CHAPTER

Cloud Programming and Software Environments

6

CHAPTER OUTLINE

Summary **336**
6.1 Features of Cloud and Grid Platforms **336**
6.1.1 Cloud Capabilities and Platform Features 336
6.1.2 Traditional Features Common to Grids and Clouds 336
6.1.3 Data Features and Databases 340
6.1.4 Programming and Runtime Support 341
6.2 Parallel and Distributed Programming Paradigms **343**
6.2.1 Parallel Computing and Programming Paradigms 344
6.2.2 MapReduce, Twister, and Iterative MapReduce 345
6.2.3 Hadoop Library from Apache 355
6.2.4 Dryad and DryadLINQ from Microsoft 359
6.2.5 Sawzall and Pig Latin High-Level Languages 365
6.2.6 Mapping Applications to Parallel and Distributed Systems 368
6.3 Programming Support of Google App Engine **370**
6.3.1 Programming the Google App Engine 370
6.3.2 Google File System (GFS) 373
6.3.3 BigTable, Google's NOSQL System 376
6.3.4 Chubby, Google's Distributed Lock Service 379
6.4 Programming on Amazon AWS and Microsoft Azure **379**
6.4.1 Programming on Amazon EC2 380
6.4.2 Amazon Simple Storage Service (S3) 382
6.4.3 Amazon Elastic Block Store (EBS) and SimpleDB 383
6.4.4 Microsoft Azure Programming Support 384
6.5 Emerging Cloud Software Environments **387**
6.5.1 Open Source Eucalyptus and Nimbus 387
6.5.2 OpenNebula, Sector/Sphere, and OpenStack 389
6.5.3 Manjrasoft Aneka Cloud and Appliances 393
6.6 Bibliographic Notes and Homework Problems **399**
Acknowledgement **399**
References **399**
Homework Problems **405**

SUMMARY

This chapter is devoted to programming real cloud platforms. MapReduce, BigTable, Twister, Dryad, DryadLINQ, Hadoop, Sawzall, and Pig Latin are introduced and assessed. We use concrete examples to explain the implementation and application requirements in the cloud. We review core service models and access technologies. Cloud services provided by Google App Engine (GAE), Amazon Web Service (AWS), and Microsoft Windows Azure are illustrated by example applications. In particular, we illustrate how to programming the GAE, AWS EC2, S3, and EBS. We review the open-source Eucalyptus, Nimbus, and OpenNebula and the startup Manjrasoft Aneka system for cloud computing.

6.1 FEATURES OF CLOUD AND GRID PLATFORMS

In this section, we summarize important features in real cloud and grid platforms. In four tables, we cover the capabilities, traditional features, data features, and features for programmers and runtime systems to use. The entries in these tables are source references for anyone who wants to program the cloud efficiently. To get the most from this chapter, readers should refresh their familiarity with and understanding of the languages and software tools for service-oriented architecture (SOA) and web services introduced in Chapter 5.

6.1.1 Cloud Capabilities and Platform Features

Commercial clouds need broad capabilities, as summarized in Table 6.1. These capabilities offer cost-effective utility computing with the elasticity to scale up and down in power. However, as well as this key distinguishing feature, commercial clouds offer a growing number of additional capabilities commonly termed "Platform as a Service" (PaaS). For Azure, current platform features include Azure Table, queues, blobs, Database SQL, and web and Worker roles. Amazon is often viewed as offering "just" Infrastructure as a Service (IaaS), but it continues to add platform features including SimpleDB (similar to Azure Table), queues, notification, monitoring, content delivery network, relational database, and MapReduce (Hadoop). Google does not currently offer a broad-based cloud service, but the Google App Engine (GAE) offers a powerful web application development environment.

Table 6.2 lists some low-level infrastructure features. Table 6.3 lists traditional programming environments for parallel and distributed systems that need to be supported in Cloud environments. They can be supplied as part of system (Cloud Platform) or user environment. Table 6.4 presents features emphasized by clouds and by some grids. Note that some of the features in Table 6.4 have only recently been offered in a major way. In particular, these features are not offered on academic cloud infrastructures such as Eucalyptus, Nimbus, OpenNebula, or Sector/Sphere (although Sector is a data parallel file system or DPFS classified in Table 6.4). We will cover these emerging cloud programming environments in Section 6.5.

6.1.2 Traditional Features Common to Grids and Clouds

In this section, we concentrate on features related to workflow, data transport, security, and availability concerns that are common to today's computing grids and clouds.

Table 6.1 Important Cloud Platform Capabilities

Capability	Description
Physical or virtual computing platform	The cloud environment consists of some physical or virtual platforms. Virtual platforms have unique capabilities to provide isolated environments for different applications and users.
Massive data storage service, distributed file system	With large data sets, cloud data storage services provide large disk capacity and the service interfaces that allow users to put and get data. The distributed file system offers massive data storage service. It can provide similar interfaces as local file systems.
Massive database storage service	Some distributed file systems are sufficient to provide the underlying storage service application developers need to save data in a more semantic way. Just like DBMS in the traditional software stack, massive database storage services are needed in the cloud.
Massive data processing method and programming model	Cloud infrastructure provides thousands of computing nodes for even a very simple application. Programmers need to be able to harness the power of these machines without considering tedious infrastructure management issues such as handling network failure or scaling the running code to use all the computing facilities provided by the platforms.
Workflow and data query language support	The programming model offers abstraction of the cloud infrastructure. Similar to the SQL language used for database systems, in cloud computing, providers have built some workflow language as well as data query language to support better application logic.
Programming interface and service deployment	Web interfaces or special APIs are required for cloud applications: J2EE, PHP, ASP, or Rails. Cloud applications can use Ajax technologies to improve the user experience while using web browsers to access the functions provided. Each cloud provider opens its programming interface for accessing the data stored in massive storage.
Runtime support	Runtime support is transparent to users and their applications. Support includes distributed monitoring services, a distributed task scheduler, as well as distributed locking and other services. They are critical in running cloud applications.
Support services	Important support services include data and computing services. For example, clouds offer rich data services and interesting data parallel execution models like MapReduce.

6.1.2.1 Workflow

As introduced in Section 5.5, workflow has spawned many projects in the United States and Europe. Pegasus, Taverna, and Kepler are popular, but no choice has gained wide acceptance. There are commercial systems such as Pipeline Pilot, AVS (dated), and the LIMS environments. A recent entry is Trident [2] from Microsoft Research which is built on top of Windows Workflow Foundation. If Trident runs on Azure or just any old Windows machine, it will run workflow proxy services on external (Linux) environments. Workflow links multiple cloud and noncloud services in real applications on demand.

6.1.2.2 Data Transport

The cost (in time and money) of data transport in (and to a lesser extent, out of) commercial clouds is often discussed as a difficulty in using clouds. If commercial clouds become an important component of

Table 6.2 Infrastructure Cloud Features

Accounting: Includes economies; clearly an active area for commercial clouds

Appliances: Preconfigured virtual machine (VM) image supporting multifaceted tasks such as message-passing interface (MPI) clusters

Authentication and authorization: Could need single sign-on to multiple systems

Data transport: Transports data between job components both between and within grids and clouds; exploits custom storage patterns as in BitTorrent

Operating systems: Apple, Android, Linux, Windows

Program library: Stores images and other program material

Registry: Information resource for system (system version of metadata management)

Security: Security features other than basic authentication and authorization; includes higher level concepts such as trust

Scheduling: Basic staple of Condor, Platform, Oracle Grid Engine, etc.; clouds have this implicitly as is especially clear with Azure Worker Role

Gang scheduling: Assigns multiple (data-parallel) tasks in a scalable fashion; note that this is provided automatically by MapReduce

Software as a Service (SaaS): Shared between clouds and grids, and can be supported without special attention; Note use of services and corresponding service oriented architectures are very successful and are used in clouds very similarly to previous distributed systems.

Virtualization: Basic feature of clouds supporting "elastic" feature highlighted by Berkeley as characteristic of what defines a (public) cloud; includes virtual networking as in ViNe from University of Florida

Table 6.3 Traditional Features in Cluster, Grid, and Parallel Computing Environments

Cluster management: ROCKS and packages offering a range of tools to make it easy to bring up clusters

Data management: Included metadata support such as RDF triple stores (Semantic web success and can be built on MapReduce as in SHARD); SQL and NOSQL included in

Grid programming environment: Varies from link-together services as in Open Grid Services Architecture (OGSA) to GridRPC (Ninf, GridSolve) and SAGA

OpenMP/threading: Can include parallel compilers such as Cilk; roughly shared memory technologies. Even transactional memory and fine-grained data flow come here

Portals: Can be called (science) gateways and see an interesting change in technology from portlets to HUBzero and now in the cloud: Azure Web Roles and GAE

Scalable parallel computing environments: MPI and associated higher level concepts including ill-fated HP FORTRAN, PGAS (not successful but not disgraced), HPCS languages (X-10, Fortress, Chapel), patterns (including Berkeley dwarves), and functional languages such as F# for distributed memory

Virtual organizations: From specialized grid solutions to popular Web 2.0 capabilities such as Facebook

Workflow: Supports workflows that link job components either within or between grids and clouds; relate to LIMS Laboratory Information Management Systems.

Table 6.4 Platform Features Supported by Clouds and (Sometimes) Grids

Blob: Basic storage concept typified by Azure Blob and Amazon S3

DPFS: Support of file systems such as Google (MapReduce), HDFS (Hadoop), and Cosmos (Dryad) with compute-data affinity optimized for data processing

Fault tolerance: As reviewed in [1] this was largely ignored in grids, but is a major feature of clouds

MapReduce: Support MapReduce programming model including Hadoop on Linux, Dryad on Windows HPCS, and Twister on Windows and Linux. Include new associated languages such as Sawzall, Pregel, Pig Latin, and LINQ

Monitoring: Many grid solutions such as Inca. Can be based on publish-subscribe

Notification: Basic function of publish-subscribe systems

Programming model: Cloud programming models are built with other platform features and are related to familiar web and grid models

Queues: Queuing system possibly based on publish-subscribe

Scalable synchronization: Apache Zookeeper or Google Chubby. Supports distributed locks and used by BigTable. Not clear if (effectively) used in Azure Table or Amazon SimpleDB

SQL: Relational database

Table: Support of table data structures modeled on Apache Hbase or Amazon SimpleDB/Azure Table. Part of NOSQL movement

Web role: Used in Azure to describe important link to user and can be supported otherwise with a portal framework. This is the main purpose of GAE

Worker role: Implicitly used in both Amazon and grids but was first introduced as a high-level construct by Azure

the national cyberinfrastructure we can expect that high-bandwidth links will be made available between clouds and TeraGrid. The special structure of cloud data with blocks (in Azure blobs) and tables could allow high-performance parallel algorithms, but initially, simple HTTP mechanisms are used to transport data [3–5] on academic systems/TeraGrid and commercial clouds.

6.1.2.3 Security, Privacy, and Availability

The following techniques are related to security, privacy, and availability requirements for developing a healthy and dependable cloud programming environment. We summarize these techniques here. Some of these issues have been discussed in Section 4.4.6 with plausible solutions.

- Use virtual clustering to achieve dynamic resource provisioning with minimum overhead cost.
- Use stable and persistent data storage with fast queries for information retrieval.
- Use special APIs for authenticating users and sending e-mail using commercial accounts.
- Cloud resources are accessed with security protocols such as HTTPS and SSL.
- Fine-grained access control is desired to protect data integrity and deter intruders or hackers.
- Shared data sets are protected from malicious alteration, deletion, or copyright violations.
- Features are included for availability enhancement and disaster recovery with life migration of VMs.
- Use a reputation system to protect data centers. This system only authorizes trusted clients and stops pirates.

6.1.3 Data Features and Databases

In the following paragraphs, we review interesting programming features related to the program library, blobs, drives, DPFS, tables, and various types of databases including SQL, NOSQL, and nonrelational databases and special queuing services.

6.1.3.1 Program Library

Many efforts have been made to design a VM image library to manage images used in academic and commercial clouds. The basic cloud environments described in this chapter also include many management features allowing convenient deployment and configuring of images (i.e., they support IaaS).

6.1.3.2 Blobs and Drives

The basic storage concept in clouds is blobs for Azure and S3 for Amazon. These can be organized (approximately, as in directories) by containers in Azure. In addition to a service interface for blobs and S3, one can attach "directly" to compute instances as Azure drives and the Elastic Block Store for Amazon. This concept is similar to shared file systems such as Lustre used in TeraGrid. The cloud storage is intrinsically fault-tolerant while that on TeraGrid needs backup storage. However, the architecture ideas are similar between clouds and TeraGrid, and the Simple Cloud File Storage API [6] could become important here.

6.1.3.3 DPFS

This covers the support of file systems such as Google File System (MapReduce), HDFS (Hadoop), and Cosmos (Dryad) with compute-data affinity optimized for data processing. It could be possible to link DPFS to basic blob and drive-based architecture, but it's simpler to use DPFS as an application-centric storage model with compute-data affinity and blobs and drives as the repository-centric view.

In general, data transport will be needed to link these two data views. It seems important to consider this carefully, as DPFS file systems are precisely designed for efficient execution of data-intensive applications. However, the importance of DPFS for linkage with Amazon and Azure is not clear, as these clouds do not currently offer fine-grained support for compute-data affinity. We note here that Azure Affinity Groups are one interesting capability [7]. We expect that initially blobs, drives, tables, and queues will be the areas where academic systems will most usefully provide a platform similar to Azure (and Amazon). Note the HDFS (Apache) and Sector (UIC) projects in this area.

6.1.3.4 SQL and Relational Databases

Both Amazon and Azure clouds offer relational databases and it is straightforward for academic systems to offer a similar capability unless there are issues of huge scale where, in fact, approaches based on tables and/or MapReduce might be more appropriate [8]. As one early user, we are developing on FutureGrid a new private cloud computing model for the Observational Medical Outcomes Partnership (OMOP) for patient-related medical data which uses Oracle and SAS where FutureGrid is adding Hadoop for scaling to many different analysis methods.

Note that databases can be used to illustrate two approaches to deploying capabilities. Traditionally, one would add database software to that found on computer disks. This software is executed, providing your database instance. However, on Azure and Amazon, the database is installed on a separate VM independent from your job (worker roles in Azure). This implements "SQL as a Service." It may have some performance issues from the messaging interface, but the "aaS" deployment clearly simplifies one's system. For N platform features, one only needs N services, whereas number of possible images with alternative approaches is a prohibitive 2^N.

6.1.3.5 Table and NOSQL Nonrelational Databases

A substantial number of important developments have occurred regarding simplified database structures—termed "NOSQL" [9,10]—typically emphasizing distribution and scalability. These are present in the three major clouds: BigTable [11] in Google, SimpleDB [12] in Amazon, and Azure Table [13] for Azure. Tables are clearly important in science as illustrated by the VOTable standard in astronomy [14] and the popularity of Excel. However, there does not appear to be substantial experience in using tables outside clouds.

There are, of course, many important uses of nonrelational databases, especially in terms of triple stores for metadata storage and access. Recently, there has been interest in building scalable RDF triple stores based on MapReduce and tables or the Hadoop File System [8,15], with early success reported on very large stores. The current cloud tables fall into two groups: Azure Table and Amazon SimpleDB are quite similar [16] and support lightweight storage for "document stores," while BigTable aims to manage large distributed data sets without size limitations.

All these tables are schema-free (each record can have different properties), although BigTable has a schema for column (property) families. It seems likely that tables will grow in importance for scientific computing, and academic systems could support this using two Apache projects: Hbase [17] for BigTable and CouchDB [18] for a document store. Another possibility is the open source SimpleDB implementation M/DB [19]. The new Simple Cloud APIs [6] for file storage, document storage services, and simple queues could help in providing a common environment between academic and commercial clouds.

6.1.3.6 Queuing Services

Both Amazon and Azure offer similar scalable, robust queuing services that are used to communicate between the components of an application. The messages are short (less than 8 KB) and have a Representational State Transfer (REST) service interface with "deliver at least once" semantics. They are controlled by timeouts for posting the length of time allowed for a client to process. One can build a similar approach (on the typically smaller and less challenging academic environments), basing it on publish-subscribe systems such as ActiveMQ [20] or NaradaBrokering [21,22] with which we have substantial experience.

6.1.4 Programming and Runtime Support

Programming and runtime support are desired to facilitate parallel programming and provide runtime support of important functions in today's grids and clouds. Various MapReduce systems are reviewed in this section.

6.1.4.1 Worker and Web Roles

The roles introduced by Azure provide nontrivial functionality, while preserving the better affinity support that is possible in a nonvirtualized environment. Worker roles are basic schedulable processes and are automatically launched. Note that explicit scheduling is unnecessary in clouds for individual worker roles and for the "gang scheduling" supported transparently in MapReduce. Queues are a critical concept here, as they provide a natural way to manage task assignment in a fault–tolerant, distributed fashion. Web roles provide an interesting approach to portals. GAE is largely aimed at web applications, whereas science gateways are successful in TeraGrid.

6.1.4.2 MapReduce

There has been substantial interest in "data parallel" languages largely aimed at loosely coupled computations which execute over different data samples. The language and runtime generate and provide efficient execution of "many task" problems that are well known as successful grid applications. However, MapReduce, summarized in Table 6.5, has several advantages over

Table 6.5 Comparison of MapReduce Type Systems

	Google MapReduce [28]	**Apache Hadoop** [23]	**Microsoft Dryad** [26]	**Twister** [29]	**Azure Twister** [30]
Programming Model	MapReduce	MapReduce	DAG execution, extensible to MapReduce and other patterns	Iterative MapReduce	Currently just MapReduce; will extend to Iterative MapReduce
Data Handling	GFS (Google File System)	HDFS (Hadoop Distributed File System)	Shared directories and local disks	Local disks and data management tools	Azure blob storage
Scheduling	Data locality	Data locality; rack-aware, dynamic task scheduling using global queue	Data locality; network topology optimized at runtime; static task partitions	Data locality; static task partitions	Dynamic task scheduling through global queue
Failure Handling	Reexecution of failed tasks; duplicated execution of slow tasks	Reexecution of failed tasks; duplicated execution of slow tasks	Reexecution of failed tasks; duplicated execution of slow tasks	Reexecution of iterations	Reexecution of failed tasks; duplicated execution of slow tasks
HLL Support	Sawzall [31]	Pig Latin [32,33]	DryadLINQ [27]	Pregel [34] has related features	N/A
Environment	Linux cluster	Linux clusters, Amazon Elastic MapReduce on EC2	Windows HPCS cluster	Linux cluster, EC2	Windows Azure, Azure Local Development Fabric
Intermediate Data Transfer	File	File, HTTP	File, TCP pipes, shared-memory FIFOs	Publish-subscribe messaging	Files, TCP

traditional implementations for many task problems, as it supports dynamic execution, strong fault tolerance, and an easy-to-use high-level interface. The major open source/commercial MapReduce implementations are Hadoop [23] and Dryad [24–27] with execution possible with or without VMs.

Hadoop is currently offered by Amazon, and we expect Dryad to be available on Azure. A prototype Azure MapReduce was built at Indiana University, which we will discuss shortly. On FutureGrid, we already intend to support Hadoop, Dryad, and other MapReduce approaches, including Twister [29] support for iterative computations seen in many data-mining and linear algebra applications. Note that this approach has some similarities with Cloudera [35] which offers a variety of Hadoop distributions including Amazon and Linux. MapReduce is closer to broad deployment than other cloud platform features, as there is quite a bit of experience with Hadoop and Dryad outside clouds.

6.1.4.3 Cloud Programming Models

In many ways, most of the previous sections describe programming model features, but these are "macroscopic" constructs and do not address, for example, the coding (language and libraries). Both the GAE and Manjrasoft Aneka environments represent programming models; both are applied to clouds, but are really not specific to this architecture. Iterative MapReduce is an interesting programming model that offers portability between cloud, HPC and cluster environments.

6.1.4.4 SaaS

Services are used in a similar fashion in commercial clouds and most modern distributed systems. We expect users to package their programs wherever possible, so no special support is needed to enable SaaS. We already discussed in Section 6.1.3 why "Systems software as a service" was an interesting idea in the context of a database service. We desire a SaaS environment that provides many useful tools to develop cloud applications over large data sets. In addition to the technical features, such as MapReduce, BigTable, EC2, S3, Hadoop, AWS, GAE, and WebSphere2, we need protection features that may help us to achieve scalability, security, privacy, and availability.

6.2 PARALLEL AND DISTRIBUTED PROGRAMMING PARADIGMS

We define a parallel and distributed program as a parallel program running on a set of computing engines or a distributed computing system. The term carries the notion of two fundamental terms in computer science: distributed computing system and parallel computing. A distributed computing system is a set of computational engines connected by a network to achieve a common goal of running a job or an application. A computer cluster or network of workstations is an example of a distributed computing system. Parallel computing is the simultaneous use of more than one computational engine (not necessarily connected via a network) to run a job or an application. For instance, parallel computing may use either a distributed or a nondistributed computing system such as a multiprocessor platform.

Running a parallel program on a distributed computing system (parallel and distributed programming) has several advantages for both users and distributed computing systems. From the users' perspective, it decreases application response time; from the distributed computing systems' standpoint, it increases throughput and resource utilization. Running a parallel program on a distributed computing system, however, could be a very complicated process. Therefore, to place the complexity in perspective, data flow of running a typical parallel program on a distributed system is further explained in this chapter.

What are the various system issues for running a typical parallel program in either parallel or distributed manner?

6.2.1 Parallel Computing and Programming Paradigms

Consider a distributed computing system consisting of a set of networked nodes or workers. The system issues for running a typical parallel program in either a parallel or a distributed manner would include the following [36–39]:

- **Partitioning** This is applicable to both computation and data as follows:
- **Computation partitioning** This splits a given job or a program into smaller tasks. Partitioning greatly depends on correctly identifying portions of the job or program that can be performed concurrently. In other words, upon identifying parallelism in the structure of the program, it can be divided into parts to be run on different workers. Different parts may process different data or a copy of the same data.
- **Data partitioning** This splits the input or intermediate data into smaller pieces. Similarly, upon identification of parallelism in the input data, it can also be divided into pieces to be processed on different workers. Data pieces may be processed by different parts of a program or a copy of the same program.
- **Mapping** This assigns the either smaller parts of a program or the smaller pieces of data to underlying resources. This process aims to appropriately assign such parts or pieces to be run simultaneously on different workers and is usually handled by resource allocators in the system.
- **Synchronization** Because different workers may perform different tasks, synchronization and coordination among workers is necessary so that race conditions are prevented and data dependency among different workers is properly managed. Multiple accesses to a shared resource by different workers may raise race conditions, whereas data dependency happens when a worker needs the processed data of other workers.
- **Communication** Because data dependency is one of the main reasons for communication among workers, communication is always triggered when the intermediate data is sent to workers.
- **Scheduling** For a job or program, when the number of computation parts (tasks) or data pieces is more than the number of available workers, a scheduler selects a sequence of tasks or data pieces to be assigned to the workers. It is worth noting that the resource allocator performs the actual mapping of the computation or data pieces to workers, while the scheduler only picks the next part from the queue of unassigned tasks based on a set of rules called the scheduling policy. For multiple jobs or programs, a scheduler selects a sequence of jobs or programs to be run on the distributed computing system. In this case, scheduling is also necessary when system resources are not sufficient to simultaneously run multiple jobs or programs.

