8

Programming Models for Cloud Computing

Learning Objectives

The objectives of this chapter are to

- Give a brief description about the programming models available in cloud
- Point out the advantages and disadvantages of each programming model
- Point out the characteristic of each programming model
- Describe the working base of each programming model
- Point out the key features of the programming model
- Discuss the suitability of each programming model to different kinds of application

Preamble

Cloud computing is a broad area that has garnered the attention of several people and is now much more than a buzzword. This technology is one of the few that have directly affected human beings. In simple terms, a cloud offers services to its customers in several ways. One of the important ways is through Software as a Service (SaaS). In SaaS, a software application is given as a service to the users. There are several properties associated with *cloud* applications: scalability, elasticity, and multitenancy. Each and every cloud application should have the aforementioned properties.

This is what differentiates cloud from conventional applications. It is a known fact that any application involves a specific development strategy. In general, all development processes are characterized by the use of a specific programming model chosen based on the type of application we develop. The properties

of application play a very important role in choosing a programming model. Thus, the development of this cloud application faces certain challenges.

Cloud computing is an emerging technology, and there are not many programming languages developed for it. This chapter broadly discusses the programming models available in cloud. Based on the approach or the method, the programming models can be classified into two. The first is using an existing programming model and extending it or modifying it to suit the cloud platform, and the second one is by developing a whole new model that will be suitable for cloud. These will be discussed in two separate sections. The first section of this chapter gives a brief introduction about programming models. It also discusses about cloud computing, its properties, and the impact that this field has on the technological world and the market. This section also discusses the differences between a usual application and a cloud application, as well as the big reason behind the quest for a new programming model. The existing programming models and languages used are discussed first, followed by the new programming models that are specifically designed for cloud. The chapter ends with a summary and some review questions.

8.1 Introduction

Programming model is a specific way or a method or approach that is followed for programming. There are several programming models available in general, each having their own advantages and disadvantages. Programming models form the base for any software or application development approach. The properties of these programming models decide their usage and their impact. As far as general programming models are concerned, these have continuously evolved. Each evolutionary step is characterized by certain changes to the existing methodology, and at each level, a new functionality is added, which facilitates the programmers in one or several ways. As the technology is growing, so are the problems. One programming model cannot be the solution for all the problems, even with a large number of advancements. In these cases, a whole new programming model is developed for a specific set of problems or for general use. To address an issue, any of the two approaches can be used.

Today, cloud computing is one of the most popular and extensively used technologies. As a technology, cloud is considered to be good and is popular because it allows users to primarily use computing, storage, and networking services on demand without spending for infrastructure (capital expenditure). An extensive research is going on in this segment. Many business organizations have started depending on cloud. The fast development of cloud was driven by its market-oriented nature.

Several big companies like Amazon, Google, and Microsoft have started using cloud extensively, and they intend to continue this trend.

With the advent of cloud computing, there came a new shift in the computing era. All the applications are being migrated to cloud as the future seems to be fully dependent on it. The service architecture of cloud has three basic components called service models, namely Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Among these three, SaaS is the most important model where software is given as a service from cloud. This software is specifically designed for cloud platform, and for designing a software or application, you have to use a specific programming model. Thus, a study about it is important.

Cloud has several properties that make it totally different from other conventional software applications, such as scalability, concurrency, multitenancy, and fault tolerance. There are generally two types of practices: one is extending the existing programming models or languages to cloud, and the other is to have a completely new model specifically designed for cloud. The programming models under these two areas are broadly discussed.

8.2 Extended Programming Models for Cloud

This section discusses the existing programming models that can be extended to a cloud platform. Not all programming models can be migrated to cloud. As discussed, there are several properties of cloud that make it a different technology from others. Thus, certain changes have to be made to address the issues of the cloud. Only certain programming models that have some similarity to cloud or have a structure that can eventually support certain parameters like scalability, concurrency, and multitenancy can be extended. Out of these properties, scalability and concurrency are considered to be very important for cloud. Scalability is the ability of the system to scale itself up or down according to the number of users. This is one of the reasons behind the popularity of cloud. Thus, this property should be taken care of by any programming model. Some of the programming models and frameworks that are suitable for cloud, including certain programming languages, are discussed in the following subsections.

8.2.1 MapReduce

MapReduce is a parallel programming model developed by Google [1]. This is one of the technologies that has a specific impact on the area of parallel computing as well as cloud computing. MapReduce uses a parallel programming paradigm for computation of a large amount of data by partitioning or segregating them into several pieces, processing them in parallel, and

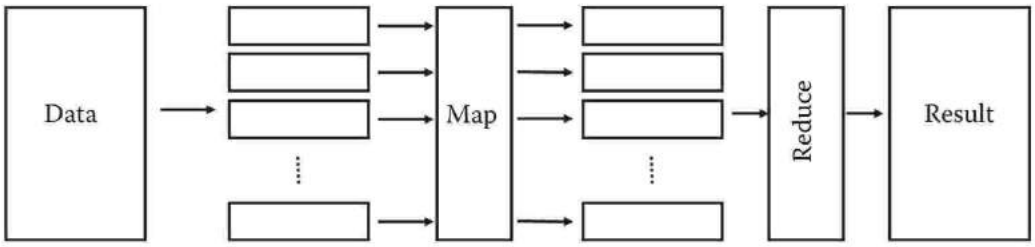

FIGURE 8.1
MapReduce.

then combining them to generate a single result. These processes are called map and reduce, respectively. The popular PaaS called Aneka also uses Map Reduce with .NET platform [2].

The first step in the map and reduce functions involves segregation of large data and sending it to map function. The map function performs map operation on data. Subsequently, the data are sent for reduce operation where the actual results are obtained. This is discussed in detail in the following subsections and shown in Figure 8.1.

8.2.1.1 Map Function

First, the large data are traced and segregated as key/value pairs. The map function accepts the key/value pair and returns an intermediate key/value pair. Usually, a map function works on a single document. The following formula depicts the exact operation:

$$Map(Key1,\ Val1) \rightarrow List(Key2,\ Val2)$$

The list of key/value pair obtained is subsequently sent to the reduce function.

