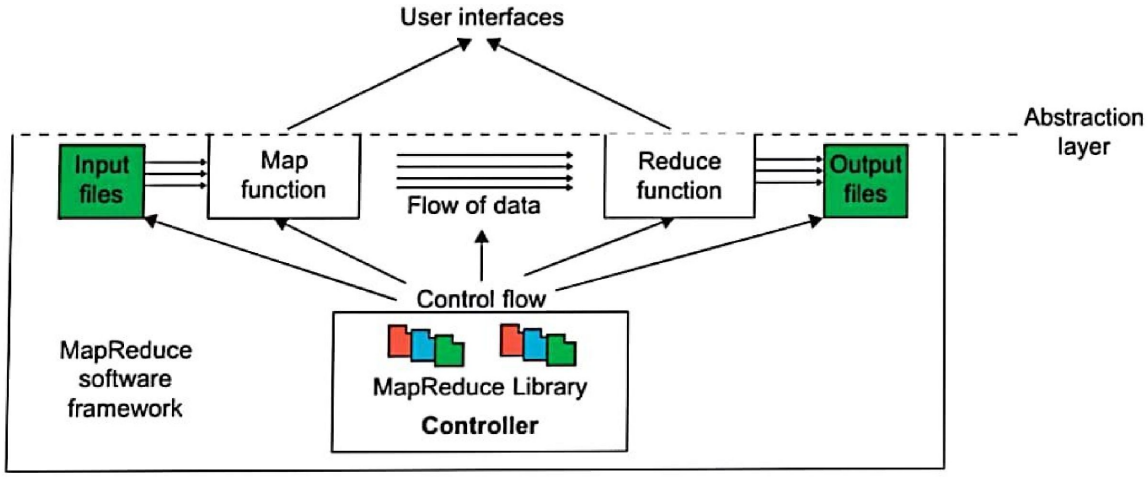
6.2.1 Parallel Computing and Programming Paradigms

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.

6.2.2 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,



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.

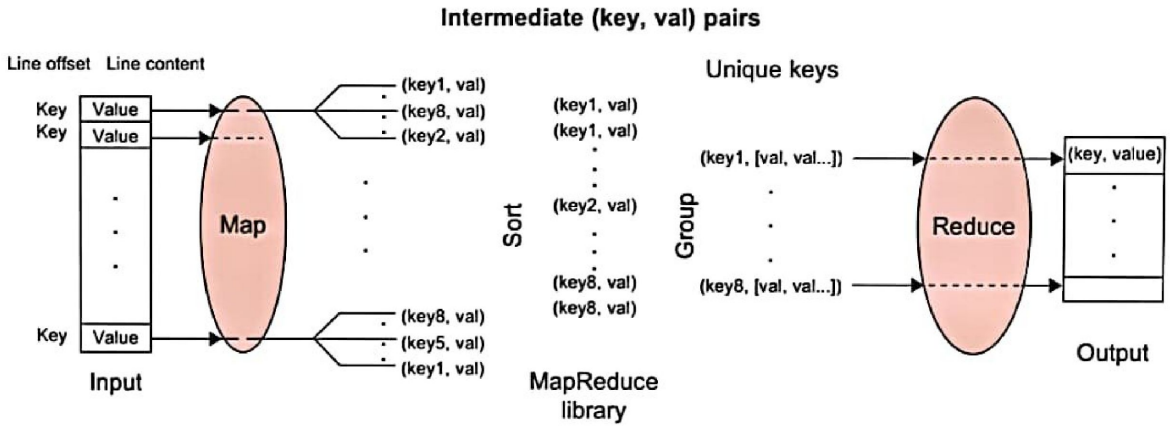
The overall structure of a user's program containing the Map, Reduce, and the Main functions is given below. The Map and Reduce are two major subroutines. They will be called to implement the desired function performed in the main program.