6.2.1.1 Motivation for Programming Paradigms

Because handling the whole data flow of parallel and distributed programming is very time-consuming and requires specialized knowledge of programming, dealing with these issues may affect the productivity of the programmer and may even result in affecting the program's time to market. Furthermore, it may detract the programmer from concentrating on the logic of the program itself.

Therefore, parallel and distributed programming paradigms or models are offered to abstract many parts of the data flow from users.

In other words, these models aim to provide users with an abstraction layer to hide implementation details of the data flow which users formerly ought to write codes for. Therefore, simplicity of writing parallel programs is an important metric for parallel and distributed programming paradigms. Other motivations behind parallel and distributed programming models are (1) to improve productivity of programmers, (2) to decrease programs' time to market, (3) to leverage underlying resources more efficiently, (4) to increase system throughput, and (5) to support higher levels of abstraction [40].

MapReduce, Hadoop, and Dryad are three of the most recently proposed parallel and distributed programming models. They were developed for information retrieval applications but have been shown to be applicable for a variety of important applications [41]. Further, the loose coupling of components in these paradigms makes them suitable for VM implementation and leads to much better fault tolerance and scalability for some applications than traditional parallel computing models such as MPI [42–44].

6.2.2 MapReduce, Twister, and Iterative MapReduce

MapReduce, as introduced in Section 6.1.4, is a software framework which supports parallel and distributed computing on large data sets [27,37,45,46]. This software framework abstracts the data flow of running a parallel program on a distributed computing system by providing users with two interfaces in the form of two functions: *Map* and *Reduce*. Users can override these two functions to interact with and manipulate the data flow of running their programs. Figure 6.1 illustrates the logical data flow from the *Map* to the *Reduce* function in MapReduce frameworks. In this framework,

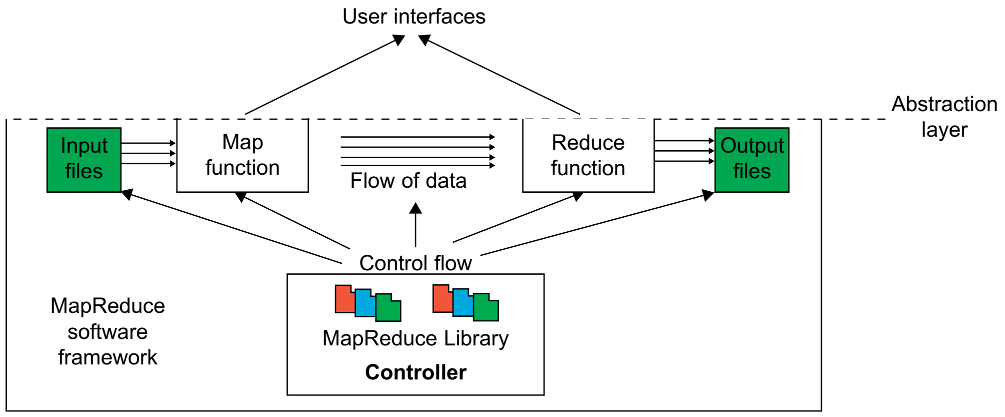


FIGURE 6.1

MapReduce framework: Input data flows through the Map and Reduce functions to generate the output result under the control flow using MapReduce software library. Special user interfaces are used to access the Map and Reduce resources.

the "value" part of the data, *(key, value)*, is the actual data, and the "key" part is only used by the MapReduce controller to control the data flow [37].

6.2.2.1 Formal Definition of MapReduce

The MapReduce software framework provides an abstraction layer with the data flow and flow of control to users, and hides the implementation of all data flow steps such as data partitioning, mapping, synchronization, communication, and scheduling. Here, although the data flow in such frameworks is predefined, the abstraction layer provides two well-defined interfaces in the form of two functions: *Map* and *Reduce* [47]. These two main functions can be overridden by the user to achieve specific objectives. Figure 6.1 shows the MapReduce framework with data flow and control flow.

Therefore, the user overrides the *Map* and *Reduce* functions first and then invokes the provided *MapReduce* (*Spec, & Results*) function from the library to start the flow of data. The MapReduce function, *MapReduce* (*Spec, & Results*), takes an important parameter which is a specification object, the *Spec*. This object is first initialized inside the user's program, and then the user writes code to fill it with the names of input and output files, as well as other optional tuning parameters. This object is also filled with the name of the *Map* and *Reduce* functions to identify these user-defined functions to the MapReduce library.

The overall structure of a user's program containing the Map, Reduce, and the Main functions is given below. The Map and Reduce are two major subroutines. They will be called to implement the desired function performed in the main program.

```
Map Function (.... )
  {
    ... ...
  }
Reduce Function (.... )
  {
    ... ...
  }
Main Function (.... )
  {
    Initialize Spec object
    ... ...
    MapReduce (Spec, & Results)
  }
```

6.2.2.2 MapReduce Logical Data Flow

The input data to both the *Map* and the *Reduce* functions has a particular structure. This also pertains for the output data. The input data to the *Map* function is in the form of a (key, value) pair. For example, the key is the line offset within the input file and the value is the content of the line. The output data from the *Map* function is structured as (key, value) pairs called intermediate (key, value) pairs. In other words, the user-defined *Map* function processes each input (key, value) pair and produces a number of (zero, one, or more) intermediate (key, value) pairs. Here, the goal is to process all input (key, value) pairs to the *Map* function in parallel (Figure 6.2).

In turn, the *Reduce* function receives the intermediate (key, value) pairs in the form of a group of intermediate values associated with one intermediate key, *(key, [set of values])*. In fact, the

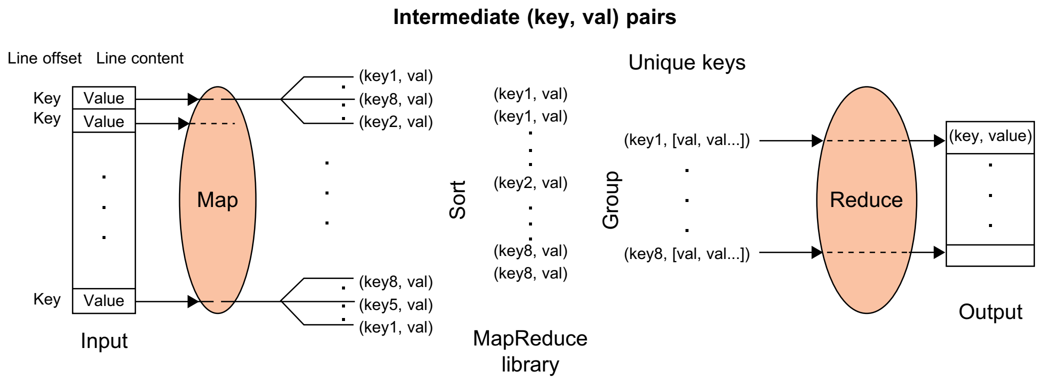


FIGURE 6.2

MapReduce logical data flow in 5 processing stages over successive (key, value) pairs.

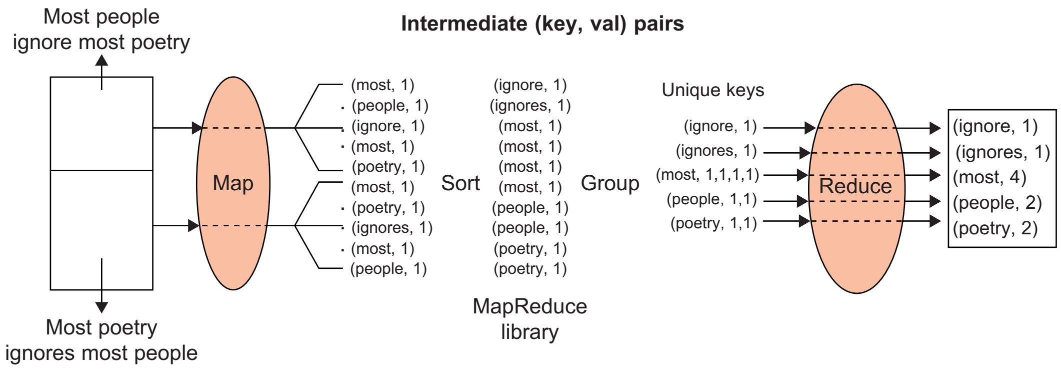


FIGURE 6.3

The data flow of a word-count problem using the MapReduce functions (Map, Sort, Group and Reduce) in a cascade operations.

MapReduce framework forms these groups by first sorting the intermediate (key, value) pairs and then grouping values with the same key. It should be noted that the data is sorted to simplify the grouping process. The *Reduce* function processes each (key, [set of values]) group and produces a set of (key, value) pairs as output.

To clarify the data flow in a sample MapReduce application, one of the well-known MapReduce problems, namely word count, to count the number of occurrences of each word in a collection of documents is presented here. Figure 6.3 demonstrates the data flow of the word-count problem for a simple input file containing only two lines as follows: (1) “most people ignore most poetry” and (2) “most poetry ignores most people.” In this case, the *Map* function simultaneously produces a number of intermediate (key, value) pairs for each line of content so that each word is the intermediate key with *1* as its intermediate value; for example, *(ignore, 1)*. Then the MapReduce library collects all the generated intermediate (key, value) pairs and sorts them to group the *1*'s for identical words; for example, *(people, [1,1])*. Groups are then sent to the *Reduce* function in parallel so that it can sum up the *1* values for each word and generate the actual number of occurrence for each word in the file; for example, *(people, 2)*.

6.2.2.3 Formal Notation of MapReduce Data Flow

The *Map* function is applied in parallel to every input (key, value) pair, and produces new set of intermediate (key, value) pairs [37] as follows:

$$(key_1, val_1) \xrightarrow{\text{Map Function}} \text{List}\,(key_2, val_2) \tag{6.1}$$

Then the MapReduce library collects all the produced intermediate (key, value) pairs from all input (key, value) pairs, and sorts them based on the "key" part. It then groups the values of all occurrences of the same key. Finally, the *Reduce* function is applied in parallel to each group producing the collection of values as output as illustrated here:

$$(key_2, \text{List}\,(val_2)) \xrightarrow{\text{Reduce Function}} \text{List}\,(val_2) \tag{6.2}$$

6.2.2.4 Strategy to Solve MapReduce Problems

As mentioned earlier, after grouping all the intermediate data, the values of all occurrences of the same key are sorted and grouped together. As a result, after grouping, each key becomes unique in all intermediate data. Therefore, finding unique keys is the starting point to solving a typical MapReduce problem. Then the intermediate (key, value) pairs as the output of the *Map* function will be automatically found. The following three examples explain how to define keys and values in such problems:

Problem 1: Counting the number of occurrences of each word in a collection of documents
Solution: unique "key": each word, intermediate "value": number of occurrences
Problem 2: Counting the number of occurrences of words having the same size, or the same number of letters, in a collection of documents
Solution: unique "key": each word, intermediate "value": size of the word
Problem 3: Counting the number of occurrences of anagrams in a collection of documents. Anagrams are words with the same set of letters but in a different order (e.g., the words "listen" and "silent").
Solution: unique "key": alphabetically sorted sequence of letters for each word (e.g., "eilnst"), intermediate "value": number of occurrences

6.2.2.5 MapReduce Actual Data and Control Flow

The main responsibility of the MapReduce framework is to efficiently run a user's program on a distributed computing system. Therefore, the MapReduce framework meticulously handles all partitioning, mapping, synchronization, communication, and scheduling details of such data flows [48,49]. We summarize this in the following distinct steps:

1. **Data partitioning** The MapReduce library splits the input data (files), already stored in GFS, into *M* pieces that also correspond to the number of map tasks.

2. **Computation partitioning** This is implicitly handled (in the MapReduce framework) by obliging users to write their programs in the form of the *Map* and *Reduce* functions. Therefore, the MapReduce library only generates copies of a user program (e.g., by a fork system call) containing the *Map* and the *Reduce* functions, distributes them, and starts them up on a number of available computation engines.
3. **Determining the master and workers** The MapReduce architecture is based on a master-worker model. Therefore, one of the copies of the user program becomes the master and the rest become workers. The master picks idle workers, and assigns the map and reduce tasks to them.
   A map/reduce *worker* is typically a computation engine such as a cluster node to run map/reduce *tasks* by executing *Map/Reduce functions*. Steps 4–7 describe the map workers.
4. **Reading the input data (data distribution)** Each map worker reads its corresponding portion of the input data, namely the input data split, and sends it to its *Map* function. Although a map worker may run more than one *Map* function, which means it has been assigned more than one input data split, each worker is usually assigned one input split only.
5. ***Map* function** Each *Map* function receives the input data split as a set of (key, value) pairs to process and produce the intermediated (key, value) pairs.
6. ***Combiner* function** This is an optional local function within the map worker which applies to intermediate (key, value) pairs. The user can invoke the *Combiner* function inside the user program. The *Combiner* function runs the same code written by users for the *Reduce* function as its functionality is identical to it. The *Combiner* function merges the local data of each map worker before sending it over the network to effectively reduce its communication costs. As mentioned in our discussion of logical data flow, the MapReduce framework sorts and groups the data before it is processed by the *Reduce* function. Similarly, the MapReduce framework will also sort and group the local data on each map worker if the user invokes the *Combiner* function.
7. ***Partitioning* function** As mentioned in our discussion of the MapReduce data flow, the intermediate (key, value) pairs with identical keys are grouped together because all values inside each group should be processed by only one *Reduce* function to generate the final result. However, in real implementations, since there are *M* map and *R* reduce tasks, intermediate (key, value) pairs with the same key might be produced by different map tasks, although they should be grouped and processed together by one *Reduce* function only.
   Therefore, the intermediate (key, value) pairs produced by each map worker are partitioned into *R* regions, equal to the number of reduce tasks, by the *Partitioning* function to guarantee that all (key, value) pairs with identical keys are stored in the same region. As a result, since reduce worker *i* reads the data of region *i* of all map workers, all (key, value) pairs with the same key will be gathered by reduce worker *i* accordingly (see Figure 6.4). To implement this technique, a *Partitioning* function could simply be a hash function (e.g., *Hash*(*key*) *mod R*) that forwards the data into particular regions. It is also worth noting that the locations of the buffered data in these *R* partitions are sent to the master for later forwarding of data to the reduce workers.
   Figure 6.5 shows the data flow implementation of all data flow steps. The following are two networking steps:
8. **Synchronization** MapReduce applies a simple synchronization policy to coordinate map workers with reduce workers, in which the communication between them starts when all map tasks finish.

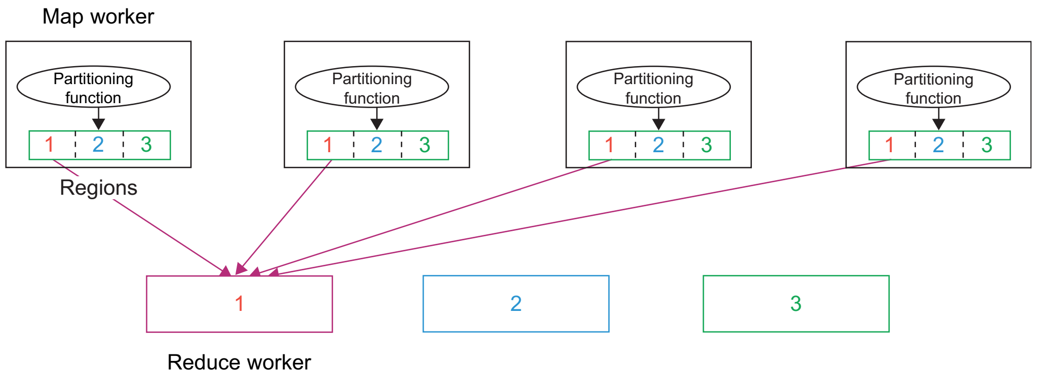


FIGURE 6.4

Use of MapReduce *partitioning* function to link the Map and Reduce workers.

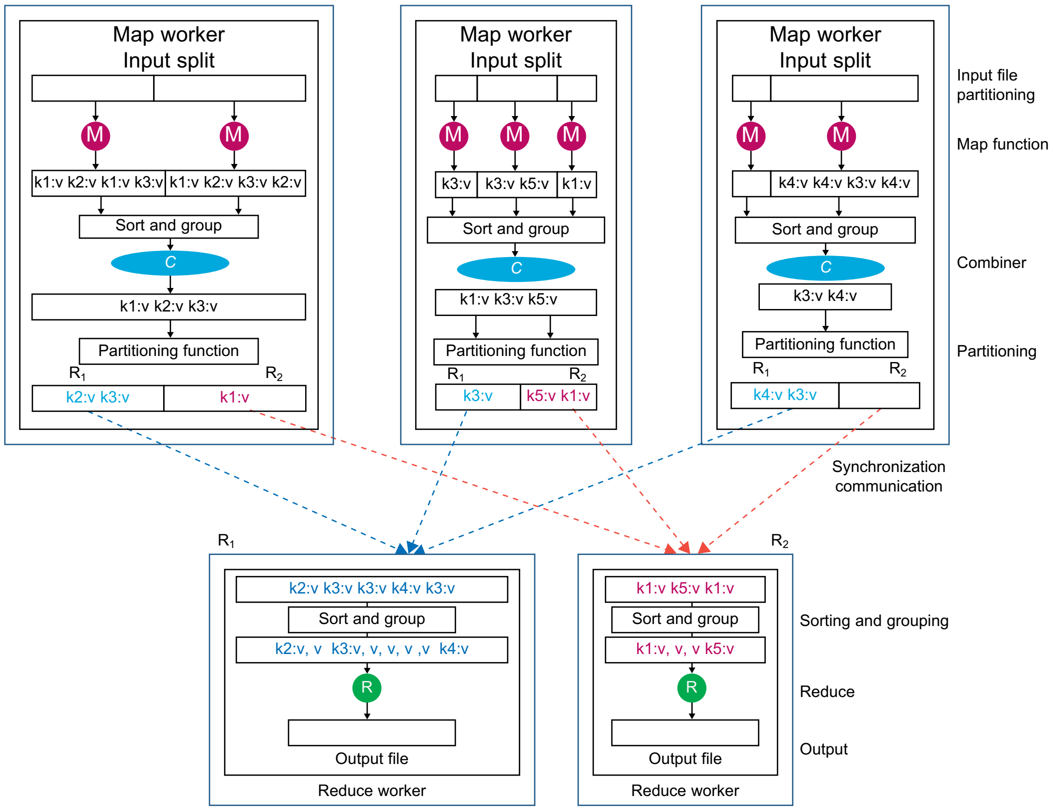


FIGURE 6.5

Data flow implementation of many functions in the Map workers and in the Reduce workers through multiple sequences of partitioning, combining, synchronization and communication, sorting and grouping, and reduce operations.

9. **Communication** Reduce worker *i*, already notified of the location of region *i* of all map workers, uses a remote procedure call to read the data from the respective region of all map workers. Since all reduce workers read the data from all map workers, all-to-all communication among all map and reduce workers, which incurs network congestion, occurs in the network. This issue is one of the major bottlenecks in increasing the performance of such systems [50–52]. A data transfer module was proposed to schedule data transfers independently [55].

   Steps 10 and 11 correspond to the reduce worker domain:
10. **Sorting and Grouping** When the process of reading the input data is finalized by a reduce worker, the data is initially buffered in the local disk of the reduce worker. Then the reduce worker groups intermediate (key, value) pairs by sorting the data based on their keys, followed by grouping all occurrences of identical keys. Note that the buffered data is sorted and grouped because the number of unique keys produced by a map worker may be more than *R* regions in which more than one key exists in each region of a map worker (see Figure 6.4).
11. ***Reduce* function** The reduce worker iterates over the grouped (key, value) pairs, and for each unique key, it sends the key and corresponding values to the *Reduce* function. Then this function processes its input data and stores the output results in predetermined files in the user's program.

To better clarify the interrelated data control and control flow in the MapReduce framework, Figure 6.6 shows the exact order of processing control in such a system contrasting with dataflow in Figure 6.5.

6.2.2.6 Compute-Data Affinity

The MapReduce software framework was first proposed and implemented by Google. The first implementation was coded in C. The implementation takes advantage of GFS [53] as the underlying layer. MapReduce could perfectly adapt itself to GFS. GFS is a distributed file system where files are divided into fixed-size blocks (chunks) and blocks are distributed and stored on cluster nodes.

As stated earlier, the MapReduce library splits the input data (files) into fixed-size blocks, and ideally performs the *Map* function in parallel on each block. In this case, as GFS has already stored files as a set of blocks, the MapReduce framework just needs to send a copy of the user's program containing the *Map* function to the nodes' already stored data blocks. This is the notion of sending computation toward data rather than sending data toward computation. Note that the default GFS block size is 64 MB which is identical to that of the MapReduce framework.

6.2.2.7 Twister and Iterative MapReduce

It is important to understand the performance of different runtimes and, in particular, to compare MPI and MapReduce [43,44,55,56]. The two major sources of parallel overhead are load imbalance and communication (which is equivalent to synchronization overhead as communication synchronizes parallel units [threads or processes] in Categories 2 and 6 of Table 6.10). The communication overhead in MapReduce can be quite high, for two reasons:

- MapReduce reads and writes via files, whereas MPI transfers information directly between nodes over the network.