8.2.1.2 Reduce Function

Once the list of key/value pair is prepared after the map functions, a sort and merge operation is applied on these key/value pairs. Usually, reduce function works on a single word. Input of distinct key/value pair is given and in return the group with the same key/value pair is obtained, as shown in the following equation:

$$Reduce(Key2, List(Val2)) \rightarrow List(Val3)$$

The resultant values give the final set of result.

The most popular example is counting the words in a file.

In the first step, the words (key) are assigned the values. The word represents the key, and the count represents the value. Whenever the word is encountered, the key/value pair is generated. This set of <word, count> pair

obtained from the map function is sent to the reduce function. The result is sorted, and the reduce function combines all the words that have the same key and the *count* (value) is automatically incremented (added) for each pair obtained. Thus, the result gives each word with its count.

Key features

- Supports parallel programming
- Fast
- Can handle a large amount of data

8.2.2 CGL-MapReduce

CGL-MapReduce is another type of MapReduce runtime and was developed by Ekanayake et al. [3]. There are specific differences between CGL-MapReduce and MapReduce. Unlike MapReduce, CGL-MapReduce uses streaming instead of a file system for all communications. This eliminates the overheads associated with file communication, and the intermediate results from the map functions are directly sent. The application was primarily created and tested for a large amount of scientific data and was compared with MapReduce. Figure 8.2 shows the working architecture of CGL-MapReduce where its several components are depicted.

1. *Map worker*: A map worker is responsible for doing map operation.
2. *Reduce worker*: A reduce worker is responsible for doing reduce operation.

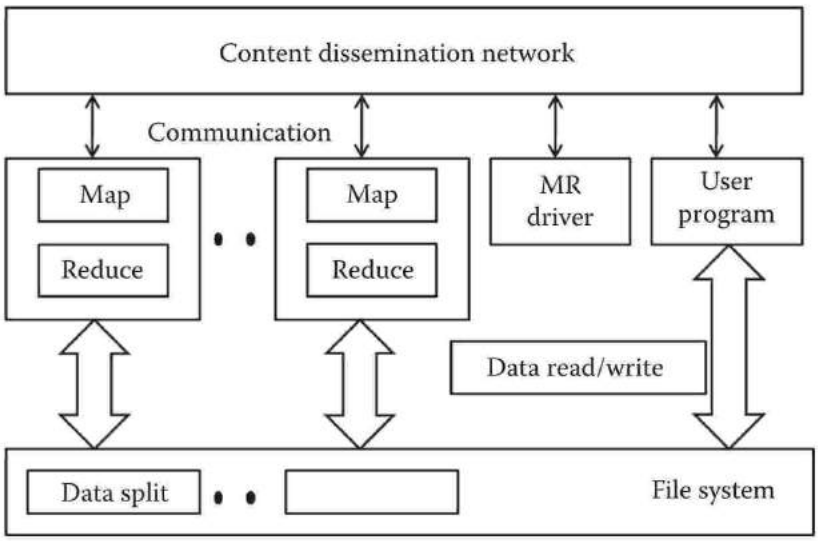

FIGURE 8.2
CGL-MapReduce. (Adapted from Ekanayake, J. et al., Mapreduce for data intensive scientific analyses, *IEEE Fourth International Conference on eScience'08, 2008 (eScience'08)*, Indianapolis, IN, IEEE, Washington, DC, 2008.)

3. *Content Dissemination Network*: Content dissemination network handles all the communication between the components using the NaradaBroker, which was developed by Ekanayake et al. [3].
4. *MRDriver*: MRDriver is a master worker and controls the other workers based on the instructions by the user program. CGL-MapReduce is different from MapReduce, the main difference being the avoidance of file system and usage of streaming. Several steps involved in CGL-MapReduce are depicted in Figure 8.3 and explained as follows [3]:
   a. *Initializing stage*: The first step involves starting the MapReduce worker nodes and configuration of the MapReduce task. Configuration of the MapReduce is a onetime process and multiple copies of it can be reused. This is one of the improvements of CGL-MapReduce, which facilitates efficient iterative MapReduce computations. These configured MapReduce are stored and executed upon the request of the user. Here, fixed data are given as input.
   b. *Map stage*: After the initialization step, MRDriver starts the map computation upon the instruction of the programmer. This is done by passing the variable data to the map tasks. This is

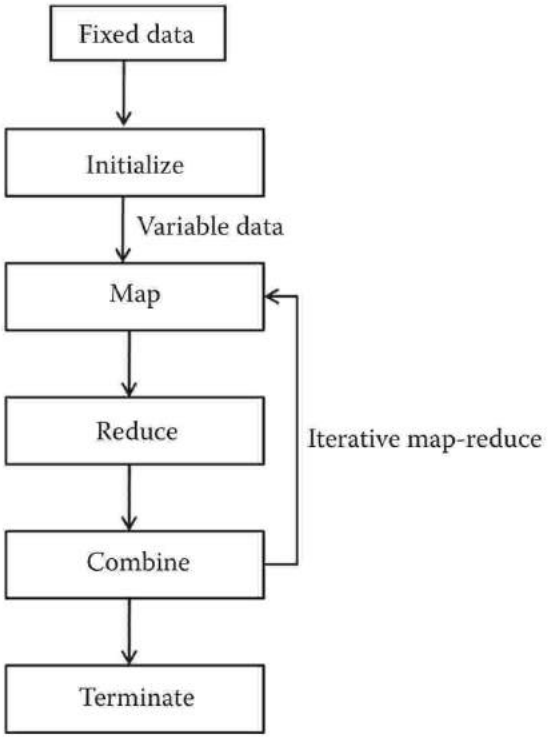

FIGURE 8.3
Steps involved in CGL-MapReduce. (Adapted from Ekanayake, J. et al., MapReduce for data intensive scientific analyses, *IEEE Fourth International Conference on eScience'08, 2008 (eScience'08)*, Indianapolis, IN, IEEE, Indianapolis, IN, 2008.)

relayed to workers for invoking configured map tasks. It also allows passing the results from one iteration to another. Finally, the map tasks are transferred directly to reduce workers using dissemination network.

c. *Reduce stage*: As soon as all the map tasks are completed, they are transferred to reduce workers, and these workers start executing tasks after they are initialized by the MRDriver. Output of the reduce function is directly sent to the user application.

d. *Combine stage*: In this stage, all the results obtained in the reduce stage are combined. There are two ways of doing it: if it is a single-pass MapReduce computation, then the results are directly combined, and if it is iterative operation, then appropriate combination is obtained such that the iteration continues successfully.

e. *Termination stage*: This is the final stage, and user program gives the command for termination. At this stage, all the workers are terminated [3].