```
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

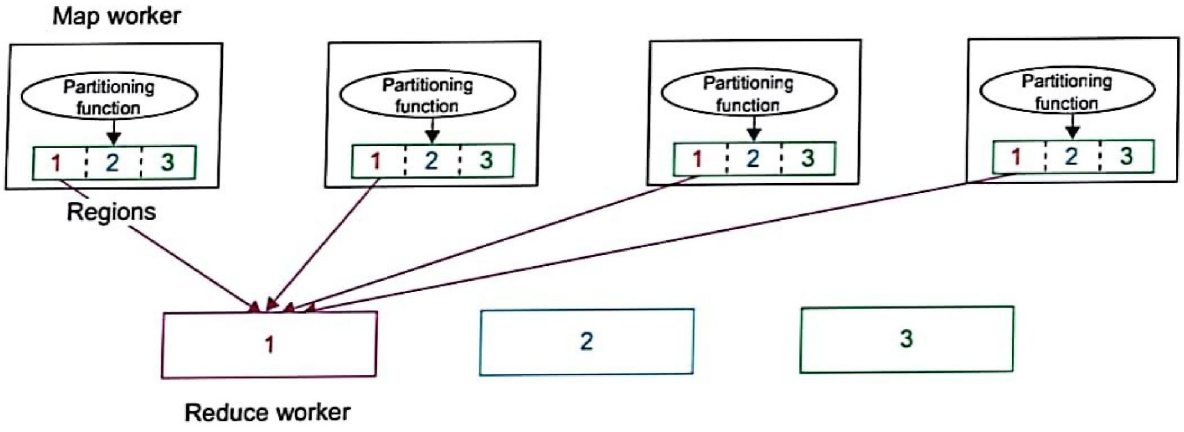


6.2.2.5 MapReduce Actual Data and Control Flow

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.



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.

6.2.2.1 Architecture of MapReduce in Hadoop

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

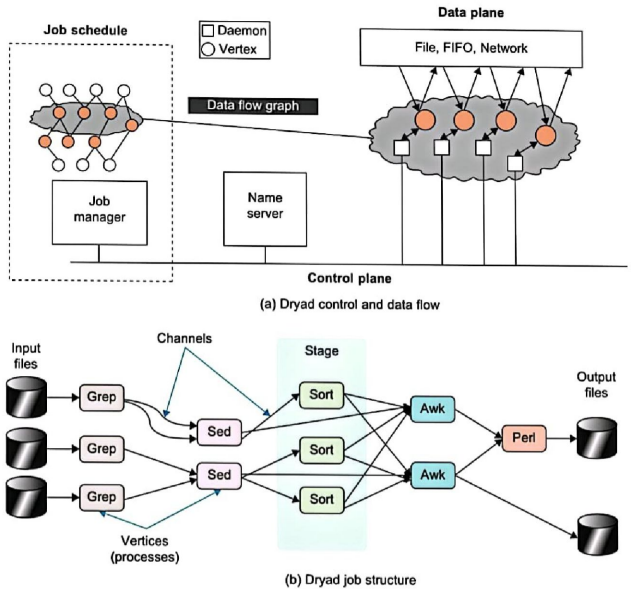


FIGURE 6.13

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.

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.

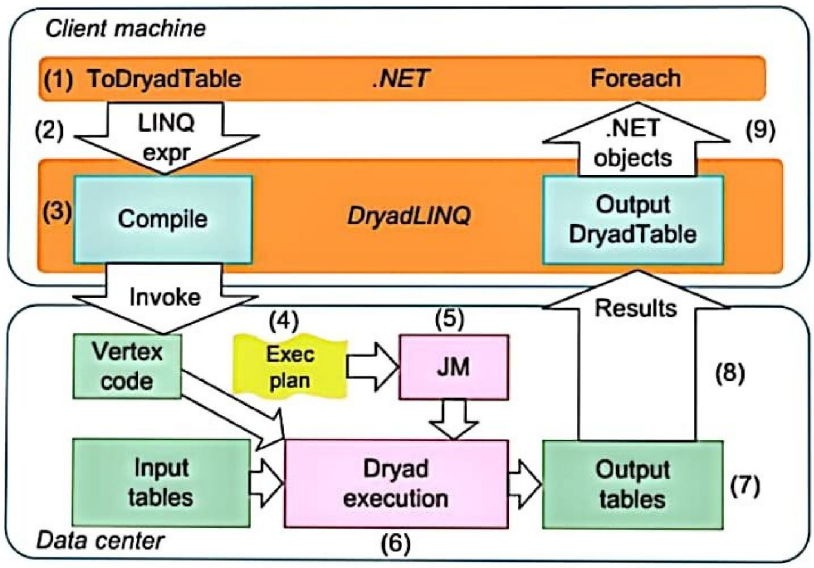


FIGURE 6.14

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.