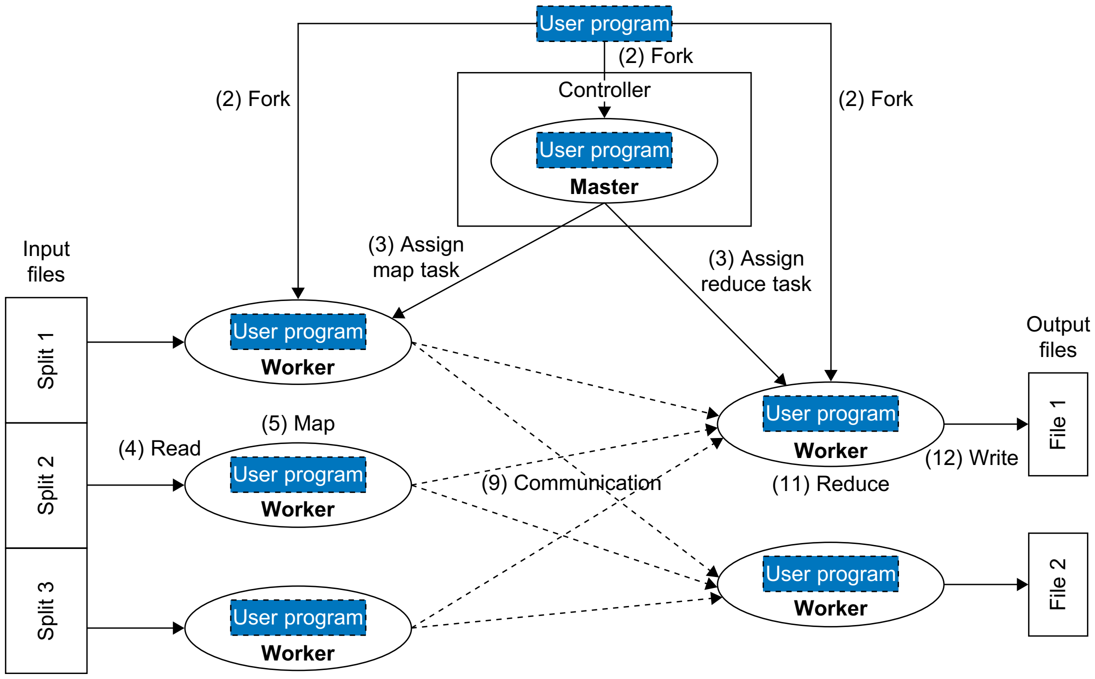


FIGURE 6.6

Control flow implementation of the MapReduce functionalities in Map workers and Reduce workers (running user programs) from input files to the output files under the control of the master user program.

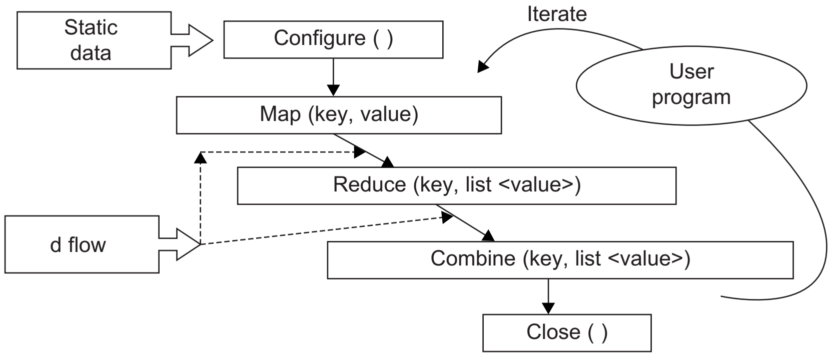
(*Courtesy of Yahoo! Pig Tutorial [54]*)

- MPI does not transfer all data from node to node, but just the amount needed to update information. We can call the MPI flow *δ flow* and the MapReduce flow *full data flow*.

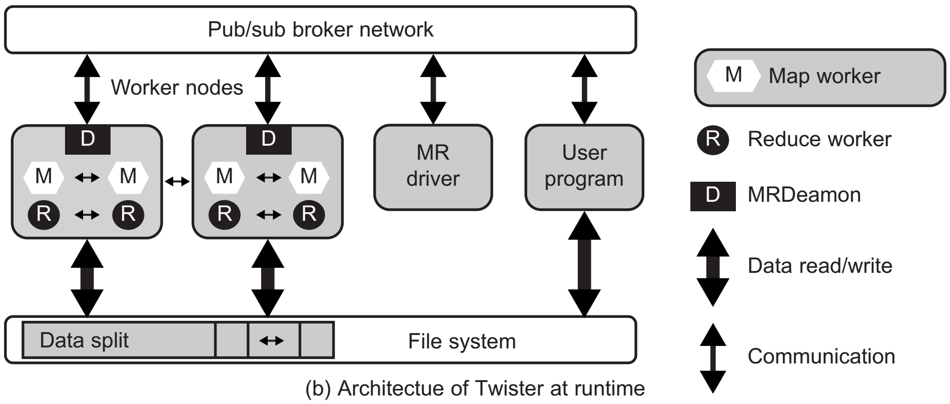
The same phenomenon is seen in all "classic parallel" loosely synchronous applications which typically exhibit an iteration structure over compute phases followed by communication phases. We can address the performance issues with two important changes:

1. Stream information between steps without writing intermediate steps to disk.
2. Use long-running threads or processors to communicate the δ (between iterations) flow.

These changes will lead to major performance increases at the cost of poorer fault tolerance and ease to support dynamic changes such as the number of available nodes. This concept [42] has been investigated in several projects [34,57–59] while the direct idea of using MPI for MapReduce applications is investigated in [44]. The Twister programming paradigm and its implementation architecture at run time are illustrated in Figure 6.7(a, b). In Example 6.1, we summarize Twister [60] whose performance results for K means are shown in Figure 6.8 [55,56], where Twister is much faster than traditional MapReduce. Twister distinguishes the static data which is never reloaded from the dynamic δ flow that is communicated.



(a) Twister for iterative MapReduce programming



(b) Architectue of Twister at runtime

FIGURE 6.7

Twister: An iterative MapReduce programming paradigm for repeated MapReduce executions.

Example 6.1 Performance of K Means Clustering in MPI, Twister, Hadoop, and DryadLINQ

The MapReduce approach leads to fault tolerance and flexible scheduling, but for some applications the performance degradation compared to MPI is serious, as illustrated in Figure 6.8 for a simple parallel K means clustering algorithm. Hadoop and DryadLINQ are more than a factor of 10 slower than MPI for the largest data set, and perform even more poorly for smaller data sets. One could use many communication mechanisms in Iterative MapReduce, but Twister chose a publish-subscribe network using a distributed set of brokers, as described in Section 5.2 with similar performance achieved with ActiveMQ and NaradaBrokering.

The Map-Reduce pair is iteratively executed in long-running threads. We compare in Figure 6.9. the different thread and process structures of 4 parallel programming paradigms: namely Hadoop, Dryad, Twister (also called MapReduce++), and MPI. Note that Dryad can use pipes and avoids costly disk writing according to the original papers [26,27].

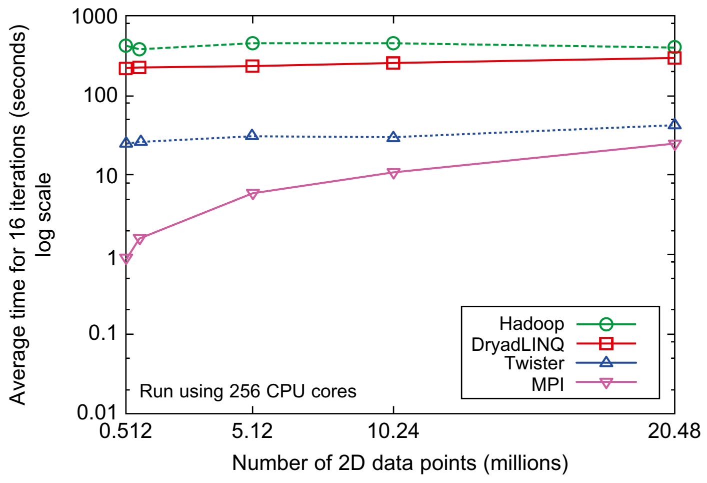


FIGURE 6.8

Performance of K means clustering for MPI, Twister, Hadoop, and DryadLINQ.

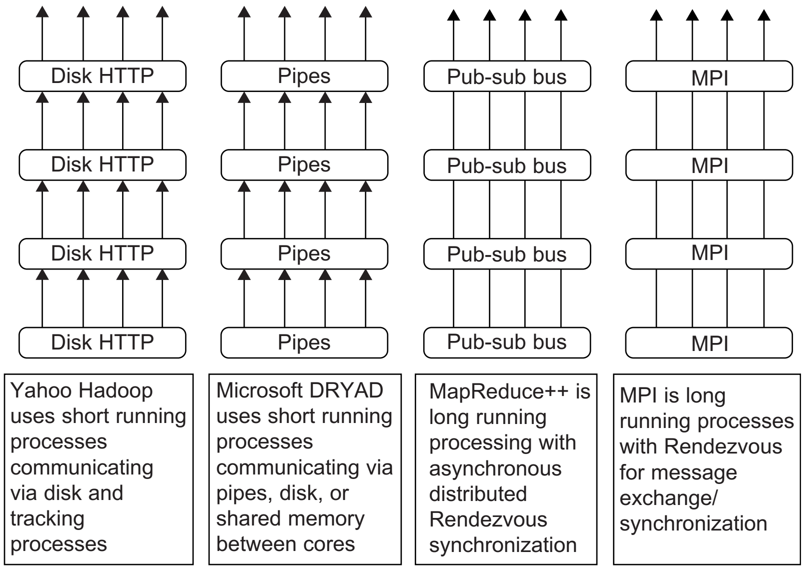

Disk HTTP
Disk HTTP
Disk HTTP
Disk HTTP
Pipes
Pipes
Pipes
Pipes
Pub-sub bus
Pub-sub bus
Pub-sub bus
Pub-sub bus
MPI
MPI
MPI
MPI
Yahoo Hadoop uses short running processes communicating via disk and tracking processes
Microsoft DRYAD uses short running processes communicating via pipes, disk, or shared memory between cores
MapReduce++ is long running processing with asynchronous distributed Rendezvous synchronization
MPI is long running processes with Rendezvous for message exchange/ synchronization

FIGURE 6.9

Thread and process structure of four parallel programming paradigms at runtimes.

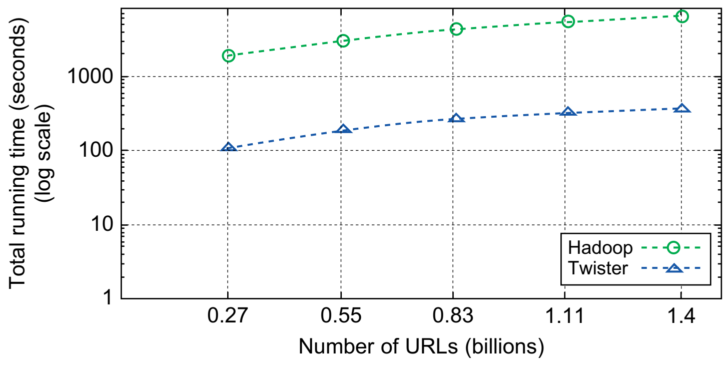


FIGURE 6.10

Performance of Hadoop and Twister on ClueWeb data set using 256 processing cores.

Example 6.2 Performance of Hadoop and Twister on ClueWeb Data Set over 256 Processor Cores

Important research areas for Iterative MapReduce include fault tolerance and scalable approaches to communication. Figure 6.10 shows [55] that iterative algorithms are found in information retrieval. This figure shows the famous Page Rank algorithm (with a kernel of iterative matrix vector multiplication) run on the public ClueWeb data sets, and independent of size, Twister is about 20 times faster than Hadoop as revealed by the gap between the top and lower curves in Figure 6.10.

6.2.3 Hadoop Library from Apache

Hadoop is an open source implementation of MapReduce coded and released in Java (rather than C) by Apache. The Hadoop implementation of MapReduce uses the *Hadoop Distributed File System (HDFS)* as its underlying layer rather than GFS. The Hadoop core is divided into two fundamental layers: the MapReduce engine and HDFS. The MapReduce engine is the computation engine running on top of HDFS as its data storage manager. The following two sections cover the details of these two fundamental layers.

HDFS: HDFS is a distributed file system inspired by GFS that organizes files and stores their data on a distributed computing system.

HDFS Architecture: HDFS has a master/slave architecture containing a single NameNode as the master and a number of DataNodes as workers (slaves). To store a file in this architecture, HDFS splits the file into fixed-size blocks (e.g., 64 MB) and stores them on workers (DataNodes). The mapping of blocks to DataNodes is determined by the NameNode. The NameNode (master) also manages the file system's metadata and namespace. In such systems, the namespace is the area maintaining the metadata, and metadata refers to all the information stored by a file system that is needed for overall management of all files. For example, NameNode in the metadata stores all information regarding the location of input splits/blocks in all DataNodes. Each DataNode, usually one per node in a cluster, manages the storage attached to the node. Each DataNode is responsible for storing and retrieving its file blocks [61].

HDFS Features: Distributed file systems have special requirements, such as performance, scalability, concurrency control, fault tolerance, and security requirements [62], to operate efficiently. However, because HDFS is not a general-purpose file system, as it only executes specific types of applications, it does not need all the requirements of a general distributed file system. For example, security has never been supported for HDFS systems. The following discussion highlights two important characteristics of HDFS to distinguish it from other generic distributed file systems [63].

HDFS Fault Tolerance: One of the main aspects of HDFS is its fault tolerance characteristic. Since Hadoop is designed to be deployed on low-cost hardware by default, a hardware failure in this system is considered to be common rather than an exception. Therefore, Hadoop considers the following issues to fulfill reliability requirements of the file system [64]:

- **Block replication** To reliably store data in HDFS, file blocks are replicated in this system. In other words, HDFS stores a file as a set of blocks and each block is replicated and distributed across the whole cluster. The replication factor is set by the user and is three by default.
- **Replica placement** The placement of replicas is another factor to fulfill the desired fault tolerance in HDFS. Although storing replicas on different nodes (DataNodes) located in different racks across the whole cluster provides more reliability, it is sometimes ignored as the cost of communication between two nodes in different racks is relatively high in comparison with that of different nodes located in the same rack. Therefore, sometimes HDFS compromises its reliability to achieve lower communication costs. For example, for the default replication factor of three, HDFS stores one replica in the same node the original data is stored, one replica on a different node but in the same rack, and one replica on a different node in a different rack to provide three copies of the data [65].
- **Heartbeat and Blockreport messages** Heartbeats and Blockreports are periodic messages sent to the NameNode by each DataNode in a cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly, while each Blockreport contains a list of all blocks on a DataNode [65]. The NameNode receives such messages because it is the sole decision maker of all replicas in the system.

HDFS High-Throughput Access to Large Data Sets (Files): Because HDFS is primarily designed for batch processing rather than interactive processing, data access throughput in HDFS is more important than latency. Also, because applications run on HDFS typically have large data sets, individual files are broken into large blocks (e.g., 64 MB) to allow HDFS to decrease the amount of metadata storage required per file. This provides two advantages: The list of blocks per file will shrink as the size of individual blocks increases, and by keeping large amounts of data sequentially within a block, HDFS provides fast streaming reads of data.

HDFS Operation: The control flow of HDFS operations such as write and read can properly highlight roles of the NameNode and DataNodes in the managing operations. In this section, the control flow of the main operations of HDFS on files is further described to manifest the interaction between the user, the NameNode, and the DataNodes in such systems [63].

- **Reading a file** To read a file in HDFS, a user sends an "open" request to the NameNode to get the location of file blocks. For each file block, the NameNode returns the address of a set of DataNodes containing replica information for the requested file. The number of addresses depends on the number of block replicas. Upon receiving such information, the user calls the *read* function to connect to the closest DataNode containing the first block of the file. After the first block is streamed from the respective DataNode to the user, the established connection is terminated and the same process is repeated for all blocks of the requested file until the whole file is streamed to the user.
- **Writing to a file** To write a file in HDFS, a user sends a "create" request to the NameNode to create a new file in the file system namespace. If the file does not exist, the NameNode notifies the user and allows him to start writing data to the file by calling the *write* function. The first block of the file is written to an internal queue termed the data queue while a data streamer monitors its writing into a DataNode. Since each file block needs to be replicated by a predefined factor, the data streamer first sends a request to the NameNode to get a list of suitable DataNodes to store replicas of the first block.

The steamer then stores the block in the first allocated DataNode. Afterward, the block is forwarded to the second DataNode by the first DataNode. The process continues until all allocated DataNodes receive a replica of the first block from the previous DataNode. Once this replication process is finalized, the same process starts for the second block and continues until all blocks of the file are stored and replicated on the file system.

6.2.3.1 Architecture of MapReduce in Hadoop

The topmost layer of Hadoop is the MapReduce engine that manages the data flow and control flow of MapReduce jobs over distributed computing systems. Figure 6.11 shows the MapReduce engine architecture cooperating with HDFS. Similar to HDFS, the MapReduce engine also

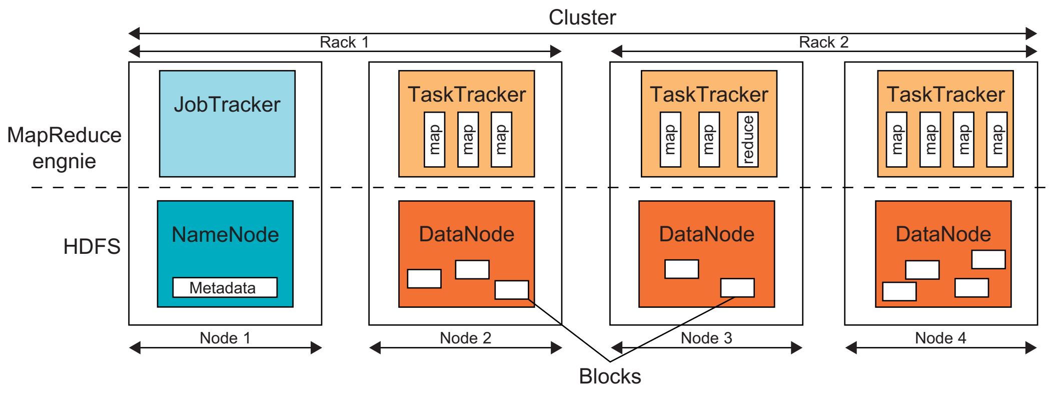


FIGURE 6.11

HDFS and MapReduce architecture in Hadoop where boxes with different shadings refer to different functional nodes applied to different blocks of data.

has a master/slave architecture consisting of a single JobTracker as the master and a number of TaskTrackers as the slaves (workers). The JobTracker manages the MapReduce job over a cluster and is responsible for monitoring jobs and assigning tasks to TaskTrackers. The TaskTracker manages the execution of the map and/or reduce tasks on a single computation node in the cluster.

Each TaskTracker node has a number of simultaneous execution slots, each executing either a map or a reduce task. Slots are defined as the number of simultaneous threads supported by CPUs of the TaskTracker node. For example, a TaskTracker node with N CPUs, each supporting M threads, has $M * N$ simultaneous execution slots [66]. It is worth noting that each data block is processed by one map task running on a single slot. Therefore, there is a one-to-one correspondence between map tasks in a TaskTracker and data blocks in the respective DataNode.

With a neat diagram explaining the data flow in running a MapReduce job at various task trackers using Hadoop Library

6.2.3.2 Running a Job in Hadoop

Three components contribute in running a job in this system: a user node, a JobTracker, and several TaskTrackers. The data flow starts by calling the *runJob(conf)* function inside a user program running on the user node, in which *conf* is an object containing some tuning parameters for the MapReduce framework and HDFS. The *runJob(conf)* function and *conf* are comparable to the *MapReduce(Spec, &Results)* function and *Spec* in the first implementation of MapReduce by Google. Figure 6.12 depicts the data flow of running a MapReduce job in Hadoop [63].

- **Job Submission** Each job is submitted from a user node to the JobTracker node that might be situated in a different node within the cluster through the following procedure:
- A user node asks for a new job ID from the JobTracker and computes input file splits.
- The user node copies some resources, such as the job's JAR file, configuration file, and computed input splits, to the JobTracker's file system.
- The user node submits the job to the JobTracker by calling the *submitJob()* function.
- **Task assignment** The JobTracker creates one map task for each computed input split by the user node and assigns the map tasks to the execution slots of the TaskTrackers. The JobTracker considers the localization of the data when assigning the map tasks to the TaskTrackers. The JobTracker

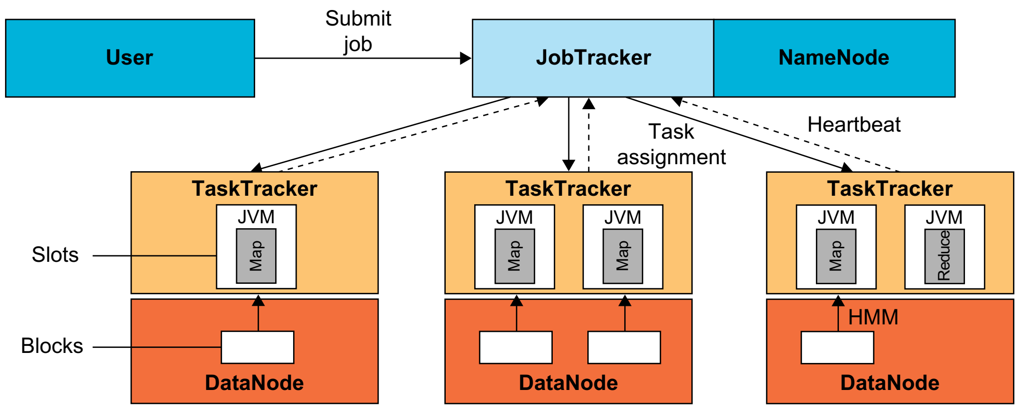


FIGURE 6.12

Data flow in running a MapReduce job at various task trackers using the Hadoop library.

also creates reduce tasks and assigns them to the TaskTrackers. The number of reduce tasks is predetermined by the user, and there is no locality consideration in assigning them.

- **Task execution** The control flow to execute a task (either map or reduce) starts inside the TaskTracker by copying the job JAR file to its file system. Instructions inside the job JAR file are executed after launching a Java Virtual Machine (JVM) to run its map or reduce task.
- **Task running check** A task running check is performed by receiving periodic heartbeat messages to the JobTracker from the TaskTrackers. Each heartbeat notifies the JobTracker that the sending TaskTracker is alive, and whether the sending TaskTracker is ready to run a new task.

6.2.4 Dryad and DryadLINQ from Microsoft

Two runtime software environments are reviewed in this section for parallel and distributed computing, namely the Dryad and DryadLINQ, both developed by Microsoft.

6.2.4.1 Dryad

Dryad is more flexible than MapReduce as the data flow of its applications is not dictated/predetermined and can be easily defined by users. To achieve such flexibility, a Dryad program or job is defined by a *directed acyclic graph* (DAG) where vertices are computation engines and edges are communication channels between vertices. Therefore, users or application developers can easily specify arbitrary DAGs to specify data flows in jobs.

