture: perspectives on an emerging discipline. Upper Saddle River, NJ: Prentice-Hall; 1996.
[106] Gamma E, Helm R, Johnson R, Vlissides JM. ISBN: 0201633612 Design patterns: elements of reusable software design. Boston, MA, USA: Addison-Wesley; 1995.
[107] Box D. A guide to developing and running connected systems with indigo, MSDN magazine. January 2004. Available at: <http://msdn.microsoft.com/en-us/magazine/cc135505.aspx>.
[108] Erl T. Service-oriented architecture: concepts, technology, and design. Upper Saddle River, NJ, USA: Prentice Hall PTR; 2009.
[109] Bell M. Introduction to service-oriented modeling. In: Service-oriented modeling: service analysis, design, and architecture. ISBN: 978-0-470-14111-3, Hoboken, NJ, USA: Wiley & Sons, 2008.
[110] OASIS. Reference architecture foundation for service oriented architecture, version 1.0, October 2009. [Online Document] Available at <http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/soa-ra-cd-02.pdf>.
[111] Nelson BJ. PARC CSL-81-9 Remote procedure call. Palo Alto, CA: Xerox Palo Alto Resource Center; 1981.
[112] Birrell AD, Nelson BJ. Implementing remote procedure calls. ACM T Comput Syst 1984;2(1).
[113] Slee M, Agarwal A, Kwiatkowski M. Thrift: scalable cross-language service implementation. [Online Document] Available at <http://incubator.apache.org/thrift/static/thrift-20070401.pdf>.
[114] Buyya R, Ranjan R, Calheiros RN. InterCloud: utility-oriented federation of cloud computing environments for scaling of application services. Proceedings of the tenth international conference on algorithms and architectures for parallel processing (ICA3PP 2010, LNCS 6081, Springer, Germany). Busan, South Korea: May 21−23, 2010. pp. 13−31.
[115] Beloglazov A, Buyya R, Lee YC, Zomaya A. A taxonomy and survey of energy-efficient data centers and cloud computing systems. In: Zelkowitz M, editor. Advances in computers, 82. Amsterdam, The Netherlands: Elsevier; 2011.
[116] Buyya R, Beloglazov A, Abawajy J. Energy-efficient management of data center resources for cloud computing: a vision, architectural elements, and open challenges. Proceedings of the 2010 international conference on parallel and distributed processing techniques and applications (PDPTA 2010). Las Vegas, NV, USA: July 12−15, 2010.
[117] Garg S, Buyya. R. In: Murugesan S, Gangadharan G, editors. Green cloud computing and environmental sustainability, harnessing green it: principles and practices. West Sussex, UK: Wiley Press; 2011.
[118] Kaplan J, Forrest W, Kindler N. Revolutionizing data center energy efficiency. San Francisco, CA, USA: McKinsey; 2008.
[119] Oppenheimer D, Ganapathi A, Patterson DA. Why do internet services fail, and what can be done about it? Proceedings of the fourth conference on USENIX symposium on internet technologies and systems. vol. 4, Seattle, WA: March 26−28, 2003.
[120] Baliga J, Ayre RWA, Hinton K, Tucker RS. Green cloud computing: balancing energy in processing, storage and transport. Proc IEEE 2011;99(1):149−67.
[121] Kleinrock L. A vision for the Internet. ST J Res 2005;.
[122] Vecchiola C, Duncan D, Buyya R. The structure of new IT Frontier: market oriented cloud computing−part II. Strateg Facil Mag 2010;(Issue 10):59−66 [Pacific & Strategic Holdings Pte Ltd, Singapore]
[123] Buyya R, Yeo CS, Venugopal S, Broberg J, Brandic. I. Cloud computing and emerging it platforms: vision, hype, and reality for delivering computing as the 5th utility. Future Gener Comp Syst 2009;25 (6):599−616.