Key features

- Uses streaming for communication
- Supports parallelization
- Iterative in nature
- Can handle a large amount of data

8.2.3 Cloud Haskell: Functional Programming

Cloud Haskell is based on the Haskell functional programming language. These functional programming languages are function based and work like mathematical functions. Usually, the output here is based on the input value only. Functional programming consists of components that are often immutable. Here, in Cloud Haskell, the immutable components form the basis. Immutable components are the components whose states cannot be modified after they have been created. Cloud Haskell is a domain-specific language [4].

The programming model is completely based on message-passing interface, which can be used for highly reliable real-time application. The main aim is to use concurrent programming paradigm. The Cloud consists of several applications and users who are independent and are large in numbers. Thus, to address the issue, concurrency has to be ensured for a large number of requests. Cloud Haskell provides the concurrency features with complete

independence for the application. The data are completely isolated and the applications cannot breach the boundary and access other data. As mentioned earlier, the communication is completely based on message-passing model. According to the developers, there needs to be a separate cost model that would determine the data movement. There are several advantages of Haskell apart from the use of Erlang's fault tolerance model. Its main advantage is that it is a pure functional language. All the data are immutable by default and functions are idempotent. Idempotency here refers to the ability of the functions to restart from any points on its own without the use of any external entity such as a distributed database when the function is aborted due to hardware error.

A cost model has been introduced specifically in Cloud Haskell, which was not present in Haskell. Here, the cost model determines the communication cost between the processes. This model is not designed to have a shared memory. As the model supports concurrency and these processes are isolated, fault tolerance is expected to an extent and the failure of one process would not affect others. There are several other novel features that are included into this model such as serialization. Whenever a programmer tries to use the distributed code, he had to run the code in a remote system. This was not possible in Haskell versions. Thus, in this model, the developers introduce a mechanism that will automatically take care of this issue without extending the compiler.

In simple terms, the serialization is taken care of with full abstraction so that the programmer does not know it. The model also takes care of failure, which is an important issue in cloud. There are many ways in which failure might affect the functioning of the system. As the number of systems connected and users increase, failure rate also increases. Usually, to confront partial failure, the whole system is restarted. This is considered as one of the inefficient solutions as when it comes to cloud the number of distributed nodes is high, and so restarting a system can prove to be very costly. Thus, to maintain fault tolerance, Erlang's method is used. Even if a function fails because of certain reasons, it can be separately restarted without affecting the other parts and processes. Here, the components are immutable and otherwise called pure functions [4].

Key features

- Fault tolerant
- Domain-specific language
- Uses concurrent programming paradigm
- No shared memory
- Idempotent
- Purely functional

8.2.4 MultiMLton: Functional Programming

MultiMLton is specifically used for multiprocessor environments and is an extension of MLton. MLton is an open-source, whole-program, optimizing standard ML compiler [5]. It is used for expressing and implementing various kinds of fine-grained parallelism. It has the capability to efficiently handle a large number of lightweight threads. It also combines new language abstractions and compiler analyses. There are several differences between MultiMLton and its other counterparts like Erlang and ConcurrentML. It provides additional parallelism whenever possible and whenever the result is profitable by using deterministic concurrency within threads. This increases speed and will eventually give good performance. It also provides message-passing aware composable speculative actions [5]. It provides isolation among groups of communicating threads [5]. The communication between the threads happens through simple message-passing techniques. It allows construction of asynchronous events that integrate abstract synchronous communication protocols.

The MultiMLton model consists of a component that is known as a parasite. These parasites are mini threads that depend on a main or master thread for its execution. As mentioned earlier, communication between threads can be synchronous and asynchronous. Figure 8.4 shows a simple communication between the parasite of one thread and another thread.

Key features

- Can efficiently handle a large number of lightweight threads.
- Suitable for multiprocessor environment.
- Deterministic concurrency is used wherever necessary.

8.2.5 Erlang: Functional Programming

Erlang is one of the important types of functional programming language. It was developed by Joe Armstrong and is one of the very few languages that are highly fault tolerant. Even Haskell (another programming language) uses Erlang's fault tolerance model [4].

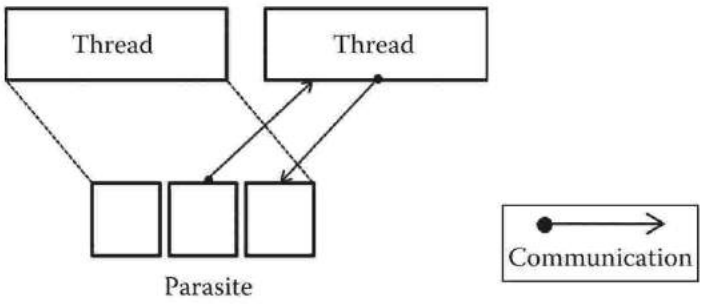

FIGURE 8.4
Message passing.

Erlang is a concurrent programming language that uses asynchronous message-passing techniques for communication. Here, communication can be done only through a message-passing mechanism. Creating and deleting processes are very easy, and message-passing processors require less memory; hence, it is considered as lightweight. It follows a process based on a model of concurrency.

This programming model is used to develop real-time systems where timing is very critical. Fault tolerance is one of the important properties of Erlang. It does not have a shared memory as mentioned earlier; all the interactions are done through message passing. It has good memory management system and allocates and deallocates memory automatically with dynamic (runtime) garbage collection. Thus, a programmer need not worry about the memory management, and thereby memory-related errors can be avoided. One of the most striking properties of Erlang is that it allows code replacement in the runtime. A user can change the code while a program is running and can use the same new code simultaneously.

Erlang, by definition, is a declarative language. It can also be regarded as a functional programming language. Functional programming languages are the languages that depend only on input data. This property is usually called immutability. This is the base property that makes functional programming the most suitable programming model for cloud. These properties allow the programmers to use concurrency and to scale the system, which is one of the important properties of cloud.