Given a DAG, Dryad assigns the computational vertices to the underlying computation engines (cluster nodes) and controls the data flow through edges (communication between cluster nodes). Data partitioning, scheduling, mapping, synchronization, communication, and fault tolerance are major implementation details hidden by Dryad to facilitate its programming environment. Because the data flow of a job is arbitrary for this system, only the control flow of the runtime environment is further explained here. As shown in Figure 6.13(a), the two main components handling the control flow of Dryad are the job manager and the name server.

In Dryad, the distributed job is represented as a DAG where each vertex is a program and edges represent data channels. Thus, the whole job will be constructed by the application programmer who defines the processing procedures as well as the flow of data. This logical computation graph will be automatically mapped onto the physical nodes by the Dryad runtime. A Dryad job is controlled by the job manager, which is responsible for deploying the program to the multiple nodes in the cluster. It runs either within the computing cluster or as a process in the user's workstation which can access the cluster. The job manager has the code to construct the DAG as well as the library to schedule the work running on top of the available resources. Data transfer is done via channels without involving the job manager. Thus, the job manager should not be the performance bottleneck. In summary, the job manager

1. Constructs a job's communication graph (data flow graph) using the application-specific program provided by the user.
2. Collects the information required to map the data flow graph to the underlying resources (computation engine) from the name server.

The cluster has a name server which is used to enumerate all the available computing resources in the cluster. Thus, the job manager can contact the name server to get the topology of the whole

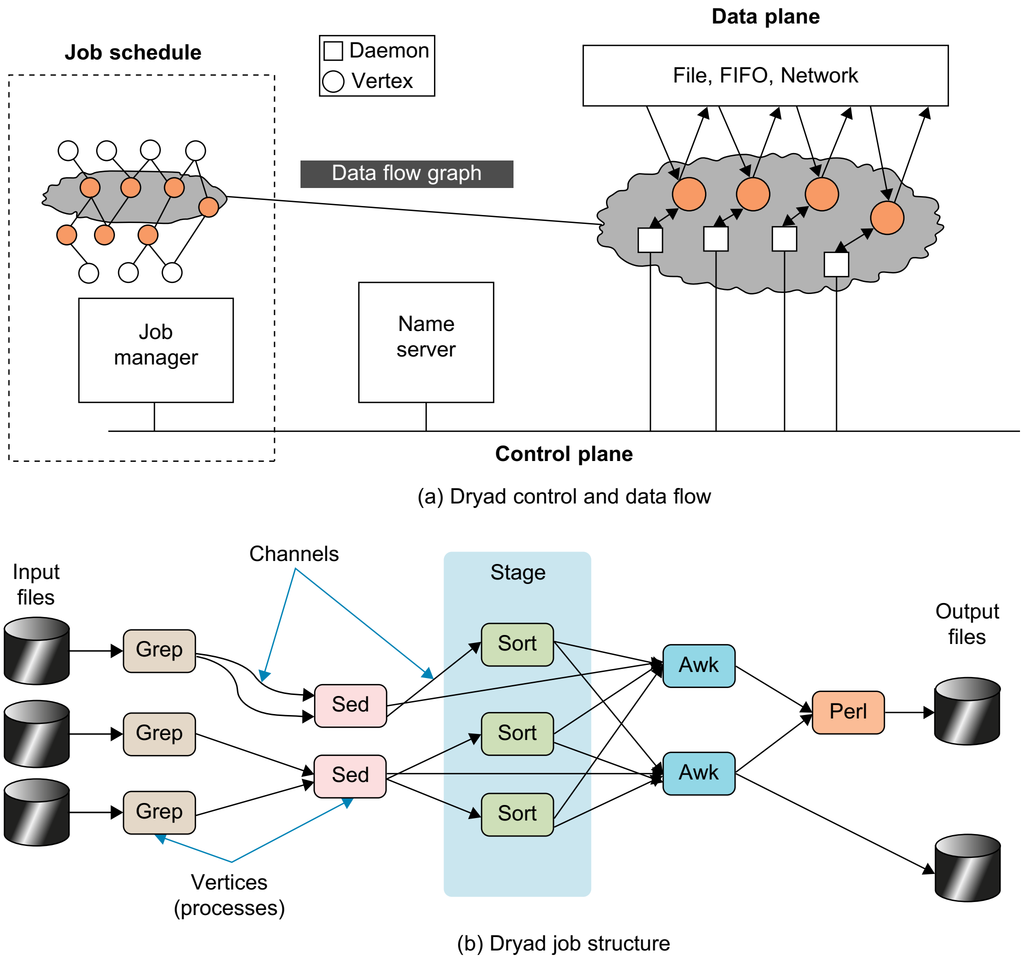


FIGURE 6.13

Dryad framework and its job structure, control and data flow.

(*Courtesy of Isard, et al., ACM SIGOPS Operating Systems Review, 2007* [26])

cluster and make scheduling decisions. A *processing daemon* runs in each computing node in the cluster. The binary of the program will be sent to the corresponding processing node directly from the job manager. The daemon can be viewed as a proxy so that the job manager can communicate with the remote vertices and monitor the state of the computation. By gathering this information, the name server provides the job manager with a perfect view of the underlying resources and network topology. Therefore, the job manager is able to:

1. Map the data flow graph to the underlying resources.
2. Schedule all necessary communications and synchronization *across* the respective resources.

It also considers data and computation locality when mapping the data flow graph to the underlying resources [26]. When the data flow graph is mapped on a set of computation engines, a light daemon runs on each cluster node to run the assigned tasks. Each task is defined by the user using an application-specific program. During runtime, the job manager communicates with each daemon to monitor the state of the computation of the node and its communication with its preceding and succeeding nodes. At runtime, the channels are used to transport the structured items between vertices which represent the processing programs. There are several types of communication mechanisms for implementing channels such as shared memory, TCP sockets, or even distributed file systems.

The execution of a Dryad job can be considered a 2D distributed set of pipes. Traditional UNIX pipes are 1D pipes, with each node in the pipe as a single program. Dryad's 2D distributed pipe system has multiple processing programs in each vertex node. In this way, large-scale data can be processed simultaneously. Figure 6.13(b) shows the Dryad 2D pipe job structure. During 2D pipe execution, Dryad defines many operations to construct and change the DAG dynamically. The operations include creating new vertices, adding graph edges, merging two graphs, as well as handling job input and output. Dryad also has a fault-tolerant mechanism built in. As it is built on a DAG, there are typically two types of failures: vertex failures and channel failures, which are handled differently.

As there are many nodes in the cluster, the job manager can choose another node to re-execute the corresponding job assigned to the failed node. In case of an edge failure, the vertex that created the channel will be re-executed and a new channel will be created and touch the corresponding nodes again. Dryad provides other mechanisms in addition to the runtime graph refinements which are used for improving execution performance. As a general framework, Dryad can be used in many situations, including scripting language support, map-reduce programming, and SQL service integration.

6.2.4.2 DryadLINQ from Microsoft

DryadLINQ is built on top of Microsoft's Dryad execution framework (see http://research.microsoft.com/en-us/projects/DryadLINQ/). Dryad can perform acyclic task scheduling and run on large-scale servers. The goal of DryadLINQ is to make large-scale, distributed cluster computing available to ordinary programmers. Actually, DryadLINQ, as the name implies, combines two important components: the Dryad distributed execution engine and .NET Language Integrated Query (LINQ). LINQ is particularly for users familiar with a database programming model. Figure 6.14 shows the flow of execution with DryadLINQ. The execution is divided into nine steps as follows:

1. A .NET user application runs, and creates a DryadLINQ expression object. Because of LINQ's deferred evaluation, the actual execution of the expression has not occurred.
2. The application calls *ToDryadTable* triggering a data-parallel execution. The expression object is handed to *DryadLINQ*.
3. DryadLINQ compiles the LINQ expression into a distributed Dryad execution plan. The expression is decomposed into subexpressions, each to be run in a separate Dryad vertex. Code and static data for the remote Dryad vertices are generated, followed by the serialization code for the required data types.
4. DryadLINQ invokes a custom Dryad job manager which is used to manage and monitor the execution flow of the corresponding task.
5. The job manager creates the job graph using the plan created in step 3. It schedules and spawns the vertices as resources become available.

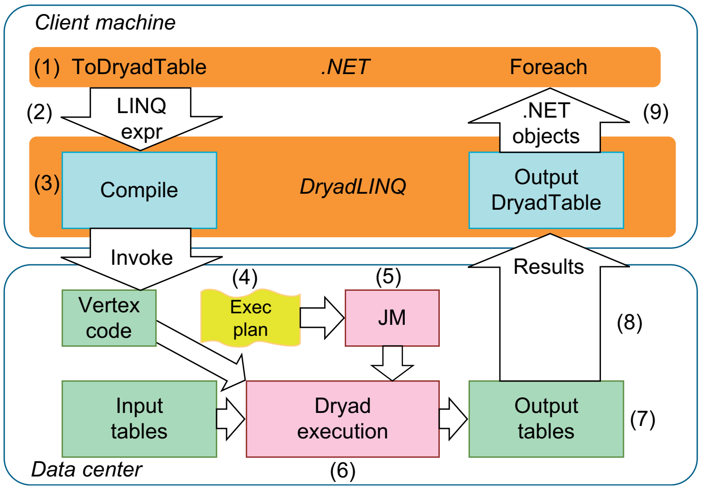


FIGURE 6.14

LINQ-expression execution in DryadLINQ.

(*Courtesy of Yu, et al.* [27])

6. Each Dryad vertex executes a vertex-specific program.
7. When the Dryad job completes successfully it writes the data to the out table(s).
8. The job manager process terminates, and it returns control back to DryadLINQ. DryadLINQ creates the local *DryadTable* objects encapsulating the output of the execution. The *DryadTable* objects here might be the input to the next phase.
9. Control returns to the user application. The iterator interface over a DryadTable allows the user to read its contents as .NET objects.

Not all programs go through all nine steps. Some programs may go through fewer steps. Based on the preceding description, DryadLINQ enables users to integrate their current programming language (C#) with a compiler and a runtime execution engine. The following example shows how to write a histogram in DryadLINQ.

Example 6.3 A Histogram Written in DryadLINQ

We show a histogram program written in DryadLINQ for counting the frequency of each word in a text file. The entire program is listed here for readers to trace through to familiarize with the high-level language for Dryad programming. For details, the reader is referred to the article that introduces by Yu, et al. [27].

```
[Serializable]
public struct Pair {
  string word;
  int count;
```

```
    public Pair(string w, int c)
    {
      word = w;
      count = c;
    }
    public override string ToString() {
      return word + ":" + count.ToString();
    }
  }
  public static IQueryable<Pair> Histogram(
              string directory,
              string filename,
              int k)
  {
     DryadDataContext ddc = new DryadDataContext("file://" + directory);
     DryadTable<LineRecord> table =
               ddc.GetPartitionedTable<LineRecord>(filename);
     IQueryable<string> words =
          table.SelectMany(x => x.line.Split(' ').AsEnumerable());
     IQueryable<IGrouping<string, string>> groups = words.GroupBy(x => x);
     IQueryable<Pair> counts = groups.Select(x => new Pair(x.Key, x.Count()));
     IQueryable<Pair> ordered = counts.OrderByDescending(x => x.Count);
     IQueryable<Pair> top = ordered.Take(k);
     return top;
  }
```

Example 6.4 Sample Execution of a Histogram of the Word-Count Problem

The execution flow of such programs is very similar to the word-count program in the MapReduce framework. Table 6.6 shows how this program can be executed on a sample text input. Each row in the table presents the execution effect of each code line in the program. The execution of the program will go through all of the steps that a typical DryadLINQ program will involve.

Programmers do not have to worry about parallel execution of the program or consider fault tolerance. Scalability, reliability, and some other difficult issues involved in distributed systems are all hidden

Table 6.6 Sample Execution of Histogram

Operator	Output
Table	“A line of words of wisdom”
SelectMany	[“A”, “line”, “of”, “words”, “of”, “wisdom”]
GroupBy	[[“A”], [“line”], [“of”, “of”], [“words”], [“wisdom”]]
Select	[{“A”, 1}, {“line”, 1}, {“of”, 2}, {“words”, 1}, {“wisdom”, 1}]
OrderByDescending	[{“of”, 2}, {“A”, 1}, {“line”, 1}, {“words”, 1}, {“wisdom”, 1}]
Take(3)	[{“of”, 2}, {“A”, 1}, {“line”, 1}]

inside the DryadLINQ framework, and the programs are more concerned with the application program logic. This can greatly reduce the level of programming skill required for parallel data process programming.

Example 6.5 Hadoop Implementation of a MapReduce WebVisCounter Program

In this example, we present a practical MapReduce program coded in Hadoop. Called WebVisCounter, this sample program counts the number of times users connect to or visit a given web site using a particular operating system (e.g., Windows XP or Linux Ubuntu). The input data is shown here. A single line of a typical web server log file has eight fields separated by tabs or spaces with the following meanings:

1. **176.123.143.12** (IP address of the machine connected)
2. **–** (a separator)
3. **[10/Sep/2010:01:11:30-1100]** (A timestamp of the visit in the format: DD:Mon:YYYY HH:MM:SS, and the –11:00 is the offset from Greenwich Mean Time)
4. **"GET /gse/apply/int_research_app_form.pdf HTTP/1.0"** (GET requests the file, /gse/apply/int_research_app_form.pdf, using the HTTP/1.0 protocol)
5. **200** (The status code to reflect success of the user's request)
6. **1363148** (The number of bytes transferred)
7. **"http://www.eng.usyd.edu.au"** (User starts here before reaching server)
8. **"Mozilla/4.7[en](WinXp; U)"** (The browser used to get to the web site, the version of the browser, the language version, and the operating system)

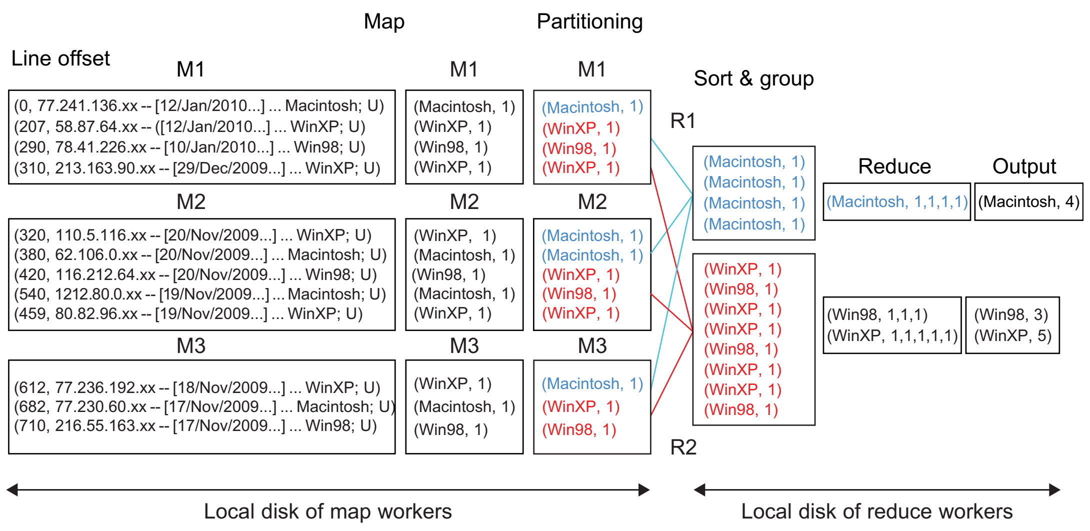

Map
Partitioning
Line offset
M1
M1
M1
Sort & group
(0, 77.241.136.xx -- [12/Jan/2010...] ... Macintosh; U)
(207, 58.87.64.xx -- ([12/Jan/2010...] ... WinXP; U)
(290, 78.41.226.xx -- [10/Jan/2010...] ... Win98; U)
(310, 213.163.90.xx -- [29/Dec/2009...] ... WinXP; U)
(Macintosh, 1)
(WinXP, 1)
(Win98, 1)
(WinXP, 1)
(Macintosh, 1)
(WinXP, 1)
(Win98, 1)
(WinXP, 1)
R1
(Macintosh, 1)
(Macintosh, 1)
(Macintosh, 1)
(Macintosh, 1)
Reduce
Output
(Macintosh, 1,1,1,1)
(Macintosh, 4)
M2
M2
M2
(320, 110.5.116.xx -- [20/Nov/2009...] ... WinXP; U)
(380, 62.106.0.xx -- [20/Nov/2009...] ... Macintosh; U)
(420, 116.212.64.xx -- [20/Nov/2009...] ... Win98; U)
(540, 1212.80.0.xx -- [19/Nov/2009...] ... Macintosh; U)
(459, 80.82.96.xx -- [19/Nov/2009...] ... WinXP; U)
(WinXP, 1)
(Macintosh, 1)
(Win98, 1)
(Macintosh, 1)
(WinXP, 1)
(Macintosh, 1)
(Macintosh, 1)
(WinXP, 1)
(Win98, 1)
(WinXP, 1)
(WinXP, 1)
(Win98, 1)
(WinXP, 1)
(WinXP, 1)
(Win98, 1)
(WinXP, 1)
(WinXP, 1)
(Win98, 1)
(Win98, 1,1,1)
(WinXP, 1,1,1,1,1)
(Win98, 3)
(WinXP, 5)
M3
M3
M3
(612, 77.236.192.xx -- [18/Nov/2009...] ... WinXP; U)
(682, 77.230.60.xx -- [17/Nov/2009...] ... Macintosh; U)
(710, 216.55.163.xx -- [17/Nov/2009...] ... Win98; U)
(WinXP, 1)
(Macintosh, 1)
(Win98, 1)
(Macintosh, 1)
(WinXP, 1)
(Win98, 1)
R2
Local disk of map workers
Local disk of reduce workers

FIGURE 6.15

Data flow in WebVisCounter program execution.

Because the output of interest is the number of times users connect to a given web site using a particular operating system, the *Map* function parses each line to extract the type of the operating system used (e.g.,*WinXp*) as a key and assigns a value (*1* in this case) to it. The *Reduce* function in turn sums up the number of *1*s for each unique key (operating system type in this case). Figure 6.15 shows the associated data flow for the WebVisCounter program.

6.2.5 Sawzall and Pig Latin High-Level Languages

Sawzall is a high-level language [31] built on top of Google's MapReduce framework. Sawzall is a scripting language that can do parallel data processing. As with MapReduce, Sawzall can do distributed, fault-tolerant processing of very large-scale data sets, even at the scale of the data collected from the entire Internet. Sawzall was developed by Rob Pike with an initial goal of processing Google's log files. In this regard it was hugely successful and changed a batch day-long enterprise into an interactive session, enabling new approaches to using such data to be developed. Figure 6.16 shows the overall model of data flow and processing procedures in the Sawzall framework. Sawzall has recently been released as an open source project.

First the data is partitioned and processed locally with an on-site processing script. The local data is filtered to get the necessary information. The aggregator is used to get the final results based on the emitted data. Many of Google's applications fit this model. Users write their applications using the Sawzall scripting language. The Sawzall runtime engine translates the corresponding scripts to MapReduce programs running on many nodes. The Sawzall program can harness the power of cluster computing automatically as well as attain reliability from redundant servers.

Example 6.6 The File Summary Program in Sawzall

Here is a simple example of using Sawzall for data processing on clusters (the example is from the Sawzall paper at [31]). Suppose we want to process a set of files with records in each file. Each record contains one floating-point number. We want to calculate the number of records, the sum of the values, and the sum of the squares of the values. The relevant code is as follows:

```
count: table sum of int;
total: table sum of float;
```

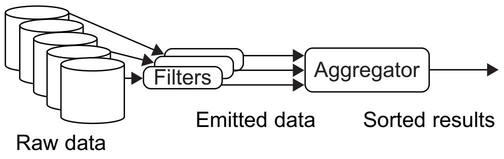


FIGURE 6.16

The overall flow of filtering, aggregating, and collating in Sawzall.

```
sum_of_squares: table sum of float;
x: float = input;
emit count <- 1;
emit total <- x;
emit sum_of_squares <- x * x;
```

The first three lines declare the aggregators as *count*, *total*, and *sum_of_squares*. *table* is a keyword which defines an aggregator type. These particular tables are *sum* tables and they will automatically add up the values emitted to them. For each input record, Sawzall initializes the predefined variable input to the uninterpreted byte string of the input. The line *x: float = input;* converts the input into a floating-point number and stores it in local variable *x*. The three *emit* statements send the intermediate values to the aggregators. One can translate the Sawzall scripts into MapReduce programs and run them with multiple servers.

6.2.5.1 Pig Latin

Pig Latin is a high-level data flow language developed by Yahoo! [33] that has been implemented on top of Hadoop in the Apache Pig project [67]. Pig Latin, Sawzall and DryadLINQ are different approaches to building languages on top of MapReduce and its extensions. They are compared in Table 6.7.

DryadLINQ is building directly on SQL while the other two languages are of NOSQL heritage, although Pig Latin supports major SQL constructs including Join, which is absent in Sawzall. Each language automates the parallelism, so you only think about manipulation of individual elements and then invoke supported collective operations. This is possible, of course, because needed parallelism can be cleanly implemented by independent tasks with "side effects" only presented in

Table 6.7 Comparison of High-Level Data Analysis Languages

	Sawzall	**Pig Latin**	**DryadLINQ**
Origin	Google	Yahoo!	Microsoft
Data Model	Google protocol buffer or basic	Atom, Tuple, Bag, Map	Partition file
Typing	Static	Dynamic	Static
Category	Interpreted	Compiled	Compiled
Programming Style	Imperative	Procedural: sequence of declarative steps	Imperative and declarative
Similarity to SQL	Least	Moderate	A lot!
Extensibility (User-Defined Functions)	No	Yes	Yes
Control Structures	Yes	No	Yes
Execution Model	Record operations + fixed aggregations	Sequence of MapReduce operations	DAGs
Target Runtime	Google MapReduce	Hadoop (Pig)	Dryad

Table 6.8 Pig Latin Data Types

Data Type	Description	Example
Atom	Simple atomic value	'Clouds'
Tuple	Sequence of fields of any Pig Latin type	('Clouds', 'Grids')
Bag	Collection of tuples with each member of the bag allowed a different schema	{ ('Clouds', 'Grids') ('Clouds', ('IaaS', 'PaaS')) }
Map	A collection of data items associated with a set of keys; the keys are a bag of atomic data	['Microsoft' → { ('Windows') ('Azure') } 'Redhat' → 'Linux']

Table 6.9 Pig Latin Operators

Command	Description
LOAD	Read data from the file system.
STORE	Write data to the file system.
FOREACH GENERATE	Apply an expression to each record and output one or more records.
FILTER	Apply a predicate and remove records that do not return true.
GROUP/COGROUP	Collect records with the same key from one or more inputs.
JOIN	Join two or more inputs based on a key.
CROSS	Cross product two or more inputs.
UNION	Merge two or more data sets.
SPLIT	Split data into two or more sets, based on filter conditions.
ORDER	Sort records based on a key.
DISTINCT	Remove duplicate tuples.
STREAM	Send all records through a user-provided binary.
DUMP	Write output to stdout.
LIMIT	Limit the number of records.

supported collective operations. This is an important general approach to parallelism and was seen, for example, a long time ago in High Performance Fortran [68]. There are several discussions of Pig and Pig Latin in the literature [69,70], and here we summarize the language features. Table 6.8 lists the four data types in Pig Latin and Table 6.9 the 14 operators.