[124] Garg SK, Buyya R. Cooperative Networking Market-oriented resource management and scheduling: a taxonomy and survey. New York, USA: Wiley Press; 2011.
[125] Smith. R. The contract-net protocol: high-level communication and control in a distributed problem solver. IEEE Trans Comput 1980;4:1104−13.
[126] Fu Y, Chase J, Chun B, Schwab S, Vahdat. A. SHARP: an architecture for secure resource peering. ACM SIGOPS Oper Syst Rev 2003;37(5):133−48.
[127] Lai K, Rasmusson L, Adar E, Zhang L, Huberman BA. Tycoon: an implementation of a distributed, market-based resource allocation system. Multiagent Grid Syst 2005;1(3):169−82.
[128] AuYoung A, Chun B, Snoeren A, Vahdat A. Resource allocation in federated distributed computing infrastructures. In: Proceedings first workshop on operating system and architectural support for the on-demand IT infrastructure. Boston, MA, USA: October 2004.
[129] Irwin DE, Chase JS, Grit LE, Yumerefendi AR, Becker D, Yocum K. Sharing networked resources with brokered leases. Proceedings of 2006 USENIX annual technical conference, USENIX 2006. Boston, MA, USA: June 2006.
[130] Mattess M, Vecchiola C, Buyya R. Managing peak loads by leasing cloud infrastructure services from a spot market. 12th IEEE international conference on high performance computing and communications (HPCC 2010). Melbourne, Australia: 2010.
[131] Buyya R, Pandey S, Vecchiola C. Cloudbus toolkit for market oriented cloud computing. Proceeding of the first international conference on cloud computing (CloudCom 2009, Springer, Germany). Beijing, China: December 1−4, 2009.
[132] Calheiros R, Ranjan R, Beloglazov A, De Rose C, Buyya R. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw Prac Exper 2011;41(1):23−50 Wiley Press, New York, NY, USA.
[133] Beloglazov A, Buyya R. Optimal Online Deterministic Algorithms and Adaptive Heuristics for Energy and Performance Efficient Dynamic Consolidation of Virtual Machines in Cloud Data Centers. Concurrency Comput Prac Exper 2012;24(13):1397−420 Wiley Press, New York, NY, USA.
[134] Brown R, Masanet E, Nordman B, Tschudi B, Shehabi A, Stanley J, et al. Report to congress on server and data center energy efficiency: Public law 109−431. Berkeley, CA, USA: Lawrence Berkeley National Laboratory; 2008.
[135] Barroso LA Fan X. Power provisioning of warehouse-sized computer. Proceedings of the 34th annual symposium on computer architecture.San Diego, CA, USA: 2007. pp. 13−23.
[136] Cloud Storage Technical Working Group. Cloud data management interface (CDMI) v1.0. Storage Network Industry Association (SNIA); 2010. [Online Document]. Available at <www.snia.org/tech_activities/standards/curr_standards/cdmi/CDMI_SNIA_Architecture_v1.0.pdf>.
[137] Open Cloud Standard Incubator. Interoperable clouds: a white paper from the open cloud standards incubator, DSP-IS0101. Distributed Management Task Force (DMTF); November 2009. [Online Document] Available at <www.dmtf.org/sites/default/files/standards/documents/DSP-IS0101_1.0.0.pdf>.
[138] Open Cloud Standard Incubator. Architecture for managing clouds: a white paper from the open cloud standards incubator, DSP-IS0102. Distributed Management Task Force (DMTF); June 2010. [Online Document] Available at <www.dmtf.org/sites/default/files/standards/documents/DSP-IS0102_1.0.0.pdf>.
[139] Open Cloud Standard Incubator. Use cases and interactions for managing clouds: a white paper from the open cloud standards incubator, DSP-IS0103. Distributed Management Task Force (DMTF); June 2010. [Online Document] Available at <www.dmtf.org/sites/default/files/standards/documents/DSP-IS0103_1.0.0.pdf>.
[140] Bowen JA. Legal issues in cloud computing. Cloud computing: principles and paradigms. New York, NY, USA: Wiley Press; 2011.