Key features

- Fault tolerant
- Robust
- Concurrent
- Uses functional programming model

8.2.5.1 CloudI

CloudI is an open-source cloud computing framework developed by Michael Truog. It allows building a cloud platform (a private cloud) without actual virtualization. It is basically for back-end server processing tasks. These tasks require soft real-time transaction processing external to database usage in any language such as C, C++, Python, Java, Erlang, and Ruby [6]. CloudI is built over the Erlang programming language. It is aimed at being fault tolerant and has a highly scalable architecture (a cloud at the lowest level; bringing Erlang's fault tolerance to polyglot development). It consists of a messaging application programming interface (API) that allows CloudI services to send requests.

CloudI API supports both publish/supply and request/reply messages. Apart from fault tolerance, which is one of its important properties, CloudI can easily support other languages or a polyglot environment. It is not necessary for a user

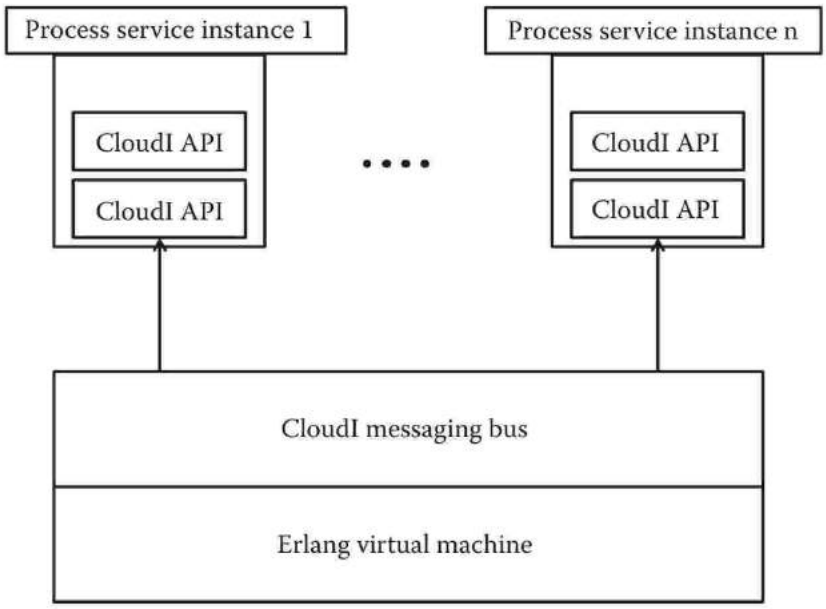

FIGURE 8.5
CloudI communication. (Adapted from Bringing Erlang's fault-tolerance to polyglot development, available [online]: http://www.toptal.com/erlang/a-cloud-at-the-lowest-level-built-in-erlang.)

to know Erlang completely to work with CloudI. Any service module in CloudI is a miniature of the central module. Each service module contains CloudI API.

Here, all the communications is done through CloudI API. Figure 8.5 depicts the interrelations and the communication between the CloudI service instance and process service instances. The CloudI messaging bus acts as an interface between the APIs and the Erlang VM. Each process service consists of CloudI API and is responsible for communication.

Key features

- Uses functional programming model
- Fault tolerant
- Supports many languages
- Supports extensive scalability

8.2.6 SORCER: Object-Oriented Programming

Service-ORiented Computing EnviRonment (SORCER) is an object oriented cloud platform for transdisciplinary service abstractions proposed by Michael Sobolewski, which also supports high-performance computing. It is a federated service-to-service (S2S) metacomputing environment that treats service providers as network peers with well-defined semantics of a service object-oriented architecture [8].

Transdisciplinary computing consists of several service providers. All service providers have a collaborative federation, and they use an orchestrated

workflow for their services. As soon as the work is over, the collaboration is dissolved and the service providers search for other federations to join. This method is wholly network based. Here, each service is an independent and self-sustaining entity called remote service provider, which performs specific network tasks. Thus, the whole method is network centric. For controlling the whole system, an operating system (OS) is required, which will enable the providers to give instructions related to interaction between services.

The users do not use this technology as for managing the orchestration and the interaction between the services. The users have to know command line codes and script, thereby making it a tedious job. Hence, to solve this problem, service-oriented operating system (SOOS) is used. Here, metaprogram is considered as a process expression of remote component programs, and the SOOS makes all the decisions about place, time, and other details regarding running the remote component program. The metaprograms are programs that remotely manipulate the other programs as its data. Here, it is used as a specification for service collaboration.

Figure 8.6 shows the difference between the different types of computing mechanisms. These include multidisciplinary, interdisciplinary, and transdisciplinary computing. The metaprogram is used for transdisciplinary computing. The diagram clearly shows the place where metaprogram is executed. Figure 8.6a shows a discipline that consists of several data, operations, and control components connected together in a workflow to give the result. Figure 8.6b shows multidisciplinary models that involved two or more disciplines but not intercommunication between the disciplines. Similarly, Figure 8.6c shows an interdisciplinary model where intercommunication between the disciplines is involved. Finally, Figure 8.6d shows the transdisciplinary models where multiple disciplines with interdisciplinary communication are present, and here the components from outside the discipline can also contribute to the results. These transdisciplinary models use the metaprogram.

The whole concept of SORCER is based on remote processing. Here, the corresponding computational components are autonomically provisioned by using metainstructions that are produced by service metaprocessor. This is done instead of sending the executable files directly through the Internet. Metaprogram and metainstruction are service command and service instruction, respectively.

Here, a metacompute OS is used, which manages the whole dynamic federation of service providers and resources. With its own control strategy, it allows all the service providers to collaborate among themselves, which is done according to the metaprogram definition given. The metaprogram, in terms of metainstruction, can be submitted to metacompute OS.

There are several layers in SORCER, as shown in Figure 8.7, and among them SORCER OS is the most important. In the figure, the layers are grouped according to their functionality. The command virtual platform and the object virtual platform collectively form the platform cloud.