Example 6.7 Parallel Programming with Sawzall and Pig Latin

First one would read in data with commands such as:

```
Queries = LOAD 'filewithdata.txt' USING myUDF() AS (userId, queryString, timestamp);
```

The *LOAD* command returns a handle to the bag *Queries*. *myUDF()* is an optional custom reader which is an example of a user-defined function. The *AS* syntax defines the schema of the tuples that make up the bag *Queries*. The data can now be processed by commands such as:

```
Expanded_queries = FOREACH queries GENERATE userID, expandQueryUDF(queryString);
```

The example maps each tuple in *queries* as determined by the user-defined function *expandQueryUDF. FOREACH* runs over all tuples in the bag. Alternatively, one could use *FILTER* as in the following example to remove all tuples with a *userID* equal to *Alice* as a string:

```
Real_queries = FILTER queries BY userID neq 'Alice';
```

Pig Latin offers the equivalent of a SQL JOIN capability using COGROUP as in

```
Grouped_data = COGROUP results BY queryString, revenue BY queryString;
```

Where results and revenue are bags of tuples (either from LOAD or processing of LOADed data)

```
Results: queryString, url, position)
Revenue: (queryString, adSlot, amount)
```

COGROUP is more general than *JOIN* in the sense that *COGROUP* does not produce a set of tuples (*queryString*, *url*, *position*, *adSlot*, *amount*), but rather a tuple consisting of three fields. The first field is *querystring*, the second field is all tuples from *Results* with the value of *querystring*, and the third field is all tuples from *Revenue* with the value of *queryString*. *FLATTEN* can map the result of *COGROUP* to SQL Join (*queryString*, *url*, *position*, *adSlot*, *amount*) syntax.

Pig Latin operations are performed in the order listed as a data flow pipeline. This is in contrast to declarative SQL where one just specifies "what" has to be done, not how it is to be done. Pig Latin supports user-defined functions, as illustrated in the preceding code, as first-class operations in the language which could be an advantage over SQL. User-defined functions can be placed in *Load*, *Store*, *Group*, *Filter*, and *Foreach* operators, depending on user preference. Note that the rich set of data flow operations allowed in Pig Latin makes it similar to a scripting approach to workflow, as we discussed in Section 5.5.5. The Pig! Apache project [69] maps Pig Latin into a sequence of MapReduce operations implemented in Hadoop.

6.2.6 Mapping Applications to Parallel and Distributed Systems

In the past, Fox has discussed mapping applications to different hardware and software in terms of five application architectures [71]. These initial five categories are listed in Table 6.10, followed by a sixth emerging category to describe data-intensive computing [72,73]. The original classifications largely described simulations and were not aimed directly at data analysis. It is instructive to briefly summarize them and then explain the new category. Category 1 was popular 20 years ago, but is no longer significant. It describes applications that can be parallelized with lock-step operations controlled by hardware.

Such a configuration would run on SIMD (*single-instruction and multiple-data*) machines, whereas category 2 is now much more important and corresponds to a SPMD (*single-program and multiple-data*) model running on MIMD (*multiple instruction multiple data*) machines. Here, each decomposed

Table 6.10 Application Classification for Parallel and Distributed Systems

Category	Class	Description	Machine Architecture
1	Synchronous	The problem class can be implemented with instruction-level lockstep operation as in SIMD architectures.	SIMD
2	Loosely synchronous (BSP or bulk synchronous processing)	These problems exhibit iterative compute-communication stages with independent compute (map) operations for each CPU that are synchronized with a communication step. This problem class covers many successful MPI applications including partial differential equation solutions and particle dynamics applications.	MIMD on MPP (massively parallel processor)
3	Asynchronous	Illustrated by Compute Chess and Integer Programming; combinatorial search is often supported by dynamic threads. This is rarely important in scientific computing, but it is at the heart of operating systems and concurrency in consumer applications such as Microsoft Word.	Shared memory
4	Pleasingly parallel	Each component is independent. In 1988, Fox estimated this at 20 percent of the total number of applications, but that percentage has grown with the use of grids and data analysis applications including, for example, the Large Hadron Collider analysis for particle physics.	Grids moving to clouds
5	Metaproblems	These are coarse-grained (asynchronous or data flow) combinations of categories 1-4 and 6. This area has also grown in importance and is well supported by grids and described by workflow in Section 3.5.	Grids of clusters
6	MapReduce++ (Twister)	This describes file (database) to file (database) operations which have three subcategories (see also Table 6.11): 6a) Pleasingly Parallel Map Only (similar to category 4) 6b) Map followed by reductions 6c) Iterative "Map followed by reductions" (extension of current technologies that supports linear algebra and data mining)	Data-intensive clouds a) Master-worker or MapReduce b) MapReduce c) Twister

unit executes the same program, but at any given time, there is no requirement that the same instruction be executed. Category 1 corresponds to regular problems, whereas category 2 includes dynamic irregular cases with complex geometries for solving partial differential equations or particle dynamics. Note that synchronous problems are still around, but they are run on MIMD machines with the SPMD model. Also note that category 2 consists of compute–communicate phases and the computations are synchronized by communication. No additional synchronization is needed.

Category 3 consists of asynchronously interacting objects and is often considered the people's view of a typical parallel problem. It probably does describe the concurrent threads in a modern operating system, as well as some important applications, such as event-driven simulations and areas such as search in computer games and graph algorithms. Shared memory is natural due to the low latency often needed to perform dynamic synchronization. It wasn't clear in the past, but now it appears that this category is not very common in large-scale parallel problems of importance.

Category 4 is the simplest algorithmically, with disconnected parallel components. However, the importance of this category has probably grown since the original 1988 analysis when it was estimated to account for 20 percent of all parallel computing. Both grids and clouds are very natural for this class, which does not need high-performance communication between different nodes.

Category 5 refers to the coarse-grained linkage of different "atomic" problems, and this was fully covered in Section 5.5. This area is clearly common and is expected to grow in importance. Remember the critical observation in Section 5.5 that we use a two-level programming model with the metaproblem (workflow) linkage specified in one fashion and the component problems with approaches such as those in this chapter. Grids or clouds are suitable for metaproblems as coarse-grained decomposition does not usually require stringent performance.

As noted earlier, we added a sixth category to cover data-intensive applications motivated by the clear importance of MapReduce as a new programming model. We call this category MapReduce++, and it has three subcategories: "map only" applications similar to pleasingly parallel category 4 [41,43,74,75]; the classic MapReduce with file-to-file operations consisting of parallel maps followed by parallel reduce operations; and a subcategory that captures the extended MapReduce version covered in Section 6.2.2. Note that category 6 is a subset of categories 2 and 4 with additional reading and writing of data and a specialized, loosely synchronous structure compared to that used in data analysis. This comparison is made clearer in Table 6.11.

6.3 PROGRAMMING SUPPORT OF GOOGLE APP ENGINE

Section 4.4.2 introduced the Google App Engine infrastructure in Figures 4.20. In this section, we describe the programming model supported by GAE. The access links to the GAE platform are provided in Chapter 4.

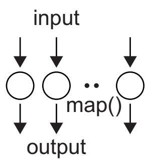
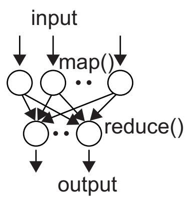
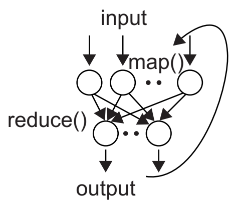
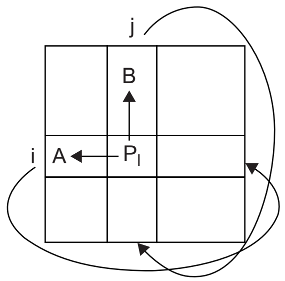
Discuss Programming the Google App Engine.

6.3.1 Programming the Google App Engine

Several web resources (e.g., http://code.google.com/appengine/) and specific books and articles (e.g., www.byteonic.com/2009/overview-of-java-support-in-google-app-engine/) discuss how to program GAE. Figure 6.17 summarizes some key features of GAE programming model for two supported languages: Java and Python. A client environment that includes an *Eclipse* plug-in for Java allows you to debug your GAE on your local machine. Also, the GWT Google Web Toolkit is available for Java web application developers. Developers can use this, or any other language using a JVM-based interpreter or compiler, such as JavaScript or Ruby. Python is often used with frameworks such as Django and CherryPy, but Google also supplies a built in *webapp* Python environment.

There are several powerful constructs for storing and accessing data. The data store is a NOSQL data management system for entities that can be, at most, 1 MB in size and are labeled by a set of schema-less properties. Queries can retrieve entities of a given kind filtered and sorted by the values of the properties. Java offers Java Data Object (JDO) and Java Persistence API (JPA) interfaces implemented by the open source Data Nucleus Access platform, while Python has a SQL-like query language called GQL. The data store is strongly consistent and uses optimistic concurrency control.

Table 6.11 Comparison of MapReduce++ Subcategories along with the Loosely Synchronous Category Used in MPI

Map-Only	**Classic MapReduce**	**Iterative MapReduce**	**Loosely Synchronous**
			
• Document conversion (e.g., PDF->HTML) • Brute force searches in cryptography • Parametric sweeps • Gene assembly • PolarGrid Matlab data analysis (www.polargrid.org)	• High-energy physics (HEP) histograms • Distributed search • Distributed sort • Information retrieval • Calculation of pairwise distances for sequences (BLAST)	• Expectation maximization algorithms • Linear algebra • Data mining including • Clustering • K-means • Deterministic annealing clustering • Multidimensional scaling (MDS)	• Many MPI scientific applications utilizing a wide variety of communication constructs including local interactions • Solving differential equations and particle dynamics with short-range forces
⟵ **Domain of MapReduce and Iterative Extensions** ⟶			**MPI**

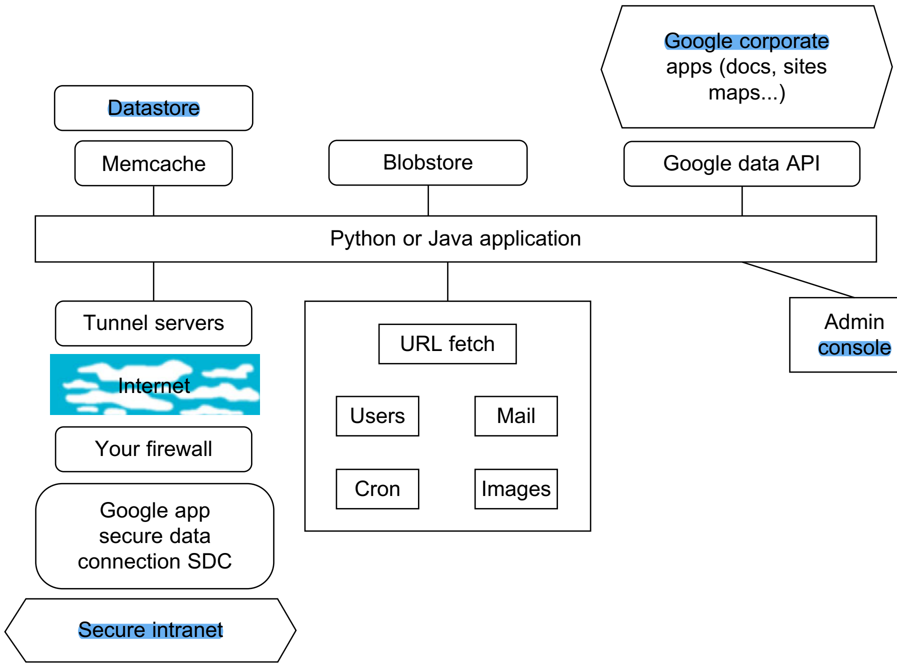


FIGURE 6.17

Programming environment for Google AppEngine.

An update of an entity occurs in a transaction that is retried a fixed number of times if other processes are trying to update the same entity simultaneously. Your application can execute multiple data store operations in a single transaction which either all succeed or all fail together. The data store implements transactions across its distributed network using "entity groups." A transaction manipulates entities within a single group. Entities of the same group are stored together for efficient execution of transactions. Your GAE application can assign entities to groups when the entities are created. The performance of the data store can be enhanced by in-memory caching using the *memcache*, which can also be used independently of the data store.

Recently, Google added the *blobstore* which is suitable for large files as its size limit is 2 GB. There are several mechanisms for incorporating external resources. The *Google SDC Secure Data Connection* can tunnel through the Internet and link your intranet to an external GAE application. The *URL Fetch* operation provides the ability for applications to fetch resources and communicate with other hosts over the Internet using HTTP and HTTPS requests. There is a specialized mail mechanism to send e-mail from your GAE application.

Applications can access resources on the Internet, such as web services or other data, using GAE's URL fetch service. The URL fetch service retrieves web resources using the same high-speed Google infrastructure that retrieves web pages for many other Google products. There are dozens of Google "corporate" facilities including maps, sites, groups, calendar, docs, and YouTube, among others. These support the *Google Data API* which can be used inside GAE.

An application can use Google Accounts for *user* authentication. Google Accounts handles user account creation and sign-in, and a user that already has a Google account (such as a Gmail account) can use that account with your app. GAE provides the ability to manipulate image data using a dedicated *Images* service which can resize, rotate, flip, crop, and enhance images. An application can perform tasks outside of responding to web requests. Your application can perform these tasks on a schedule that you configure, such as on a daily or hourly basis using "cron jobs," handled by the *Cron* service.

Alternatively, the application can perform tasks added to a queue by the application itself, such as a background task created while handling a request. A GAE application is configured to consume resources up to certain limits or quotas. With quotas, GAE ensures that your application won't exceed your budget, and that other applications running on GAE won't impact the performance of your app. In particular, GAE use is free up to certain quotas.

6.3.2 Google File System (GFS)

GFS was built primarily as the fundamental storage service for Google's search engine. As the size of the web data that was crawled and saved was quite substantial, Google needed a distributed file system to redundantly store massive amounts of data on cheap and unreliable computers. None of the traditional distributed file systems can provide such functions and hold such large amounts of data. In addition, GFS was designed for Google applications, and Google applications were built for GFS. In traditional file system design, such a philosophy is not attractive, as there should be a clear interface between applications and the file system, such as a POSIX interface.

There are several assumptions concerning GFS. One is related to the characteristic of the cloud computing hardware infrastructure (i.e., the high component failure rate). As servers are composed of inexpensive commodity components, it is the norm rather than the exception that concurrent failures will occur all the time. Another concerns the file size in GFS. GFS typically will hold a large number of huge files, each 100 MB or larger, with files that are multiple GB in size quite common. Thus, Google has chosen its file data block size to be 64 MB instead of the 4 KB in typical traditional file systems. The I/O pattern in the Google application is also special. Files are typically written once, and the write operations are often the appending data blocks to the end of files. Multiple appending operations might be concurrent. There will be a lot of large streaming reads and only a little random access. As for large streaming reads, highly sustained throughput is much more important than low latency.

Thus, Google made some special decisions regarding the design of GFS. As noted earlier, a 64 MB block size was chosen. Reliability is achieved by using replications (i.e., each chunk or data block of a file is replicated across more than three chunk servers). A single master coordinates access as well as keeps the metadata. This decision simplified the design and management of the whole cluster. Developers do not need to consider many difficult issues in distributed systems, such as distributed consensus. There is no data cache in GFS as large streaming reads and writes represent neither time nor space locality. GFS provides a similar, but not identical, POSIX file system accessing interface. The distinct difference is that the application can even see the physical location of file blocks. Such a scheme can improve the upper-layer applications. The customized API can simplify the problem and focus on Google applications.

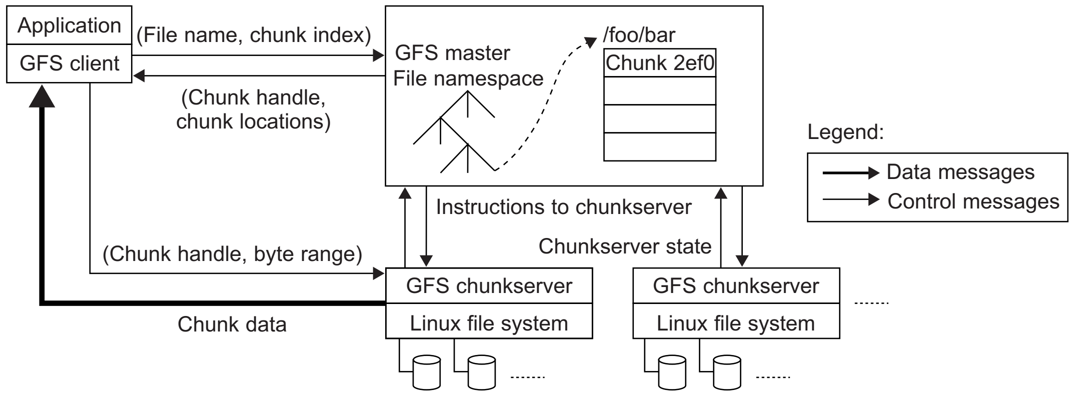


FIGURE 6.18

Architecture of Google File System (GFS).

(*Courtesy of S. Ghemawat, et al.* [53])

The customized API adds snapshot and record append operations to facilitate the building of Google applications.

Figure 6.18 shows the GFS architecture. It is quite obvious that there is a single master in the whole cluster. Other nodes act as the chunk servers for storing data, while the single master stores the metadata. The file system namespace and locking facilities are managed by the master. The master periodically communicates with the chunk servers to collect management information as well as give instructions to the chunk servers to do work such as load balancing or fail recovery.

The master has enough information to keep the whole cluster in a healthy state. With a single master, many complicated distributed algorithms can be avoided and the design of the system can be simplified. However, this design does have a potential weakness, as the single GFS master could be the performance bottleneck and the single point of failure. To mitigate this, Google uses a shadow master to replicate all the data on the master, and the design guarantees that all the data operations are performed directly between the client and the chunk server. The control messages are transferred between the master and the clients and they can be cached for future use. With the current quality of commodity servers, the single master can handle a cluster of more than 1,000 nodes.

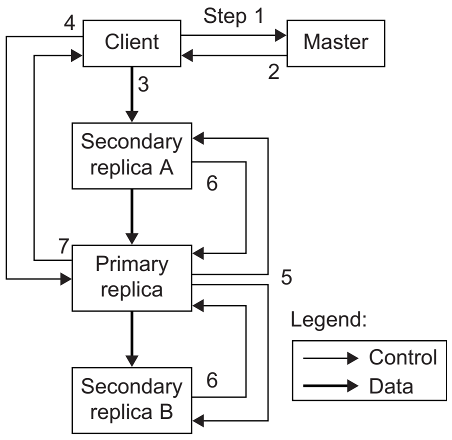


FIGURE 6.19

Data mutation sequence in GFS.

Figure 6.19 shows the data mutation (write, append operations) in GFS. Data blocks must be created for all replicas. The goal is to minimize involvement of the master. The mutation takes the following steps:

1. The client asks the master which chunk server holds the current lease for the chunk and the locations of the other replicas. If no one has a lease, the master grants one to a replica it chooses (not shown).
2. The master replies with the identity of the primary and the locations of the other (secondary) replicas. The client caches this data for future mutations. It needs to contact the master again only when the primary becomes unreachable or replies that it no longer holds a lease.
3. The client pushes the data to all the replicas. A client can do so in any order. Each chunk server will store the data in an internal LRU buffer cache until the data is used or aged out. By decoupling the data flow from the control flow, we can improve performance by scheduling the expensive data flow based on the network topology regardless of which chunk server is the primary.
4. Once all the replicas have acknowledged receiving the data, the client sends a write request to the primary. The request identifies the data pushed earlier to all the replicas. The primary assigns consecutive serial numbers to all the mutations it receives, possibly from multiple clients, which provides the necessary serialization. It applies the mutation to its own local state in serial order.
5. The primary forwards the write request to all secondary replicas. Each secondary replica applies mutations in the same serial number order assigned by the primary.
6. The secondaries all reply to the primary indicating that they have completed the operation.
7. The primary replies to the client. Any errors encountered at any replicas are reported to the client. In case of errors, the write corrects at the primary and an arbitrary subset of the secondary replicas. The client request is considered to have failed, and the modified region is left in an inconsistent state. Our client code handles such errors by retrying the failed mutation. It will make a few attempts at steps 3 through 7 before falling back to a retry from the beginning of the write.

Thus, besides the writing operation provided by GFS, special appending operations can be used to append the data blocks to the end of the files. The reason for providing such operations is that some of the Google applications need a lot of append operations. For example, while crawlers are gathering data from the web, the contents of web pages will be appended to page files. Thus, the appending operation is provided and optimized. The client specifies data to be appended and GFS appends it to the file atomically at least once. GFS picks the offset and the clients cannot decide the offset of the data position. The appending operation works for concurrent writers.

GFS was designed for high fault tolerance and adopted some methods to achieve this goal. Master and chunk servers can be restarted in a few seconds, and with such a fast recovery capability, the window of time in which the data is unavailable can be greatly reduced. As we mentioned before, each chunk is replicated in at least three places and can tolerate at least two data crashes for a single chunk of data. The shadow master handles the failure of the GFS master. For data integrity, GFS makes checksums on every 64 KB block in each chunk. With the previously discussed design and implementation, GFS can achieve the goals of high availability (HA), high performance, and large scale. GFS demonstrates how to support large-scale processing workloads on commodity hardware designed to tolerate frequent component failures optimized for huge files that are mostly appended and read.