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.

—

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

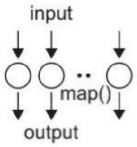
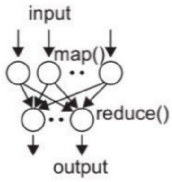
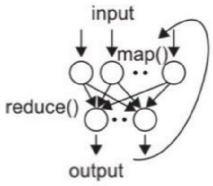
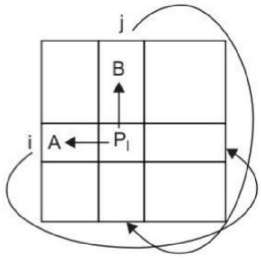
	Sawzall	Pig Latin	DryadLINQ
Origin	Google	Yahoo!	Microsoft
Data Model	Google protocol buffer or basic	Atom, Tuple, Bag, Map	Partition file
Typing	Static	Dynamic	Static
Category	Interpreted	Compiled	Compiled
Programming Style	Imperative	Procedural: sequence of declarative steps	Imperative and declarative
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

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

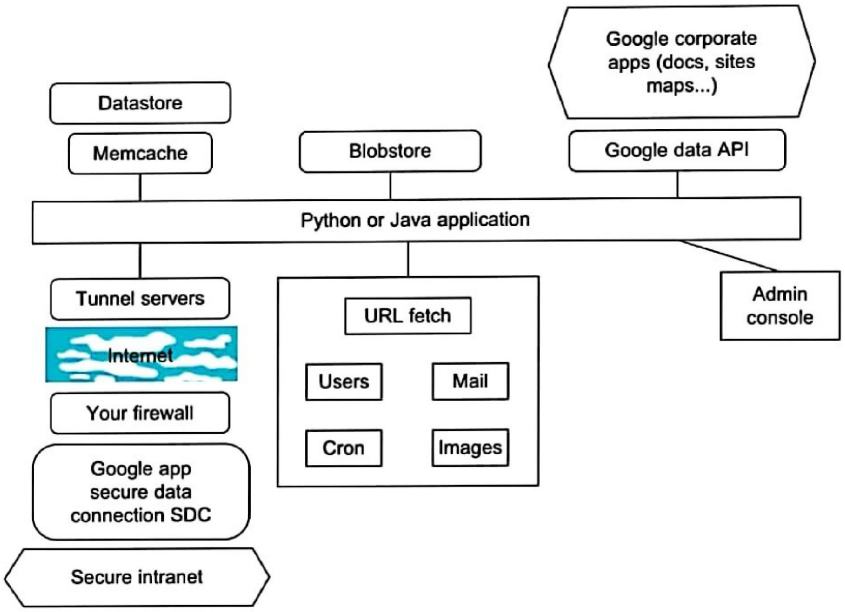


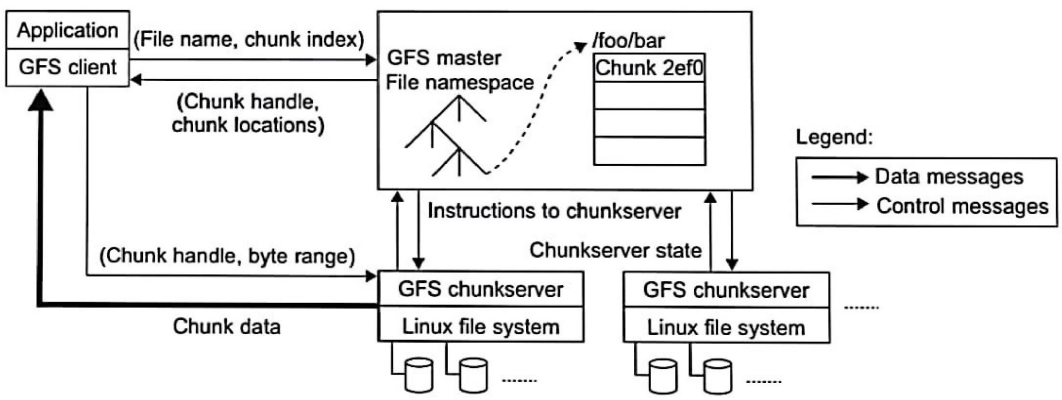
FIGURE 6.17

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.