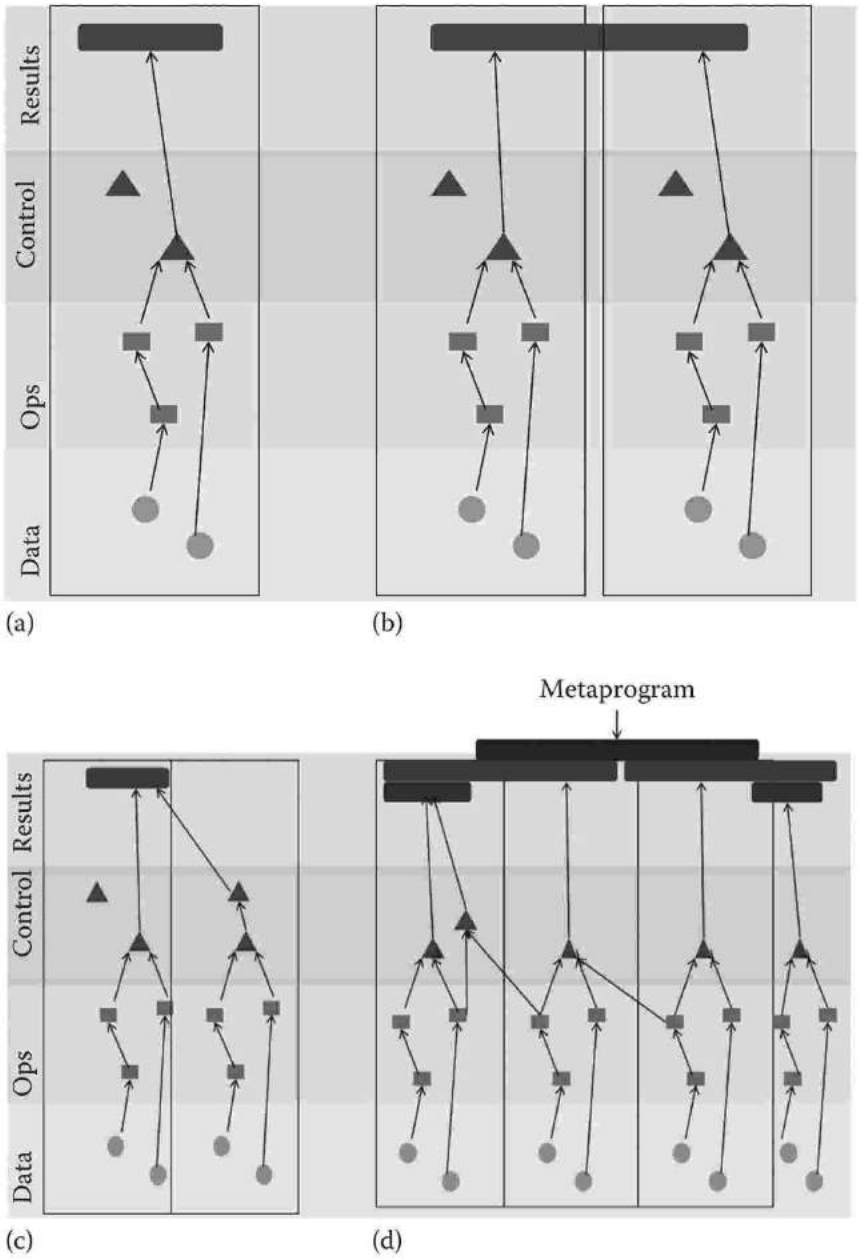

FIGURE 8.6
Types of operations: (a) discipline, (b) multidisciplinary, (c) interdisciplinary, and (d) transdisciplinary. (Adapted from Sobolewski, M., Object-oriented service clouds for transdisciplinary computing, in *Cloud Computing and Services Science*, Springer, New York, 2012, pp. 3–31.)

The service nodes and the domain specific (DS) service providers collectively do the service cloud operation. The aforementioned operations with command native platforms and evaluator filters are controlled by metaprocessor. The whole thing is managed by service operating system and above which service programs or service requestors request the services.

SORCER is focused on metamogramming. Metamogramming is a process in which metamodeling or programming environment is used for creation of symbolic process expression. Thus, SORCER introduces a

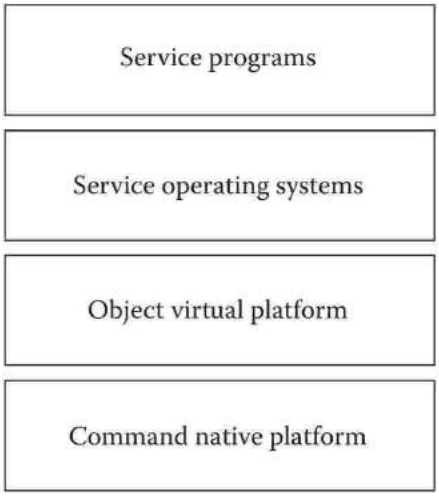

FIGURE 8.7
SORCER architecture. (Adapted from Sobolewski, M., Object-oriented service clouds for transdisciplinary computing, in *Cloud Computing and Services Science*, Springer, New York, 2012, pp. 3–31.)

metacompute OS that consists of all the required system services, which includes object metaprogramming, federated file system, and autonomic resource management. These properties are to support service object–oriented programming. It basically provides solution for multiple transdisciplinary applications that require complex transdisciplinary solutions. SORCER is deployed over FIPER.

FIPER is federated intelligent project environment, and its main aim is to provide federation of distributed service objects that provide engineering data, applications, and tools on a network. In short, SORCER metamodeling architecture is based on domain/management/carrier, or the DMC triplet [8].

Key features

- Object-oriented nature
- Service-oriented architecture
- Transdisciplinary

8.2.7 Programming Models in Aneka

Aneka is a PaaS developed by Manjrasoft, Inc. There are three programming models defined and used by Aneka, which are discussed in the following.

8.2.7.1 Task Execution Model

In this model, the application is divided into several tasks. Each task is executed independently. As defined by the developers of Aneka, tasks are work

units that are executed by the scheduler in any order. As it involves a large number of tasks, it is very much suitable for distributed applications, specifically applications where several independent results are involved, and these independent results are then combined to give a final result. For these kinds of problems, the independent results can be considered as output from the tasks, and later these results can be combined by the user. It also supports dynamic creation of tasks [9].

Key features

- Suitable for distributed application
- Independent tasks

8.2.7.2 Thread Execution Model

In thread model, the applications are executed using processes. Each process consists of one or more threads. As defined by the developers of Aneka, threads are a sequence of instructions that can be executed in parallel with other instructions. Thus, in thread execution model, the execution is taken care by the system [10].

Key features

- Supports heavy parallelization.
- Multiple thread programming is available.