6.3.3 BigTable, Google's NOSQL System

In this section, we continue discussing key technologies in the Google cloud environment. We already discussed the most well-known Google technology, MapReduce, in Section 6.2.2, and Sawzall in Section 6.2.5. Here, we focus on another innovative Google technology: BigTable. We will cover Chubby in Section 6.3.4 and covered GFS in previous section. BigTable was designed to provide a service for storing and retrieving structured and semistructured data. BigTable applications include storage of web pages, per-user data, and geographic locations. Here we use web pages to represent URLs and their associated data, such as contents, crawled metadata, links, anchors, and page rank values. Per-user data has information for a specific user and includes such data as user preference settings, recent queries/search results, and the user's e-mails. Geographic locations are used in Google's well-known Google Earth software. Geographic locations include physical entities (shops, restaurants, etc.), roads, satellite image data, and user annotations.

The scale of such data is incredibly large. There will be billions of URLs, and each URL can have many versions, with an average page size of about 20 KB per version. The user scale is also huge. There are hundreds of millions of users and there will be thousands of queries per second. The same scale occurs in the geographic data, which might consume more than 100 TB of disk space.

It is not possible to solve such a large scale of structured or semistructured data using a commercial database system. This is one reason to rebuild the data management system; the resultant system can be applied across many projects for a low incremental cost. The other motivation for rebuilding the data management system is performance. Low-level storage optimizations help increase performance significantly, which is much harder to do when running on top of a traditional database layer.

The design and implementation of the BigTable system has the following goals. The applications want asynchronous processes to be continuously updating different pieces of data and want access to the most current data at all times. The database needs to support very high read/write rates and the scale might be millions of operations per second. Also, the database needs to support efficient scans over all or interesting subsets of data, as well as efficient joins of large one-to-one and one-to-many data sets. The application may need to examine data changes over time (e.g., contents of a web page over multiple crawls).

Thus, BigTable can be viewed as a distributed multilevel map. It provides a fault-tolerant and persistent database as in a storage service. The BigTable system is scalable, which means the system has thousands of servers, terabytes of in-memory data, petabytes of disk-based data, millions of reads/writes per second, and efficient scans. Also, BigTable is a self-managing system (i.e., servers can be added/removed dynamically and it features automatic load balancing). Design/initial implementation of BigTable began at the beginning of 2004. BigTable is used in many projects, including Google Search, Orkut, and Google Maps/Google Earth, among others. One of the largest BigTable cell manages ~200 TB of data spread over several thousand machines.

The BigTable system is built on top of an existing Google cloud infrastructure. BigTable uses the following building blocks:

1. GFS: stores persistent state
2. Scheduler: schedules jobs involved in BigTable serving
3. Lock service: master election, location bootstrapping
4. MapReduce: often used to read/write BigTable data

Example 6.8 BigTable Data Model Used in Mass Media

BigTable provides a simplified data model compared to traditional database systems. Figure 6.20(a) shows the data model of a sample table, Web Table. Web Table stores the data about a web page. Each web page can be accessed by the URL. The URL is considered the row index. The column provides different data related to the corresponding URL—for example, different versions of the contents, and the anchors appearing in the web page. In this sense, BigTable is a distributed multidimensional stored sparse map.

The map is indexed by row key, column key, and timestamp—that is, (row: string, column: string, time: int64) maps to string (cell contents). Rows are ordered in lexicographic order by row key. The row range for a table is dynamically partitioned and each row range is called "Tablet." Syntax for columns is shown as a (family:qualifier) pair. Cells can store multiple versions of data with timestamps.

Such a data model is a good match for most of Google's (and other organizations') applications. For rows, *Name* is an arbitrary string and access to data in a row is atomic. This is different from the traditional

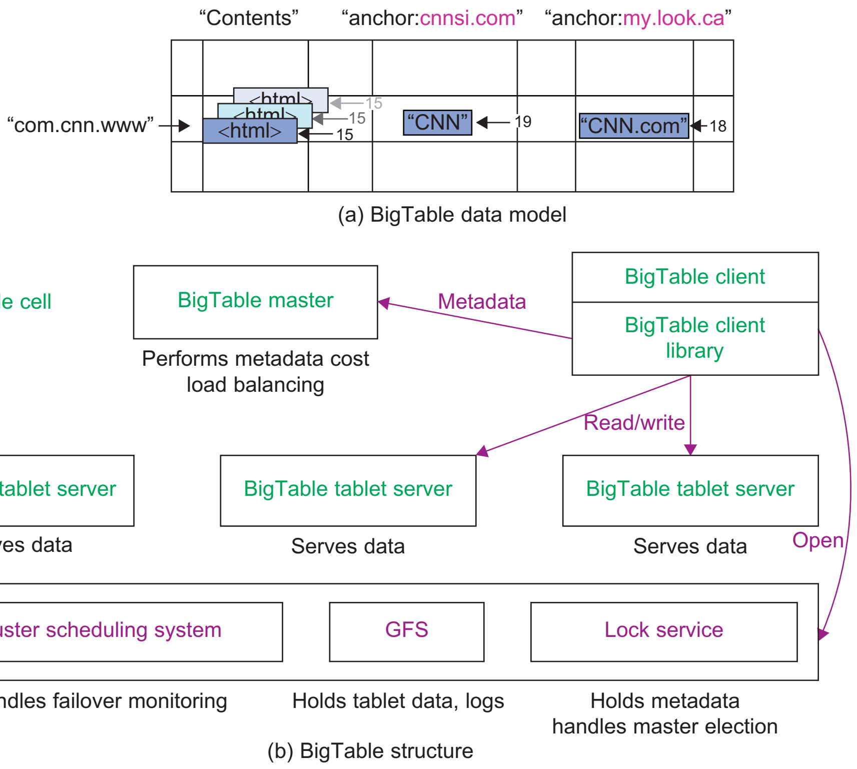


FIGURE 6.20

BigTable data model and system structure.

(*Courtesy of Chang, et al.* [11])

relational database which provides abundant atomic operations (transactions). Row creation is implicit upon storing data. Rows are ordered lexicographically, that is, close together lexicographically, usually on one or a small number of machines.

Large tables are broken into tablets at row boundaries. A tablet holds a contiguous range of rows. Clients can often choose row keys to achieve locality. The system aims for about 100 MB to 200 MB of data per tablet. Each serving machine is responsible for about 100 tablets. This can achieve faster recovery times as 100 machines each pick up one tablet from the failed machine. This also results in fine-grained load balancing, that is, migrating tablets away from the overloaded machine. Similar to the design in GFS, a master machine in BigTable makes load-balancing decisions.

Figure 6.20(b) shows the BigTable system structure. A BigTable master manages and stores the metadata of the BigTable system. BigTable clients use the BigTable client programming library to communicate with the BigTable master and tablet servers. BigTable relies on a highly available and persistent distributed lock service called Chubby [76] discussed in Section 6.3.4.

6.3.3.1 *Tablet Location Hierarchy*

Figure 6.21 shows how to locate the BigTable data starting from the file stored in Chubby. The first level is a file stored in Chubby that contains the location of the root tablet. The root tablet contains the location of all tablets in a special METADATA table. Each METADATA tablet contains the location of a set of user tablets. The root tablet is just the first tablet in the METADATA table, but is treated specially; it is never split to ensure that the tablet location hierarchy has no more than three levels.

The METADATA table stores the location of a tablet under a row key that is an encoding of the tablet's table identifier and its end row. BigTable includes many optimizations and fault-tolerant features. Chubby can guarantee the availability of the file for finding the root tablet. The BigTable master can quickly scan the tablet servers to determine the status of all nodes. Tablet servers use

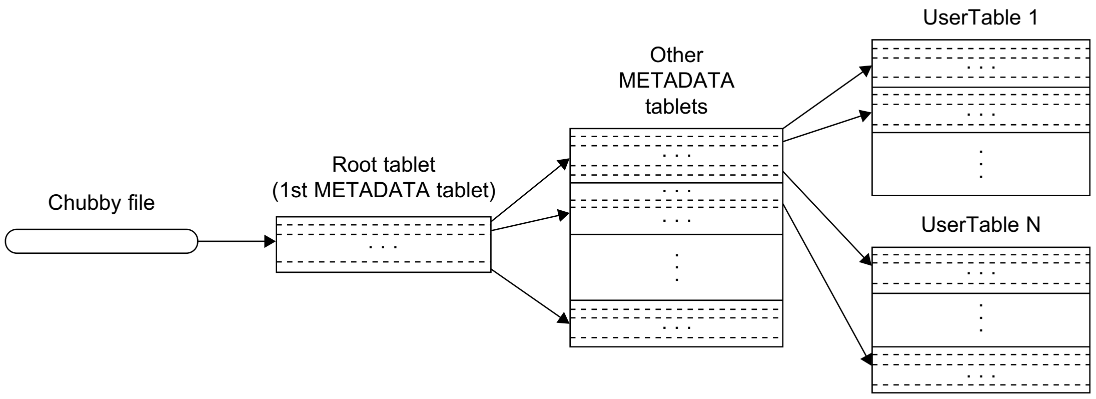


FIGURE 6.21

Tablet location hierarchy in using the BigTable.

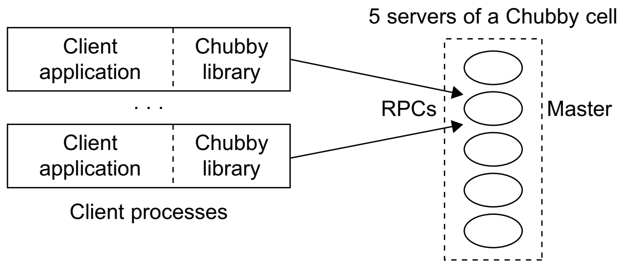


FIGURE 6.22

Structure of Google Chubby for distributed lock service.

compaction to store data efficiently. Shared logs are used for logging the operations of multiple tablets so as to reduce the log space as well as keep the system consistent.

6.3.4 Chubby, Google's Distributed Lock Service

Chubby [76] is intended to provide a coarse-grained locking service. It can store small files inside Chubby storage which provides a simple namespace as a file system tree. The files stored in Chubby are quite small compared to the huge files in GFS. Based on the Paxos agreement protocol, the Chubby system can be quite reliable despite the failure of any member node. Figure 6.22 shows the overall architecture of the Chubby system.

Each Chubby cell has five servers inside. Each server in the cell has the same file system namespace. Clients use the Chubby library to talk to the servers in the cell. Client applications can perform various file operations on any server in the Chubby cell. Servers run the Paxos protocol to make the whole file system reliable and consistent. Chubby has become Google's primary internal name service. GFS and BigTable use Chubby to elect a primary from redundant replicas.

6.4 PROGRAMMING ON AMAZON AWS AND MICROSOFT AZURE

In this section, we will consider the programming support in the AWS platform. First we will review the AWS platform and its updated service offerings. Then we will study the EC2, S3, and Simple DB services with programming examples. Returning to the programming environment features in Figures 4.22 and 4.23, Amazon (like Azure) offers a Relational Database Service (RDS) with a messaging interface sketched in Section 6.1.3. The Elastic MapReduce capability is equivalent to Hadoop running on the basic EC2 offering. Amazon has NOSQL support in SimpleDB introduced in Section 6.1.3 and discussed in Section 6.4.5. However, Amazon does not directly support BigTable as described in Section 6.3.4.

Now we will highlight a few more capabilities that are listed in Table 4.6. Amazon offers the Simple Queue Service (SQS) and Simple Notification Service (SNS), which are the cloud implementations of services discussed in Sections 5.2 and 5.4.5. Note that brokering systems run very efficiently in clouds and offer a striking model for controlling sensors and giving back-office support for a growing number of smartphones and tablets [77]. We further note the auto-scaling and elastic load balancing services which support related capabilities. Auto-scaling enables you to automatically scale your Amazon EC2 capacity up or down according to conditions that you define. With auto-scaling, you can ensure that the number of Amazon EC2 instances you're using scales up seamlessly during demand spikes to maintain performance, and scales down automatically during demand lulls to minimize cost.

Elastic load balancing automatically distributes incoming application traffic across multiple Amazon EC2 instances and allows you to avoid nonoperating nodes and to equalize load on functioning images. Both auto-scaling and elastic load balancing are enabled by CloudWatch which monitors running instances. CloudWatch is a web service that provides monitoring for AWS cloud resources, starting with Amazon EC2. It provides customers with visibility into resource utilization, operational performance, and overall demand patterns—including metrics such as CPU utilization, disk reads and writes, and network traffic.

6.4.1 Programming on Amazon EC2

Amazon was the first company to introduce VMs in application hosting. Customers can rent VMs instead of physical machines to run their own applications. By using VMs, customers can load any software of their choice. The elastic feature of such a service is that a customer can create, launch, and terminate server instances as needed, paying by the hour for active servers. Amazon provides several types of preinstalled VMs. Instances are often called *Amazon Machine Images (AMIs)* which are preconfigured with operating systems based on Linux or Windows, and additional software.

Table 6.12 defines three types of AMI. Figure 6.24 shows an execution environment. AMIs are the templates for instances, which are running VMs. The workflow to create a VM is

$$\text{Create an AMI} \rightarrow \text{Create Key Pair} \rightarrow \text{Configure Firewall} \rightarrow \text{Launch} \tag{6.3}$$

This sequence is supported by public, private, and paid AMIs shown in Figure 6.24. The AMIs are formed from the virtualized compute, storage, and server resources shown at the bottom of Figure 6.23.

Table 6.12 Three Types of AMI

Image Type	AMI Definition
Private AMI	Images created by you, which are private by default. You can grant access to other users to launch your private images.
Public AMI	Images created by users and released to the AWS community, so anyone can launch instances based on them and use them any way they like. AWS lists all public images at http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryID=171.
Paid QAMI	You can create images providing specific functions that can be launched by anyone willing to pay you per each hour of usage on top of Amazon's charges.

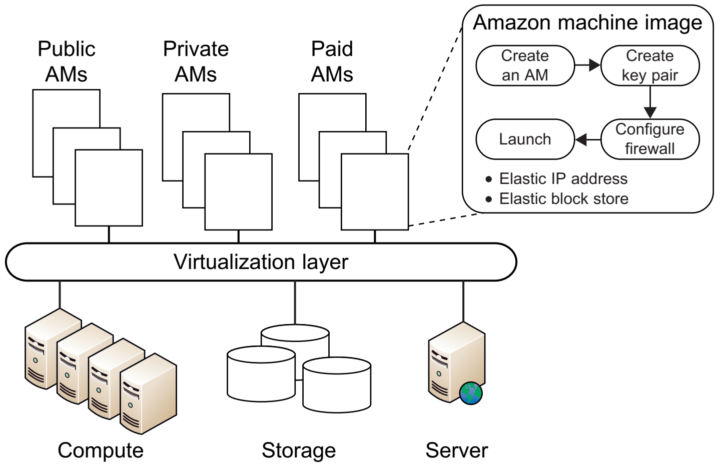


FIGURE 6.23

Amazon EC2 execution environment.

Table 6.13 Instance Types Available on Amazon EC2 (October 6, 2010)

Compute Instance	Memory GB	ECU or EC2 Units	Virtual Cores	Storage GB	32/64 Bit
Standard: small	1.7	1	1	160	32
Standard: large	7.5	4	2	850	64
Standard: extra large	15	8	4	1690	64
Micro	0.613	Up to 2		Only EBS	32 or 64
High-memory	17.1	6.5	2	420	64
High-memory: double	34.2	13	4	850	64
High-memory: quadruple	68.4	26	8	1690	64
High-CPU: medium	1.7	5	2	350	32
High-CPU: extra large	7	20	8	1690	64
Cluster compute	23	33.5	8	1690	64

Example 6.9 Use of EC2 Services in the AWS Platform

Table 6.13 defines the IaaS instances available in October 2010 in five broad classes:

1. **Standard instances** are well suited for most applications.
2. **Micro instances** provide a small number of consistent CPU resources and allow you to burst CPU capacity when additional cycles are available. They are well suited for lower throughput applications and web sites that consume significant compute cycles periodically.
3. **High-memory instances** offer large memory sizes for high-throughput applications, including database and memory caching applications.

Table 6.14 Cost of Amazon On-Demand VM Instant Types (October 6, 2010)

VM Instance Type	Size	Linux/UNIX Usage	Windows Usage
	Small (default)	$0.085 per hour	$0.12 per hour
Standard instances	Large	$0.34 per hour	$0.48 per hour
	Extra large	$0.68 per hour	$0.96 per hour
Micro instances	Micro	$0.02 per hour	$0.03 per hour
High-memory instances	Extra large	$0.50 per hour	$0.62 per hour
	Double extra large	$1.00 per hour	$1.24 per hour
	Quadruple extra large	$2.00 per hour	$2.48 per hour
Cluster compute instances	Quadruple extra large	$1.60 per hour	Not available

4. **High-CPU instances** have proportionally more CPU resources than memory (RAM) and are well suited for compute-intensive applications.
5. **Cluster compute instances** provide proportionally high CPU resources with increased network performance and are well suited for high-performance computing (HPC) applications and other demanding network-bound applications. They use 10 Gigabit Ethernet interconnections.

The cost in the third column is expressed in terms of EC2 Compute Units (ECUs) where one ECU provides the CPU capacity of a 1.0–1.2 GHz 2007 Opteron or 2007 Xeon processor. This leads to the cost per hour for CPUs shown in Table 6.14. Note that a real-world use of EC2 must pay for use of many different resources; the CPU charge in Table 6.14 is just one component and all charges (which naturally change often and so the reader should get the latest values online) are available on the AWS web site.

■

6.4.2 Amazon Simple Storage Service (S3)

Amazon S3 provides a simple web services interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web. S3 provides the object-oriented storage service for users. Users can access their objects through *Simple Object Access Protocol* (SOAP) with either browsers or other client programs which support SOAP. SQS is responsible for ensuring a reliable message service between two processes, even if the receiver processes are not running. Figure 6.24 shows the S3 execution environment.

The fundamental operation unit of S3 is called an *object*. Each object is stored in a *bucket* and retrieved via a unique, developer-assigned key. In other words, the bucket is the container of the object. Besides unique key attributes, the object has other attributes such as values, metadata, and access control information. From the programmer's perspective, the storage provided by S3 can be viewed as a very coarse-grained key-value pair. Through the key-value programming interface, users can write, read, and delete objects containing from 1 byte to 5 gigabytes of data each. There are two types of web service interface for the user to access the data stored in Amazon clouds. One is a REST (web 2.0) interface, and the other is a SOAP interface. Here are some key features of S3:

- Redundant through geographic dispersion.
- Designed to provide 99.999999999 percent durability and 99.99 percent availability of objects over a given year with cheaper reduced redundancy storage (RRS).

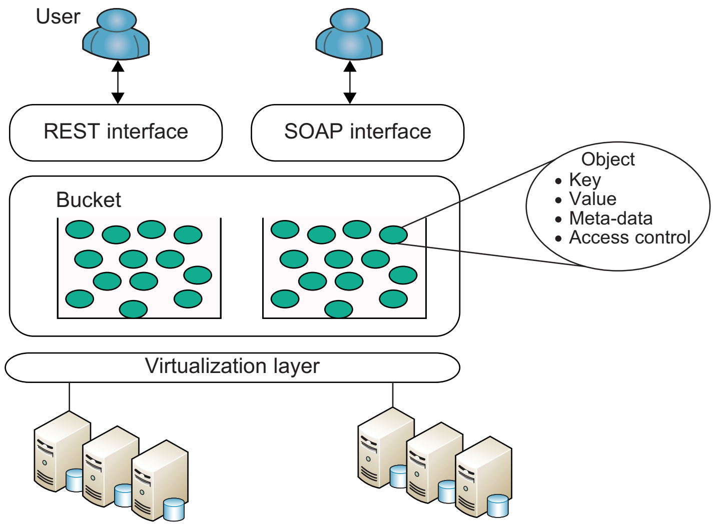


FIGURE 6.24

Amazon S3 execution environment.

- Authentication mechanisms to ensure that data is kept secure from unauthorized access. Objects can be made private or public, and rights can be granted to specific users.
- Per-object URLs and ACLs (access control lists).
- Default download protocol of HTTP. A BitTorrent protocol interface is provided to lower costs for high-scale distribution.
- \$0.055 (more than 5,000 TB) to 0.15 per GB per month storage (depending on total amount).
- First 1 GB per month input or output free and then \$.08 to \$0.15 per GB for transfers outside an S3 region.
- There is no data transfer charge for data transferred between Amazon EC2 and Amazon S3 within the same region or for data transferred between the Amazon EC2 Northern Virginia region and the Amazon S3 U.S. Standard region (as of October 6, 2010).

6.4.3 Amazon Elastic Block Store (EBS) and SimpleDB

The *Elastic Block Store* (EBS) provides the volume block interface for saving and restoring the virtual images of EC2 instances. Traditional EC2 instances will be destroyed after use. The status of EC2 can now be saved in the EBS system after the machine is shut down. Users can use EBS to save persistent data and mount to the running instances of EC2. Note that S3 is "Storage as a Service" with a messaging interface. EBS is analogous to a distributed file system accessed by traditional OS disk access mechanisms. EBS allows you to create storage volumes from 1 GB to 1 TB that can be mounted as EC2 instances.

Multiple volumes can be mounted to the same instance. These storage volumes behave like raw, unformatted block devices, with user-supplied device names and a block device interface. You can create a file system on top of Amazon EBS volumes, or use them in any other way you would use a block device (like a hard drive). Snapshots are provided so that the data can be saved incrementally. This can improve performance when saving and restoring data. In terms of pricing, Amazon provides a similar pay-per-use schema as EC2 and S3. Volume storage charges are based on the amount of storage users allocate until it is released, and is priced at $0.10 per GB/month. EBS also charges $0.10 per 1 million I/O requests made to the storage (as of October 6, 2010). The equivalent of EBS has been offered in open source clouds such Nimbus.

6.4.3.1 Amazon SimpleDB Service

SimpleDB provides a simplified data model based on the relational database data model. Structured data from users must be organized into domains. Each domain can be considered a table. The items are the rows in the table. A cell in the table is recognized as the value for a specific attribute (column name) of the corresponding row. This is similar to a table in a relational database. However, it is possible to assign multiple values to a single cell in the table. This is not permitted in a traditional relational database which wants to maintain data consistency.

Many developers simply want to quickly store, access, and query the stored data. SimpleDB removes the requirement to maintain database schemas with strong consistency. SimpleDB is priced at $0.140 per Amazon SimpleDB Machine Hour consumed with the first 25 Amazon SimpleDB Machine Hours consumed per month free (as of October 6, 2010). SimpleDB, like Azure Table, could be called "LittleTable," as they are aimed at managing small amounts of information stored in a distributed table; one could say BigTable is aimed at basic big data, whereas LittleTable is aimed at metadata. Amazon Dynamo [78] is an early research system along the lines of the production SimpleDB system.