[141] Gramm-Leach-Bliley Financial Services Modernization Act (GLB Act), Title V of the Financial Services Modernization Act of 1999, Pub. L. No. 106-102, 113 Stat. 1338, U.S. Government. November 1999.
[142] Federal Trade Commission on Business Compliance with Safeguards Rule, U.S. Government, 2009. [Online Document] Available at www.ftc.gov/bcp/edu/pubs/business/idtheft/bus54.shtm.
[143] Identity theft red flags and address discrepancies under the fair and accurate credit transactions act of 2003; final rule, Federal Trade Commission, U.S. Government, November 2007. Available at www.ftc.gov/os/fedreg/2007/november/071109redflags.pdf.
[144] Health Insurance Technology for Economic and Clinical Health (HITECH) Act, Title III of Division A and Title IV of Division B of the American Recovery and Reinvestment Act (ARRA), Pub. L. 111-5, 123 Stat. 115, U.S. Government, February 2009.
[145] Health Insurance Portability and Accountability Act (HIPAA), Pub. L. No. 104-191, 110 Stat. 1936, U. S. Government, August 1996.
[146] Uniting and Strengthening America by Providing Appropriate Tools Required to Intercept and Obstruct Terrorism Act (USA PATRIOT Act), Pub. L. No. 107-56, 115 Stat. 272, U.S. Government, 2001.
[147] EU Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the Protection of Individuals with Regard to the Processing of Personal Data and on the Free Movement of Such Data, European Commission, October 1995. Available at <http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri = CELEX:31995L0046:en:NOT>.
[148] Personal Data (Privacy) Ordinance, Office of the Privacy Commissioner for Personal Data, Hong Kong, 2009. Available at <www.pcpd.org.hk/english/ordinance/ordfull.html>.
[149] Personal Information Protection and Electronics Document Act, Minister of Justice, Canada, Feburary 2012. Available at <http://laws-lois.justice.gc.ca/PDF/P-8.6.pdf>.
[150] Wu L, Buyya R. Service level agreement (SLA) in utility computing systems. Performance and dependability in service computing: concepts, techniques and research directions. Hershey, PA, USA: IGI Global; 2011.
[151] Bouillet E, Mitra D, Ramakrishnan K. The structure and management of service level agreement in networks. IEEE J Sel Area Comm 2002;20(4):691−9.
[152] Dinesh V, Supporting service level agreements on IP networks. Proceedings of IEEE/IFIP network operations and management symposium, 92(2), New York, USA, 2004.
[153] Jin LJ, Machiraju VA. Technical Report HPL-2002-180 Analysis on service level agreement of web services. Software Technologies Laboratory, HP Laboratory; 2002
[154] Ron S Aliko P. Service level agreements, Internet NG, Internet NG Project. 2001. [Online Document] Available at <http://ing.ctit.utwente.nl/WU2/>.
[155] Rochwerger B, Caceres J, Montero R, Breitgand D, Elmroth E, Galis A, et al. The RESERVOIR model and architecture for open federated cloud computing. IBM Syst J 2009;53(4):1−11.
[156] Mather T, Kumaraswarmy S, Lathif S. Cloud security and privacy: an enterprise perspective on risks and compliance. O'Reilly Media Inc; 2009.
[157] Rittinghouse JW, Ransome JF. Cloud computing implementation, management, and security. CRC Press; 2010.
[158] Broberg J, Buyya R, Tari Z. MetaCDN: harnessing 'storage clouds' for high performance content delivery. J Netw Comp Appl 2009;32(5):1012−22 Elsevier, Amsterdam, The Netherlands.
[159] Buyya R, Pathan M, Vakali A, editors. Content delivery networks. Berlin, Germany: Springer; 2008.
[160] Pandey S, Voorsluys W, Niu S, Khandoker A, Buyya R. An autonomic cloud environment for hosting ECG data analysis services. Future Generation Comput Syst 2012;28(1):147−54 Elsevier Science, Amsterdam, The Netherlands.

[161] Jin C, Gubbi J, Buyya R, Palaniswami M, Jeeva: enterprise grid-enabled web portal for protein secondary structure prediction. Proceedings of the 16th international conference on advanced computing and communication (ADCOM 2008). Chennai, India: December 14−17, 2008.

[162] Vecchiola C, Abedini M, Kirley M, Chu X, Buyya R. Gene expression classification with a novel coevolutionary based learning classifier system on public clouds. Proceedings of the 2010 sixth IEEE international conference on e-Science workshops (IEEE CS Press, USA). Brisbane, Australia: December 7, 2010. pp. 92−97.

[163] Raghavendra K, Akilan A, Ravi N, Kumar KP, Varadan G. Satellite data product generation using aneka cloud. Research demo at the 10th IEEE international symposium on cluster, Cloud, and Grid Computing (CCGrid 2010). Melbourne, Australia: 2010.

[164] Buyya R, Abramson D. The Nimrod/G grid resource broker for economic-based scheduling ISBN: 978-0470287682 In: Buyya R, Bubendorfer K, editors. Market oriented grid and utility computing. Hoboken, NJ, USA: Wiley Press; 2009.

[165] Buyya R, Venugopal S, Chu X, Nadiminti K. System and method for grid and cloud computing, Patent No: 8,230,070. United States Patent and Trademark Office; July 24, 2012. Available at <www.uspto.gov/web/patents/patog/week30/OG/html/1380-4/US08230070-20120724.html>.

This page intentionally left blank

Index

Note: Page numbers followed by "*f*", "*t*" and "*b*" refer to figures, tables and boxes, respectively.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z