8.2.7.3 Map Reduce Model

This model is similar to the actual MapReduce model described in the first section. The Map Reduce model is implemented in the Aneka platform.

8.3 New Programming Models Proposed for Cloud

This section discusses about the programming models that are newly designed for cloud. These programming models are designed from scratch. As these models are specifically designed for cloud, the important properties of cloud, such as scalability, distributed nature, and multitenancy, form the base for designing algorithm. Unlike the previous method, there is no necessity to go according to any language specification. Mainly all the approaches concentrate on the distributed nature of algorithms. The concept of a new programming model came because of certain disadvantages that were found

in existing programming models. The main disadvantage is that you cannot eliminate certain basic properties of existing approaches. Only new parameters can be added or slight modification can be done so as to achieve the required goal. Whereas when a new programming model is designed, the designer has full liberty to include anything without any restriction. The following section also includes a few APIs and toolkits that are specifically designed for cloud.

8.3.1 Orleans

Orleans is a framework developed by Microsoft for creating cloud applications [11]. It is an attempt by Microsoft to give a software framework that will facilitate both client and server sides of cloud application development. This uses a concurrent programming paradigm.

As this application is designed for cloud, there are several characteristics that are considered. The important features of Orleans include

- Concurrency
- Scalability
- Reduced errors
- Security

The basic idea around which this whole framework is developed is the use of grains. These grains are logical computational units that are considered to be the building blocks of Orleans. The grains are independent and isolated, and all the processes are executed by using grains. A grain can have several activations. These activations are instantiations of grains in the physical server. Activations are helpful as several processes can concurrently be executed by using several activations. Though the model supports concurrency by default, a grain does not execute processes concurrently. A grain can serve only one request at a time. The next request can be served only once a request is completed.

There can be zero, one, or several activations. Thus, to support parallel execution, several activations of a single grain can be created and several requests can be served in parallel. Several advantages of this feature, as quoted by the developers, include increasing throughput and reducing latency. This feature has a great impact on Orleans. They communicate with each other by passing messages. Each grain has a state. The state determines the current position of a grain. When several activations are created for a single grain, there is a possibility that the state of the grain might change. This might lead to inconsistencies among all the activations. This issue is also addressed by Orleans.

Scalability is another issue that needs to be considered when it comes to the development of cloud application. Orleans on its part considers scalability as an important issue and tries to provide a solution. The concept of grains and activation itself considers scalability, and in addition to that, Orleans addresses the issue by the use of sharded database. Sharded database is horizontally partitioned into several components with each component (shard) consisting of several rows. This sharded database architecture is used in the back end to support the grains. Thus, grains can be fully independent and can have an independent computation. This is called grain domain. The property of distributed and independent computation is very effective since all the applications depend on database. If there is a single database, then it becomes a bottleneck. Thus, everything would depend on the single database, and if that database fails, the whole system collapses. But when the system is distributed in nature, it is not the case.

The failure of a single application results in a failure of a part of a database that would not affect other chunks of databases and thereby would not affect other applications. Thus, in a way, it becomes fault tolerant and is extensively usable and scalable as application isolation and independence is guaranteed. There is no restriction in creation of number of grains and number of activations. Thus, as the number of request increases, so are the number of processes, then we can simultaneously increase the grains and activations to serve those requests. These processes will not affect each other as they are completely isolated. And so, the concept of grains and activation itself is the solution for scalability. Figure 8.8 depicts the layer at which the Orleans works. The Orleans is implemented as a library in C# over .Net framework [11].

Apart from these issues, the model introduces transactions, and according to the developers there are several reasons for using transactions. The first reason is that the activations can change the state of the grains; thus, all the changes made by the processes need to be maintained. To maintain consistency, there is a necessity to have a separate mechanism for maintaining it as several activations can change the data simultaneously. The other reason is replication of data. Replication of data, which is inevitable in the case of cloud technology, has to be done carefully. This is because several applications are dependent on data, and replication of data can

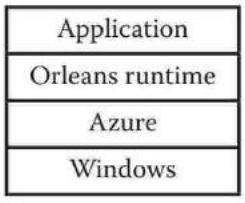

FIGURE 8.8
Orleans system. (Adapted from ORLEANS: A framework for cloud computing, available: http://research.microsoft.com/en-us/projects/orleans/.)

result in inconsistency. Thus, transactions are introduced. Apart from scalability, Orleans introduces restricted concurrency [11].

Key features

- Fault tolerant
- Sharded database
- Efficient transaction handling
- Restricted concurrency
- Security

8.3.2 BOOM and Bloom

Berkeley orders of magnitude (BOOM) is a project undertaken by UC Berkeley. It aims at creation of a programming framework designed for cloud. Bloom is a programming language created for the project BOOM. Bloom programming language is extensively distributed in nature. BOOM is highly distributed and unstructured in nature. As the name suggests, this was designed so that people can design large systems as cloud. It tries to follow a nontraditional approach to an extent possible. The approach developed is disorderly and data centric. BOOM aims at less code and more scalability. It uses a fully distributed approach. BOOM uses BOOM FS, which is a Hadoop distributed file system (HDFS). According to the analytics, the BOOM FS is 20% better than HDFS. The design goals of Bloom are as follows [13]:

- Familiar syntax
- Integrated with imperative languages
- Modularity, encapsulation, and composition

BLOOM uses the CALM principle and is built by using dedalus. The CALM principle is used for building automatic program analysis and visualization tools for reasoning about coordination and consistency, whereas dedalus is a temporal logic language. Dedalus is a time-oriented distributed programming language. It focuses on data-oriented nature based on time rather than space. Data related to space is not extensively used. It is not necessary for the programmer to understand the behavior of the interpreter or compiler since Dedalus is a pure temporal language [14].

Key features

- Modularity, encapsulation, and composition
- Extensively distributable
- Unstructured

8.3.3 GridBatch

GridBatch is a system that allows programmers to easily convert a high-level design into the actual parallel implementation in a cloud computing–like infrastructure. This was developed by Liu and Orban [15]. It does not parallelize automatically, and a programmer has to do this after understanding the application and should be able to decide upon its parallelizability factors. GridBatch provides several operators facilitating the programmers to do the tasks for parallelization. Usually, to implement an application, it should be broken down into 296 pieces of smaller tasks and the partition should be such that it has a maximum performance. To facilitate these things, GridBatch provides a set of libraries. These libraries have everything in it. The programmer needs to know how to use the library, and by using the library and operators effectively, a programmer can achieve the highest efficiency. GridBatch is specifically designed for cloud applications, but because of its simple parallel programming nature, it can be effectively used for grid computing and cluster computing as well. It supports the aforementioned environments better than its other counterparts, and it gives higher programmer productivity. As far as GridBatch is concerned, an application should have a large amount of data parallelism to have large performance gains. This GridBatch framework is suitable for analytical applications that involve a large amount of statistical data [15].