6.4.4 Microsoft Azure Programming Support

Section 4.4.4 has introduced the Azure cloud system. In this section, we describe the programming model in more detail. Some key programming components, including the client development environment, SQLAzure, and the rich storage and programming subsystems, are shown in Figure 6.25. We focus on the features of importance in developing Azure programs. First we have the underlying Azure fabric consisting of virtualized hardware together with a sophisticated control environment implementing dynamic assignment of resources and fault tolerance. This implements *domain name system* (DNS) and monitoring capabilities. Automated service management allows service models to be defined by an XML template and multiple service copies to be instantiated on request.

When the system is running, services are monitored and one can access event logs, trace/debug data, performance counters, IIS web server logs, crash dumps, and other log files. This information can be saved in Azure storage. Note that there is no debugging capability for running cloud applications, but debugging is done from a trace. One can divide the basic features into *storage* and *compute* capabilities. The Azure application is linked to the Internet through a customized compute VM called a *web role* supporting basic Microsoft web hosting. Such configured VMs are often called *appliances*. The other important compute class is the *worker role* reflecting the importance in cloud computing of a pool of compute resources that are scheduled as needed. The roles support HTTP(S) and TCP. Roles offer the following methods:

- The *OnStart()* method which is called by the Fabric on startup, and allows you to perform initialization tasks. It reports a Busy status to the load balancer until you return *true*.
- The *OnStop()* method which is called when the role is to be shut down and gives a graceful exit.
- The *Run()* method which contains the main logic.

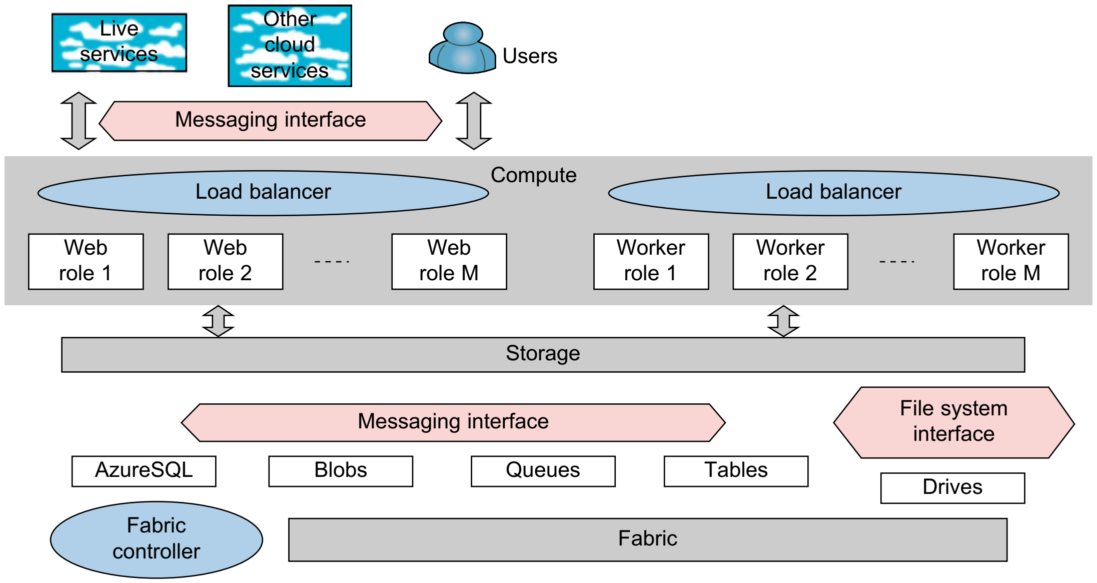


FIGURE 6.25

Features of the Azure cloud platform.

As discussed in Chapter 4, the Azure concept of roles is an interesting idea that we can expect to be expanded in terms of role types and use in other cloud environments. Figure 6.25 shows that compute roles can be *load-balanced*, similar to what GAE and AWS clouds have done (see Section 4.1).

6.4.4.1 SQLAzure

Azure offers a very rich set of *storage* capabilities, as shown in Figure 6.25. The *SQLAzure* service offers SQL Server as a service and is described in detail in Example 6.10. All the storage modalities are accessed with REST interfaces except for the recently introduced *Drives* that are analogous to Amazon EBS discussed in Section 6.4.3, and offer a file system interface as a durable NTFS volume backed by blob storage. The REST interfaces are automatically associated with URLs and all storage is replicated three times for fault tolerance and is guaranteed to be consistent in access.

The basic storage system is built from *blobs* which are analogous to S3 for Amazon. Blobs are arranged as a three-level hierarchy: *Account* → *Containers* → *Page or Block Blobs*. Containers are analogous to directories in traditional file systems with the account acting as the root. The *block blob* is used for streaming data and each such blob is made up as a sequence of blocks of up to 4 MB each, while each block has a 64 byte ID. Block blobs can be up to 200 GB in size. *Page blobs* are for random read/write access and consist of an array of pages with a maximum blob size of 1 TB. One can associate metadata with blobs as <name, value> pairs with up to 8 KB per blob.

6.4.4.2 Azure Tables

The Azure Table and Queue storage modes are aimed at much smaller data volumes. *Queues* provide reliable message delivery and are naturally used to support work spooling between web and worker roles. Queues consist of an unlimited number of messages which can be retrieved and processed at least once with an 8 KB limit on message size. Azure supports PUT, GET, and DELETE message operations as well as CREATE and DELETE for queues. Each account can have any number of Azure *tables* which consist of rows called *entities* and columns called *properties*.

There is no limit to the number of entities in a table and the technology is designed to scale well to a large number of entities stored on distributed computers. All entities can have up to 255 general properties which are <name, type, value> triples. Two extra properties, *PartitionKey* and *RowKey*, must be defined for each entity, but otherwise, there are no constraints on the names of properties—this table is very flexible! *RowKey* is designed to give each entity a unique label while *PartitionKey* is designed to be shared and entities with the same *PartitionKey* are stored next to each other; a good use of *PartitionKey* can speed up search performance. An entity can have, at most, 1 MB storage; if you need large value sizes, just store a link to a blob store in the *Table* property value. ADO.NET and LINQ support table queries.

Example 6.10 SQLAzure Data Services

Azure provides a sophisticated database programming interface. (More details can be found in www.microsoft.com/azure/sql.mspx). The SQLAzure data services can be viewed as more of a traditional relational database. Built on top of the current mature commercial software packages, SQLAzure can be considered a highly scalable, on-demand data storage and query processing utility service. The service interface of SQLAzure is based on standard web protocols and SQLAzure supports both SOAP and REST. As based on the relational database, the data model provided by SQL Data Services (SDS) is richer than the two NOSQL data management services discussed earlier (BigTable and SimpleDB).

The data model in SQLAzure includes the concepts of authority, container, and entity, with a flexible schema. After a user signs up for the data service, she can create an authority which is represented as a DNS name such as mydomain.data.database.windows.net. Here, mydomain is an authority created by the user and data.database.windows.net refers to the service. Users can create multiple authorities at any time. This DNS name will be resolved to a specific IP address and maps to a specific data center. Thus, an authority as well as its data will be located at a single data center. Under the highest level "authority" comes the concept of a container. An authority can have multiple containers (or none at all). A container has its identifications which can be used as the handle to find the corresponding container in an authority.

Containers are the places where users can put their data. Just like SimpleDB, users can begin to put data in a container without worrying about the data schema. Entities are the units that are stored in a container. An entity can store any number of user-defined properties and associated values (i.e., like the attributes and values in SimpleDB). There are two different kinds of containers: homogeneous and heterogeneous containers. A homogeneous container has all entities of the same type, like a relational database table. A heterogeneous container does not have such limitations. Figure 6.25 shows these concepts.

SDS is one of the building blocks of the Azure platform for building cloud applications. It does provide an enterprise-class data platform. Microsoft has built multiple data centers all over the world to host cloud applications from third parties. Multiple data centers can give the stored data high availability and security. Users do not need to worry about loss of data. Another key feature that SQLAzure provides is ease of development. SDS (in fact, the total Azure Platform SDK) can be integrated with Microsoft's powerful Visual Studio development environment. This can greatly improve the effectiveness and efficiency with which developers can make cloud applications.

6.5 EMERGING CLOUD SOFTWARE ENVIRONMENTS

In this section, we will assess popular cloud operating systems and emerging software environments. We cover the open source Eucalyptus and Nimbus, then examine OpenNebula, Sector/Sphere, and Open Stack. These environments were introduced in Chapter 3 from a virtualization perspective. Here, we provide details regarding programming requirements. We will also cover the Aneka cloud programming tools recently developed at the University of Melbourne.

6.5.1 Open Source Eucalyptus and Nimbus

Eucalyptus is a product from Eucalyptus Systems (www.eucalyptus.com) that was developed out of a research project at the University of California, Santa Barbara. Eucalyptus was initially aimed at bringing the cloud computing paradigm to academic supercomputers and clusters. Eucalyptus provides an AWS-compliant EC2-based web service interface for interacting with the cloud service. Additionally, Eucalyptus provides services, such as the AWS-compliant Walrus, and a user interface for managing users and images.

6.5.1.1 Eucalyptus Architecture

The Eucalyptus system is an open software environment. The architecture was presented in a Eucalyptus white paper [79,80]. Figure 3.31 introduced Eucalyptus from a virtual clustering point of view. Figure 6.26 shows the architecture based on the need to manage VM images. The system supports cloud programmers in VM image management as follows. Essentially, the system has been extended to support the development of both the computer cloud and storage cloud.

6.5.1.2 VM Image Management

Eucalyptus takes many design queues from Amazon's EC2, and its image management system is no different. Eucalyptus stores images in Walrus, the block storage system that is analogous to the Amazon S3 service. As such, any user can bundle her own root file system, and upload and then register this image and link it with a particular kernel and ramdisk image. This image is uploaded into a user-defined bucket within Walrus, and can be retrieved anytime from any availability zone. This allows users to create specialty virtual appliances (http://en.wikipedia.org/wiki/Virtual_appliance) and deploy them within Eucalyptus with ease. The Eucalyptus system is available in a commercial proprietary version, as well as the open source version we just described.

6.5.1.3 Nimbus

Nimbus [81,82] is a set of open source tools that together provide an IaaS cloud computing solution. Figure 6.27 shows the architecture of Nimbus, which allows a client to lease remote resources by deploying VMs on those resources and configuring them to represent the environment desired

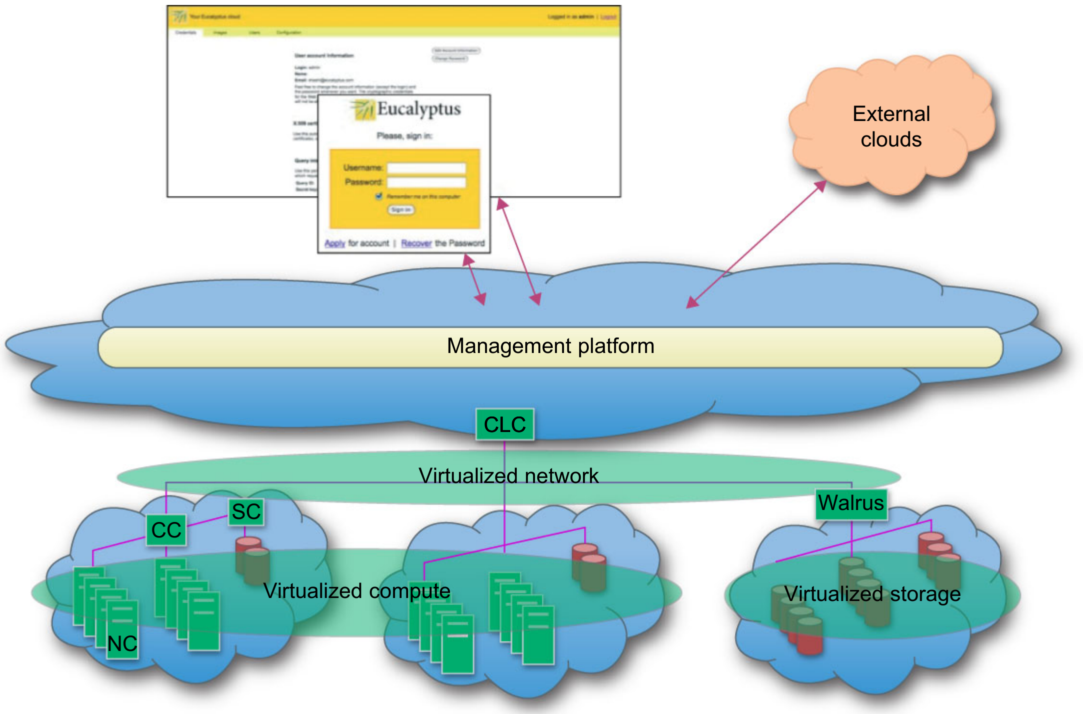


FIGURE 6.26

The Eucalyptus architecture for VM image management.

(*Courtesy of Eucalyptus LLC [81]*)

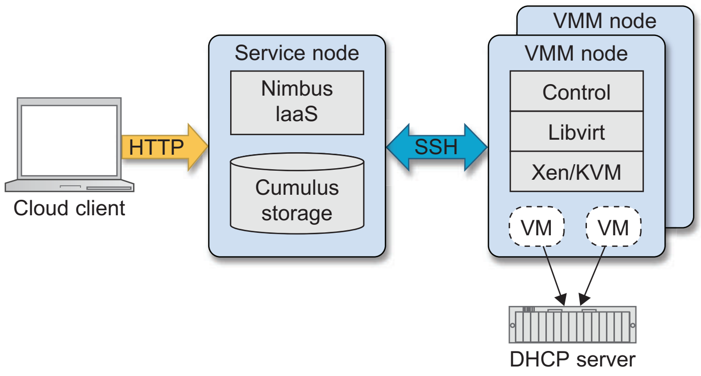


FIGURE 6.27

Nimbus cloud infrastructure.

(*Courtesy of Nimbus Project [82]*)

by the user. To this end, Nimbus provides a special web interface known as Nimbus Web [83]. Its aim is to provide administrative and user functions in a friendly interface. Nimbus Web is centered around a Python Django [84] web application that is intended to be deployable completely separate from the Nimbus service.

As shown in Figure 6.27, a storage cloud implementation called Cumulus [83] has been tightly integrated with the other central services, although it can also be used stand-alone. Cumulus is compatible with the Amazon S3 REST API [85], but extends its capabilities by including features such as quota management. Therefore, clients such as boto [86] and s2cmd [87], that work against the S3 REST API, work with Cumulus. On the other hand, the Nimbus cloud client uses the Java Jets3t library [88] to interact with Cumulus.

Nimbus supports two resource management strategies. The first is the default "resource pool" mode. In this mode, the service has direct control of a pool of VM manager nodes and it assumes it can start VMs. The other supported mode is called "pilot." Here, the service makes requests to a cluster's Local Resource Management System (LRMS) to get a VM manager available to deploy VMs. Nimbus also provides an implementation of Amazon's EC2 interface [89] that allows users to use clients developed for the real EC2 system against Nimbus-based clouds.

6.5.2 OpenNebula, Sector/Sphere, and OpenStack

OpenNebula [90,91] is an open source toolkit which allows users to transform existing infrastructure into an IaaS cloud with cloud-like interfaces. Figure 6.28 shows the OpenNebula architecture and its main components. The architecture of OpenNebula [92] has been designed to be flexible and modular to allow integration with different storage and network infrastructure configurations, and hypervisor technologies. Here, the core is a centralized component that manages the VM full life cycle, including setting up networks dynamically for groups of VMs and managing their storage requirements, such as VM disk image deployment or on-the-fly software environment creation.

Another important component is the capacity manager or scheduler. It governs the functionality provided by the core. The default capacity scheduler is a requirement/rank matchmaker. However, it is also possible to develop more complex scheduling policies, through a lease model and advance reservations [93]. The last main components are the access drivers. They provide an abstraction of the underlying infrastructure to expose the basic functionality of the monitoring, storage, and virtualization services available in the cluster. Therefore, OpenNebula is not tied to any specific environment and can provide a uniform management layer regardless of the virtualization platform.

Additionally, OpenNebula offers management interfaces to integrate the core's functionality within other data-center management tools, such as accounting or monitoring frameworks. To this end, OpenNebula implements the libvirt API [94], an open interface for VM management, as well as a command-line interface (CLI). A subset of this functionality is exposed to external users through a cloud interface. OpenNebula is able to adapt to organizations with changing resource needs, including addition or failure of physical resources [95]. Some essential features to support changing environments are live migration and VM snapshots [90].

Furthermore, when the local resources are insufficient, OpenNebula can support a hybrid cloud model by using cloud drivers to interface with external clouds. This lets organizations supplement

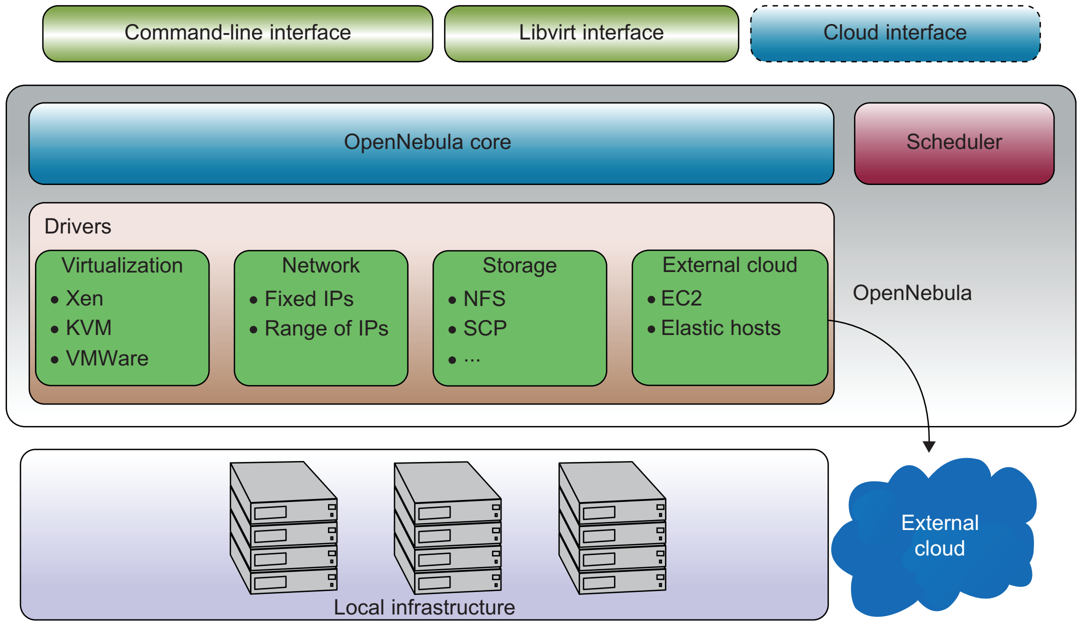


FIGURE 6.28

OpenNebula architecture and its main components.

(*Courtesy of Sotomayor, et al.* [94])

their local infrastructure with computing capacity from a public cloud to meet peak demands, or implement HA strategies. OpenNebula currently includes an EC2 driver, which can submit requests to Amazon EC2 [89] and Eucalyptus [80], as well as an ElasticHosts driver [96]. Regarding storage, an Image Repository allows users to easily specify disk images from a catalog without worrying about low-level disk configuration attributes or block device mapping. Also, image access control is applied to the images registered in the repository, hence simplifying multiuser environments and image sharing. Nevertheless, users can also set up their own images.

6.5.2.1 Sector/Sphere

Sector/Sphere is a software platform that supports very large distributed data storage and simplified distributed data processing over large clusters of commodity computers, either within a data center or across multiple data centers. The system consists of the Sector distributed file system and the Sphere parallel data processing framework [97,98]. Sector is a *distributed file system (DFS)* that can be deployed over a wide area and allows users to manage large data sets from any location with a high-speed network connection [99]. The fault tolerance is implemented by replicating data in the file system and managing the replicas.

Since Sector is aware of the network topology when it places replicas, it also provides better reliability, availability, and access throughout. The communication is performed using User Datagram Protocol (UDP) for message passing and user-defined type (UDT) [100] for data transfer.

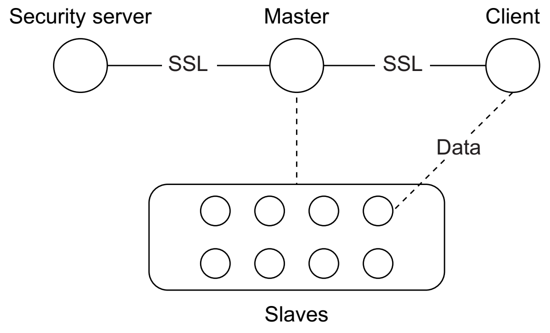


FIGURE 6.29

The Sector/Sphere system architecture.

(*Courtesy of Gu and Grossman [102]*)

Obviously, UDP is faster than TCP for message passing because it does not require connection setup, but it could become a problem if Sector is used over the Internet. Meanwhile, UDT is a reliable UDP-based application-level data transport protocol which has been specifically designed to enable high-speed data transfer over wide area high-speed networks [100]. Finally, the Sector client provides a programming API, tools, and a FUSE [101] user space file system module.

On the other hand, Sphere is a parallel data processing engine designed to work with data managed by Sector. This coupling allows the system to make accurate decisions about job scheduling and data location. Sphere provides a programming framework that developers can use to process data stored in Sector. Thus, it allows UDFs to run on all input data segments in parallel. Such data segments are processed at their storage locations whenever possible (data locality). Failed data segments may be restarted on other nodes to achieve fault tolerance. In a Sphere application, both inputs and outputs are Sector files. Multiple Sphere processing segments can be combined to support more complicated applications, with inputs/outputs exchanged/shared via the Sector file system [102].

The Sector/Sphere platform [102] is supported by the architecture shown in Figure 6.29, which is composed of four components. The first component is the security server, which is responsible for authenticating master servers, slave nodes, and users. We also have the master servers that can be considered the infrastructure core. The master server maintains file system metadata, schedules jobs, and responds to users' requests. Sector supports multiple active masters that can join and leave at runtime and can manage the requests. Another component is the slave nodes, where data is stored and processed. The slave nodes can be located within a single data center or across multiple data centers with high-speed network connections. The last component is the client component. This provides tools and programming APIs for accessing and processing Sector data.