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.

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

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.

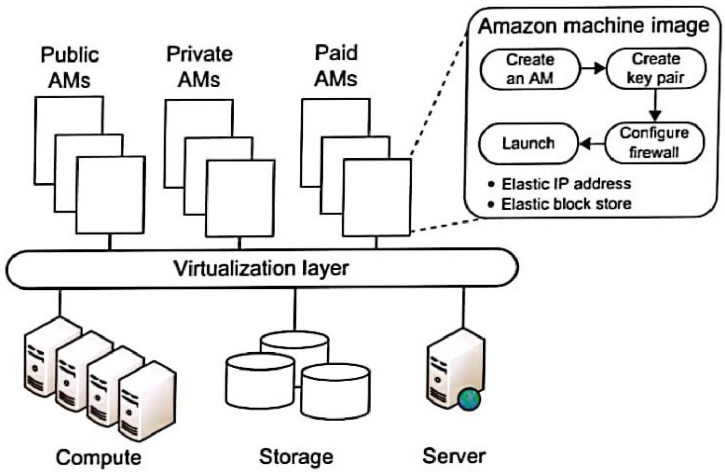
6.4.1 Programming on Amazon EC2

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.



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.
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).

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

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.

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

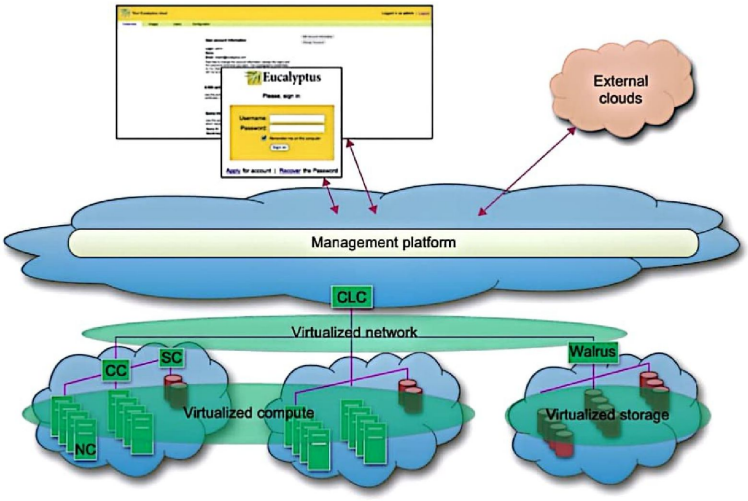


FIGURE 6.26

The Eucalyptus architecture for VM image management.

(Courtesy of Eucalyptus LLC [81])

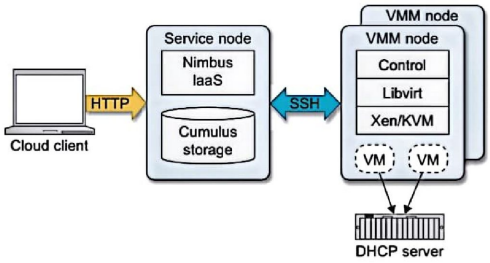


FIGURE 6.27

Nimbus cloud infrastructure.

(Courtesy of Nimbus Project [82])

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