Key features

- Simple
- Supports efficient parallelization
- Better performance gains

8.3.4 Simple API for Grid Applications

Simple API for grid applications (SAGA) is a high-level API that provides a simple, standard, and uniform interface for the most commonly required distributed functionality. It was developed by Goodale et al. [16] and is mainly used for distributed applications. It also provides separate toolkits to work with distributed applications and provides assistance for implementing commonly used programs. The SAGA API is written in C++ with C and Java language support. A library called engine provides support for runtime environment decision making by using a relevant adapter, and this is the main and most important library [17]. SAGA uses task model and asynchronous notification mechanism for defining a concurrent programming model.

Here, the concurrent units are allowed to share object states, which are one of the necessary properties. This property creates an additional overhead of concurrency control. Thus, enforced concurrency mechanism needs to be adopted. But, enforced concurrency mechanism can be a problem in

some cases as sometimes it might not suit a certain problem domain and might result in unnecessary increase in complexity and overhead. Thus, concurrency control mechanisms are not enforced. The user has to take care of concurrency. SAGA works directly over the grid toolkits like global and condor that are used for creating grids. Unlike open grid service architecture (OSGA), SAGA is not a middleware; instead, SAGA can work over OSGA. The distributed applications are deployed over SAGA.

The applications are works over distributed application pattern or user modes. Though SAGA works directly over toolkits, it is not a middleware. It is an API as the name suggests. Figure 8.9 depicts the architecture of SAGA with all the layers involved in an application development. Here, the SAGA API works over several grid components such as condor and globus. Several types of distributed application managing modes are deployed over SAGA. These involve process like MapReduce and process for managing hierarchical jobs. Over this, the actual distributed application works. An application can work directly either over SAGA or over distributed application pattern layer. Usually, the intermediate layer between SAGA and application facilitates an efficient management of resources.

Further, SAGA can be elaborated based on the two types of packages available. The first one is the packages that are related to look and feel. These look and feel API packages are important because it is responsible for providing the *look* and *feel* throughout. These include task model, security, monitoring, error handling, and attributes. Another package is called SAGA API packages. These are the basic packages that are required for SAGA.

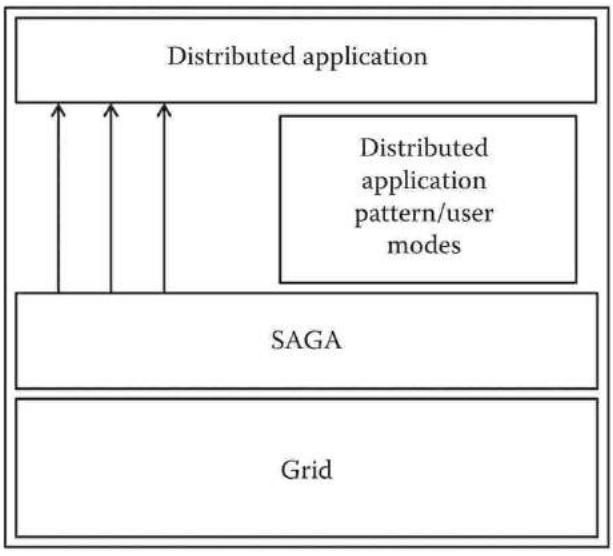

FIGURE 8.9
SAGA architecture. (Adapted from SAGA tutorial, available: http://www.cybertools.loni.org/rmodules.php.)

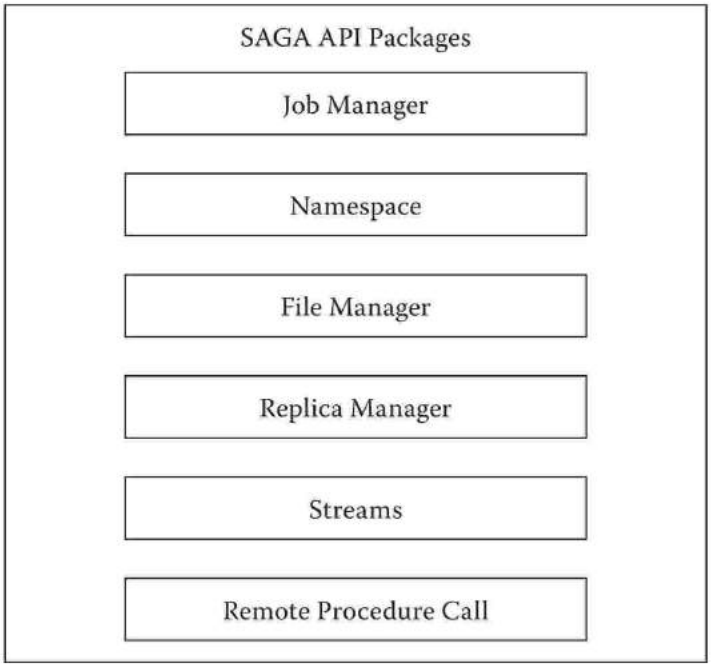

FIGURE 8.10
SAGA API packages.

These include job management, namespace, file management, replica management streams, and remote procedure call. Figure 8.10 depicts these packages.

1. *Job management*: As SAGA involves a large number of jobs to be submitted to the grid, job management function is used. This is used for submission, control, and monitoring the jobs. The function of the job manager is to submit the job to the grid resource, describe the procedure for controlling these jobs, and retrieve information about running and completed jobs. The submission of jobs is done for two modes: batch and interactive.
2. *Namespace management*: Namespace management is responsible for managing namespaces, and it works collectively with file management package.
3. *File management*: The files that are part of this SAGA are managed by the application. These files are accessed by the SAGA application without knowing the location of the file. This, along with namespace, is responsible for several operations on the file like read, seek, and write.
4. *Replica management*: Replica management describes interactions with replica systems [17]. A replica is a file registered on a logical file. This is used for creation and maintenance of logical file replicas and to search a logical file on metadata [17].