Finally, it is worthy to mention that as part of this platform, a new component has been developed. It is called Space [97] and it consists of a framework to support column-based distributed data tables. Therefore, tables are stored by columns and are segmented onto multiple slave nodes. Tables are independent and no relationships between them are supported. A reduced set of SQL operations is supported, including, but not limited to, table creation and modification, key-value updates and lookups, and select UDF operations.

6.5.2.2 OpenStack

OpenStack [103] was been introduced by Rackspace and NASA in July 2010. The project is building an open source community spanning technologists, developers, researchers, and industry to share resources and technologies with the goal of creating a massively scalable and secure cloud infrastructure. In the tradition of other open source projects, the software is open source and limited to just open source APIs such as Amazon.

Currently, OpenStack focuses on the development of two aspects of cloud computing to address compute and storage aspects with the OpenStack Compute and OpenStack Storage solutions. "OpenStack Compute is the internal fabric of the cloud creating and managing large groups of virtual private servers" and "OpenStack Object Storage is software for creating redundant, scalable object storage using clusters of commodity servers to store terabytes or even petabytes of data." Recently, an image repository was prototyped. The image repository contains an image registration and discovery service and an image delivery service. Together they deliver images to the compute service while obtaining them from the storage service. This development gives an indication that the project is striving to integrate more services into its portfolio.

With neat diagram explain OpenStack Nova system architecture

6.5.2.3 OpenStack Compute

As part of its computing support efforts, OpenStack [103] is developing a cloud computing fabric controller, a component of an IaaS system, known as Nova. The architecture for Nova is built on the concepts of shared-nothing and messaging-based information exchange. Hence, most communication in Nova is facilitated by message queues. To prevent blocking components while waiting for a response from others, deferred objects are introduced. Such objects include callbacks that get triggered when a response is received. This is very similar to established concepts from parallel computing, such as "futures," which have been used in the grid community by projects such as the CoG Kit.

To achieve the shared-nothing paradigm, the overall system state is kept in a distributed data system. State updates are made consistent through atomic transactions. Nova it implemented in Python while utilizing a number of externally supported libraries and components. This includes boto, an Amazon API provided in Python, and Tornado, a fast HTTP server used to implement the S3 capabilities in OpenStack. Figure 6.30 shows the main architecture of Open Stack Compute. In this architecture, the API Server receives HTTP requests from boto, converts the commands to and from the API format, and forwards the requests to the cloud controller.

The cloud controller maintains the global state of the system, ensures authorization while interacting with the User Manager via Lightweight Directory Access Protocol (LDAP), interacts with

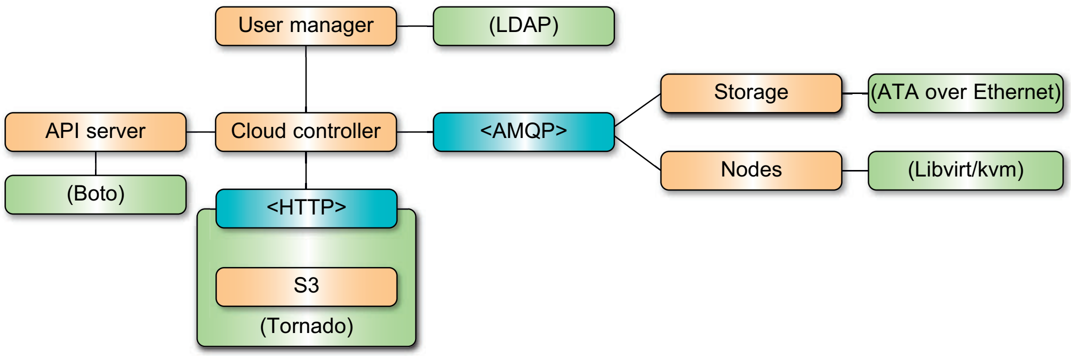


FIGURE 6.30

OpenStack Nova system architecture. The AMQP (Advanced Messaging Queuing Protocol) was described in Section 5.2.

the S3 service, and manages nodes, as well as storage workers through a queue. Additionally, Nova integrates networking components to manage private networks, public IP addressing, virtual private network (VPN) connectivity, and firewall rules. It includes the following types:

- *NetworkController* manages address and virtual LAN (VLAN) allocations
- *RoutingNode* governs the NAT (*network address translation*) conversion of public IPs to private IPs, and enforces firewall rules
- *AddressingNode* runs Dynamic Host Configuration Protocol (DHCP) services for private networks
- *TunnelingNode* provides VPN connectivity

The network state (managed in the distributed object store) consists of the following:

- VLAN assignment to a project
- Private subnet assignment to a security group in a VLAN
- Private IP assignments to running instances
- Public IP allocations to a project
- Public IP associations to a private IP/running instance

6.5.2.4 OpenStack Storage

The OpenStack storage solution is built around a number of interacting components and concepts, including a proxy server, a ring, an object server, a container server, an account server, replication, updaters, and auditors. The role of the proxy server is to enable lookups to the accounts, containers, or objects in OpenStack storage rings and route the requests. Thus, any object is streamed to or from an object server directly through the proxy server to or from the user. A ring represents a mapping between the names of entities stored on disk and their physical locations.

Separate rings for accounts, containers, and objects exist. A ring includes the concept of using zones, devices, partitions, and replicas. Hence, it allows the system to deal with failures, and isolation of zones representing a drive, a server, a cabinet, a switch, or even a data center. Weights can be used to balance the distribution of partitions on drives across the cluster, allowing users to support heterogeneous storage resources. According to the documentation, “the Object Server is a very simple blob storage server that can store, retrieve and delete objects stored on local devices.”

Objects are stored as binary files with metadata stored in the file’s extended attributes. This requires that the underlying file system is built around object servers, which is often not the case for standard Linux installations. To list objects, a container server can be utilized. Listing of containers is handled by the account server. The first release of OpenStack “Austin” Compute and Object Storage was October 22, 2010. This system has a strong developer community.

6.5.3 Manjrasoft Aneka Cloud and Appliances

Aneka (www.manjrasoft.com/) is a cloud application platform developed by Manjrasoft, based in Melbourne, Australia. It is designed to support rapid development and deployment of parallel and distributed applications on private or public clouds. It provides a rich set of APIs for transparently exploiting distributed resources and expressing the business logic of applications by using preferred programming abstractions. System administrators can leverage a collection of tools to monitor and control the deployed infrastructure. It can be deployed on a public cloud such as Amazon EC2 accessible through the Internet to its subscribers, or a private cloud constituted by a set of nodes with restricted access as shown in Figure 6.31.

Aneka acts as a workload distribution and management platform for accelerating applications in both Linux and Microsoft .NET framework environments. Some of the key advantages of Aneka over other workload distribution solutions include:

- Support of multiple programming and application environments
- Simultaneous support of multiple runtime environments
- Rapid deployment tools and framework
- Ability to harness multiple virtual and/or physical machines for accelerating application provisioning based on users' Quality of Service/service-level agreement (QoS/SLA) requirements
- Built on top of the Microsoft .NET framework, with support for Linux environments through Mono

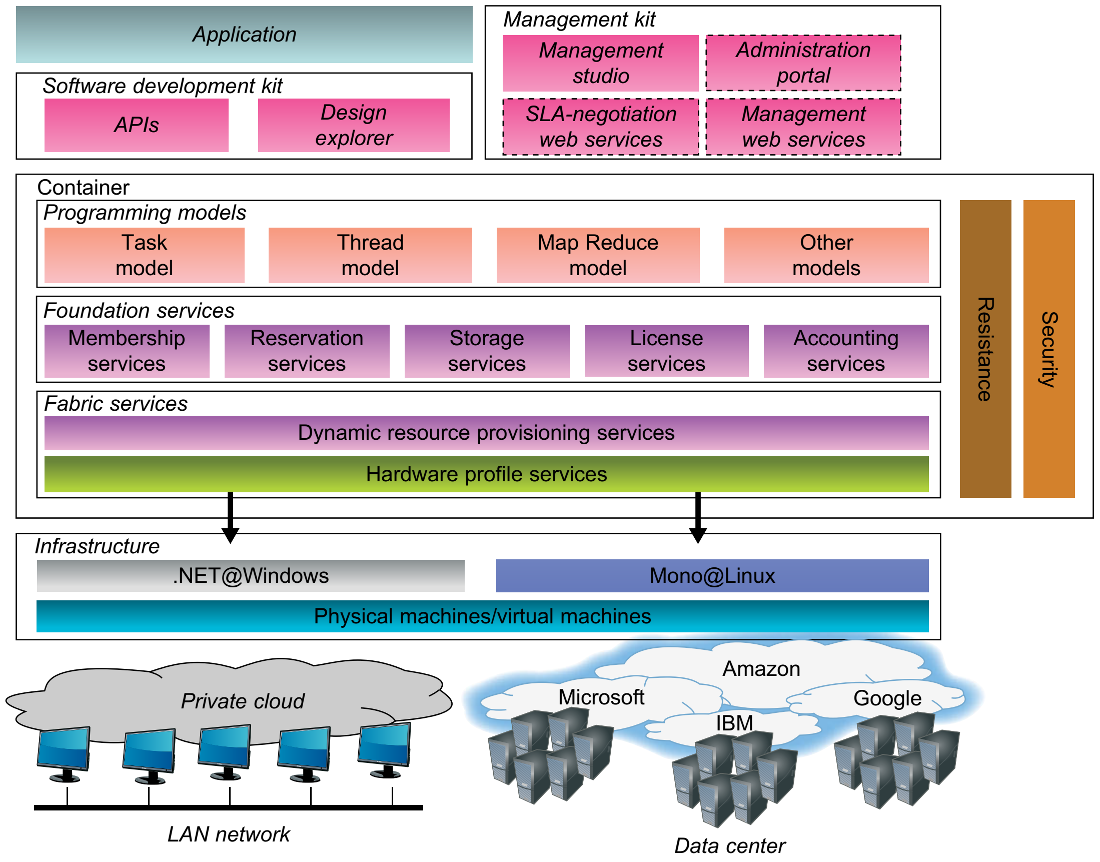


FIGURE 6.31

Architecture and components of Aneka.

(*Courtesy of Raj Buyya, Manjrasoft*)

Aneka offers three types of capabilities which are essential for building, accelerating, and managing clouds and their applications:

1. **Build** Aneka includes a new SDK which combines APIs and tools to enable users to rapidly develop applications. Aneka also allows users to build different runtime environments such as enterprise/private cloud by harnessing compute resources in network or enterprise data centers, Amazon EC2, and hybrid clouds by combining enterprise private clouds managed by Aneka with resources from Amazon EC2 or other enterprise clouds built and managed using XenServer.
2. **Accelerate** Aneka supports rapid development and deployment of applications in multiple runtime environments running different operating systems such as Windows or Linux/UNIX. Aneka uses physical machines as much as possible to achieve maximum utilization in local environments. Whenever users set QoS parameters such as deadlines, and if the enterprise resources are insufficient to meet the deadline, Aneka supports dynamic leasing of extra capabilities from public clouds such as EC2 to complete the task within the deadline (see Figure 6.32).

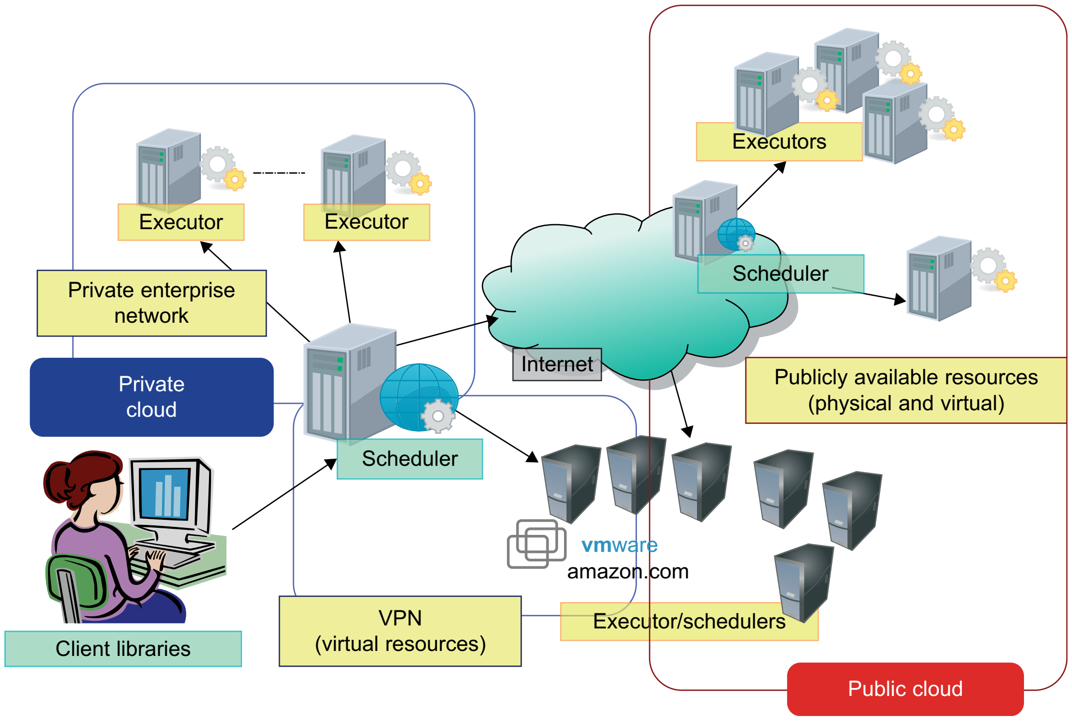


FIGURE 6.32

Aneka using private cloud resources along with dynamically leased public cloud resources.

(*Courtesy of Raj Buyya, Manjrasoft, www.manjrasoft.com/*)

3. **Manage** Management tools and capabilities supported by Aneka include a GUI and APIs to set up, monitor, manage, and maintain remote and global Aneka compute clouds. Aneka also has an accounting mechanism and manages priorities and scalability based on SLA/QoS which enables dynamic provisioning.

Here are three important programming models supported by Aneka for both cloud and traditional parallel applications:

1. Thread programming model, best solution to adopt for leveraging the computing capabilities of multicore nodes in a cloud of computers
2. Task programming model, which allows for quickly prototyping and implementing an independent bag of task applications
3. MapReduce programming model,as discussed in Section 6.2.2

6.5.3.1 Aneka Architecture

Aneka as a cloud application platform features a homogeneous distributed runtime environment for applications. This environment is built by aggregating together physical and virtual nodes hosting the Aneka container. The container is a lightweight layer that interfaces with the hosting environment and manages the services deployed on a node. The interaction with the hosting platform is mediated through the *Platform Abstraction Layer (PAL)*, which hides in its implementation all the heterogeneity of the different operating systems.

By means of the PAL it is possible to perform all the infrastructure-related tasks, such as performance and system monitoring. These activities are vital to ensure the desired QoS for applications. The PAL, together with the container, represents the hosting environment of services which implement the core capabilities of the middleware and make it a dynamically composable and extensible system. The available services can be aggregated into three major categories:

Fabric Services: Fabric services implement the fundamental operations of the infrastructure of the cloud. These services include HA and failover for improved reliability, node membership and directory, resource provisioning, performance monitoring, and hardware profiling.

Foundation Services: Foundation services constitute the core functionalities of the Aneka middleware. They provide a basic set of capabilities that enhance application execution in the cloud. These services provide added value to the infrastructure and are of use to system administrators and developers. Within this category we can list storage management, resource reservation, reporting, accounting, billing, services monitoring, and licensing. Services in this level operate across the range of supported application models.

Application Services: Application services deal directly with the execution of applications and are in charge of providing the appropriate runtime environment for each application model. They leverage foundation and fabric services for several tasks of application execution such as elastic scalability, data transfer, and performance monitoring, accounting, and billing. At this level, Aneka expresses its true potential in supporting different application models and distributed programming patterns.

Each supported application model is managed by a different collection of services that interact with the underlying layers and services to carry out application execution. In general, the middleware counterpart of each application model features at least two different services: scheduling and execution. In addition, specific models can require additional services or a different type of support. Aneka provides support for the most well-known application programming patterns, such as distributed threads, bags of tasks, and MapReduce.

Additional services can be designed and deployed in the system. This is how the infrastructure is enriched with additional features and capabilities. The SDK provides straightforward interfaces and ready-to-use components for rapid service prototyping. Deployment and integration of new services is painless and immediate: The container leverages the Spring framework and allows for dynamic integration of new components such as services.

Example 6.11 Aneka Application of Maya Rendering Case Study

Aneka has been used to create several interesting applications in domains such as life sciences, engineering, and creative media. Applications created using Aneka are able to run on enterprise or public clouds unchanged. We will briefly discuss a case study on high-speed rendering of engineering designs using Aneka software. To reduce this time, GoFront has used Aneka and created an enterprise cloud (Figure 6.33) within its company by utilizing networked PCs. GoFront Group, a unit of China Southern Railways, is China's premier and largest nationwide researcher and manufacturer of rail electric traction equipment.

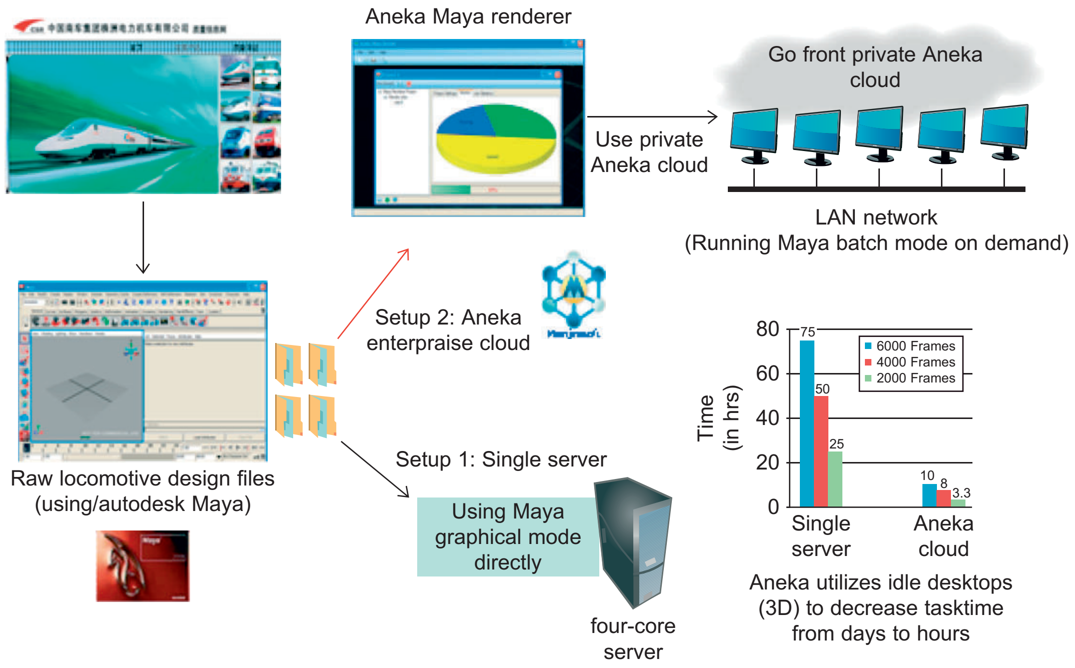

Aneka Maya renderer
Go front private Aneka cloud
Use private Aneka cloud
LAN network
(Running Maya batch mode on demand)
Setup 2: Aneka enterpraise cloud
Setup 1: Single server
Using Maya graphical mode directly
four-core server
Raw locomotive design files (using/autodesk Maya)
Time (in hrs)
80
60
40
20
0
75
50
25
10
8
3.3
6000 Frames
4000 Frames
2000 Frames
Single server
Aneka cloud
Aneka utilizes idle desktops (3D) to decrease tasktime from days to hours

FIGURE 6.33

Rendering images of locomotive design on GoFront's private cloud using Aneka.

(*Courtesy of Raj Buyya, Manjrasoft*)

The GoFront Group is responsible for designing the high-speed electric locomotive, metro car, urban transportation vehicle, and motor train. The raw design of the prototypes requires high-quality 3D images using Autodesk's Maya rendering software. By examining the 3D images, engineers identify problems in the original design and make the appropriate design improvements. However, such designs on a single four-core server took three days to render scenes with 2,000 frames.

To reduce this time, GoFront used Aneka and created an enterprise cloud by utilizing networked PCs. It used Aneka Design Explorer, a tool for rapid creation of parameter sweep applications in which the same program is executed many times on different data items (in this case, executing Maya for rendering different images). A customized Design Explorer (called Maya GUI) has been implemented for Maya rendering. Maya GUI managed parameters, generated Aneka tasks, monitored submitted Aneka tasks, and collected final rendered images. The design image used to take three days to render (2,000+ frames, each frame with more than five different camera angles). Using only a 20-node Aneka cloud, GoFront was able to reduce the rendering time from three days to three hours.

■

6.5.3.2 Virtual Appliances

Machine virtualization offers a unique opportunity to break software dependencies between applications and their hosting environments. In recent years, resource virtualization has been witnessing a renaissance fueled by efficient, freely available VM monitors for commodity systems (e.g., Xen, VMware Player, KVM, and VirtualBox) and backed by the microprocessor industry (e.g., Intel and AMD virtualization extensions). Modern system VMs provide improved flexibility, security, isolation, and resource control, while supporting a wealth of unmodified applications, making a compelling case for their use in grid computing. Grid applications also require network connectivity, and the increasing use of NAT and IP firewalls breaks the original model of each node in the Internet being a peer, and is recognized as a hindrance to programming and deploying grid computing systems.

In Aneka, VMs and P2P network virtualization technologies can be integrated into self-configuring, prepackaged "virtual appliances" to enable simple deployment of homogeneously configured virtual clusters across a heterogeneous, wide-area distributed system. Virtual appliances are VM images that are installed and configured with the entire software stack (including OS, libraries, binaries, configuration files, and auto-configuration scripts) that is required for a given application to work "out-of-the-box'' when the virtual appliance is instantiated. Compared to traditional approaches of software distribution, virtual appliances have the benefit of significantly reducing software dependence on the hosting environment.

VM conversion tools exist for most VMMs and standardization efforts that are underway will further facilitate interoperation across VMMs, such that appliance instantiation across multiple platforms should be seamless. Wide-area overlays of virtual appliances aggregate the capacity of commodity hardware and can be programmed and managed as a local area network—even though nodes may be distributed across multiple network domains. The preconfigured Grid Appliance image (www.grid-appliance.org) can encapsulate complex distributed system software in a manner that is transparent to users for easy deployment. Grid appliances run on free VMMs available for modern commodity servers and desktops.