5. *Streams management*: The main function of a stream manager is to apply the simplest possible authenticated socket connection with hooks to support application-level authorization.
6. *Remote procedure call*: This manages all the remote procedure calls and provides methods for communication.

Key features

- Simple API
- C and Java language support

8.4 Summary

The users using cloud are increasing rapidly; thus, the demand for cloud is very high. The way an application is developed in the usual environment and the way it is developed in cloud are different. This chapter discusses about the programming models that are proposed for cloud computing. As cloud computing involves development of several applications, there needs to be a specific programming model that will facilitate the development of cloud application. As cloud applications are different, the existing programming model cannot be used as such. This chapter discusses both the types of programming models for cloud: the existing programming models that can be extended to cloud and the new programming models proposed. In the first category, functional programming, parallel programming, and object-oriented programming are extended to cloud. Existing programming languages that suit cloud are also discussed. Similarly, as far as new models are concerned, the distributed nature of cloud is mainly taken into account. Several new programming models proposed are discussed as well as a few frameworks specifically designed for cloud programming.

Review Points

- *Sharded database*: Sharded database is horizontally partitioned into several components with each component (shard) consisting of several rows (see Section 8.3.1).
- *Grain*: Logical computational units considered as building blocks of Orleans (see Section 8.3.1).

- *Parasite*: A parasite is a small (lightweight) thread that depends on other threads (main) (see Section 8.2.4).
- *Immutable components*: These are the components whose state cannot be modified after they have been created (see Section 8.2.3).
- *Idempotency*: It is the ability of a function to restart without the use of any external entity or recovery mechanism, even after the system has encountered a hardware failure (see Section 8.2.3).
- *Content dissemination network*: Content dissemination network is responsible for managing communication between different components of CGL-MapReduce (see Section 8.2.2).
- *Metamogramming*: A process in which metamodeling or programming environment is used for creation of symbolic process expression (see Section 8.2.6).
- *Metaprogram*: It is a program that manipulates other programs remotely as its data (see Section 8.2.6).
- *GridBatch*: It is a system that allows programmers to easily convert a high-level design into the actual parallel implementation in a cloud computing–like infrastructure (see Section 8.3.3).
- *Condor and globus*: Grid toolkits used for creating grids (see Section 8.3.4).
- *Fault tolerant*: It refers to the ability of the system to withstand error (see Section 8.2.5).

Review Questions

1. What is the main difference(s) between Cloud Haskell and Erlang?
2. What are the pros and cons of having a new programming model?
3. Which programming language is highly fault tolerant and for which type of application was it originally designed?
4. Point out the difference(s) between MapReduce and CGL-MapReduce.
5. In Orleans, failure of one portion of the application would not affect the whole application. Justify.
6. Does Cloud Haskell have its own fault-tolerance mechanism? If no, then on whose fault mechanism it is based?
7. Which programming model (language) allows runtime code replacement?
8. What are the main reasons for using transactions in Orleans?

9. What are the components of SAGA?
10. Does GridBatch parallelize the code automatically? If yes, then how? Else, who is responsible for parallelization of the code?

References

1. Dean, J. and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *ACM Communications*, 51: 107–113, January 2008.
2. Jin, C. and R. Buyya. Mapreduce programming model for. Net-based cloud computing. *Euro-Par 2009 Parallel Processing*, Delft, The Netherlands. Springer, Berlin, Germany, 2009, pp. 417–428.
3. Ekanayake, J., Pallickara, S., and G. Fox. Mapreduce for data intensive scientific analyses. *IEEE Fourth International Conference on eScience*, Indianapolis, IN. IEEE, Washington, DC, 2008.
4. Epstein, J., A. P. Black, and S. Peyton-Jones. Towards Haskell in the cloud. *ACM SIGPLAN Notices* 46(12): 118–129, 2011.
5. Sivaramakrishnan, K. C., L. Ziarek, and S. Jagannathan. CML: Migrating MultiMLton to the cloud.
6. A cloud at the lowest level. Available [Online]: http://cloudi.org/. Accessed November 10, 2013.
7. Bringing Erlang's fault-tolerance to polyglot development. Available [Online]: http://www.toptal.com/erlang/a-cloud-at-the-lowest-level-built-in-erlang. Accessed December 10, 2013.
8. Sobolewski, M. Object-oriented service clouds for transdisciplinary computing. In *Cloud Computing and Services Science*. Springer, New York, 2012, pp. 3–31.
9. Developing thread model applications—ManjraSoft. Available [Online]: www.manjrasoft.com/download/ThreadModel.pdf. Accessed November 5, 2013.
10. Developing task model applications—ManjraSoft. Available [Online]: www.manjrasoft.com/download/TaskModel.pdf. Accessed November 6, 2013.
11. Bykov, S. et al. Orleans: A framework for cloud computing. Technical Report MSR-TR-2010-159, Microsoft Research, 2010.
12. ORLEANS: A framework for cloud computing. Available [Online]: http://research.microsoft.com/en-us/projects/orleans/. Accessed November 10, 2013.
13. Neil, C. Cloud programming: From doom and gloom to BOOM and Bloom. UC Berkeley, Berkeley, CA.
14. BOOM: Berkeley Orders of Magnitude. Available [Online]: http://boom.cs.berkeley.edu/. Accessed November 3, 2013.
15. Liu, H. and D. Orban. Gridbatch: Cloud computing for large-scale data-intensive batch applications. *8th IEEE International Symposium on Cluster Computing and the Grid, 2008 (CCGRID'08)*. IEEE, 2008.
16. Tom, G. et al. A simple API for grid application (SAGA). Available [Online]: www.ogf.org/documents/GFD.90.pdf. Accessed November 20, 2013.

17. Miceli, C. et al. Programming abstractions for data intensive computing on clouds and grids. *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE Computer Society, 2009.
18. SAGA Tutorial. Available [Online]: http://www.cybertools.loni.org/rmodules.php. Accessed December 5, 2013.

Further Reading

Developing MapReduce.NET applications—ManjraSoft. Available [Online]: www.manjrasoft.com/download/2.0/MapReduceModel.pdf.

Erlang programming language. Available [Online]: http://www.erlang.org/.