CHAPTER

Cloud Application Development 11

In the previous chapters our discussion was focused on research issues in cloud computing. Now we examine computer clouds from the perspective of an application developer. This chapter presents a few recipes that are useful in assembling a cloud computing environment on a local system and in using basic cloud functions.

It is fair to assume that the population of application developers and cloud users is and will continue to be very diverse. Some cloud users have developed and run parallel applications on clusters or other types of systems for many years and expect an easy transition to the cloud. Others are less experienced but willing to learn and expect a smooth learning curve. Many view cloud computing as an opportunity to develop new businesses with minimum investment in computing equipment and human resources.

The questions we address here are: How easy is it to use the cloud? How knowledgeable should an application developer be about networking and security? How easy is it to port an existing application to the cloud? How easy is it to develop a new cloud application?

The answers to these questions are different for the three cloud delivery models, *SaaS, PaaS*, and *IaaS*; the level of difficulty increases as we move toward the base of the cloud service pyramid, as shown in Figure 11.1. Recall that *SaaS* applications are designed for end users and are accessed over the Web; in this case, users must be familiar with the API of a particular application. *PaaS* provides a set of tools and services designed to facilitate application coding and deploying; *IaaS* provides the hardware and the software for servers, storage, and networks, including operating systems and storage management software. The *IaaS* model poses the most challenges; thus, we restrict our discussion to the *IaaS* cloud computing model and concentrate on the most popular services offered at this time, the *Amazon Web Services* (AWS).

Though the *AWS* are well documented, the environment they provide for cloud computing requires some effort to benefit from the full spectrum of services offered. In this section we report on lessons learned from the experience of a group of students with a strong background in programming, networking, and operating systems; each of them was asked to develop a cloud application for a problem of interest in his or her own research area. Here we first discuss several issues related to cloud security, a major stumbling block for many cloud users; then we present a few recipes for the development of cloud applications, and finally we analyze several cloud applications developed by individuals in this group over a period of less than three months.

In the second part of this chapter we discuss several applications. Cognitive radio networks (CRNs) and a cloud-based simulation of a distributed trust management system are investigated in Section 11.10. In Section 11.11 we discuss a cloud service for CRN trust management using a history-based algorithm, and we analyze adaptive audio streaming from a cloud in Section 11.12. A cloud-based optimal FPGA

Cloud Computing. http://dx.doi.org/10.1016/B978-0-12-404627-6.00011-7

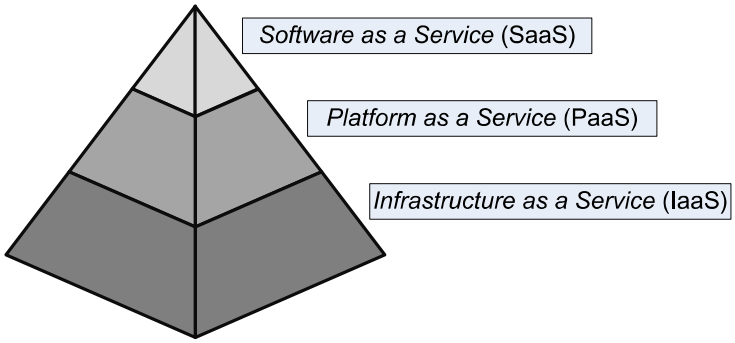

FIGURE 11.1

A pyramid model of cloud computing paradigms. The infrastructure provides the basic resources, the platform adds an environment to facilitate the use of these resources, while software allows direct access to services.

(Field-Programmable Gate Arrays) synthesis with multiple instances running different design options is presented in Section 11.13.

11.1 *Amazon Web Services: EC2* instances

Figure 11.2 displays the *Amazon Management Console* (AMC) window listing the Amazon Web Services offered at the time of this writing. The services are grouped into several categories: computing and networking, storage and content delivery, deployment and management, databases, and application services.

In spite of the wealth of information available from the providers of cloud services, the learning curve of an application developer is still relatively steep. The examples discussed in this chapter are designed to help overcome some of the hurdles faced when someone first attempts to use the *AWS*. Due to space limitations we have chosen to cover only a few of the very large number of combinations of services, operating systems, and programming environments supported by *AWS*.

In Section 3.1 we mentioned that new services are continually added to *AWS*; the look and feel of the Web pages changes over time. The screen shots we've selected reflect the state of the system at the time of the writing of this book, the second half of 2012.

To access *AWS* one must first create an account at `http://aws.amazon.com/`. Once the account is created, the AMC allows the user to select one of the services, e.g., *EC2*, and then start an instance.

Recall that an *AWS EC2 instance* is a virtual server started in a region and the availability zone is selected by the user. Instances are grouped into a few classes, and each class has available to it a specific amount of resources, such as: CPU cycles, main memory, secondary storage, and communication and I/O bandwidth. Several operating systems are supported by *AWS*, including *Amazon Linux, Red Hat Enterprise Linux, 6.3, SUSE Linux Enterprise Server 11, Ubuntu Server 12.04.1*, and several versions of *Microsoft Windows* (see Figure 11.3).

The next step is to create an (AMI)[1] on one of the platforms supported by *AWS* and start an instance using the *RunInstance* API. If the application needs more than 20 instances, a special form must be

[1] An AMI is a unit of deployment. It is an environment that includes all information necessary to set up and boot an instance.

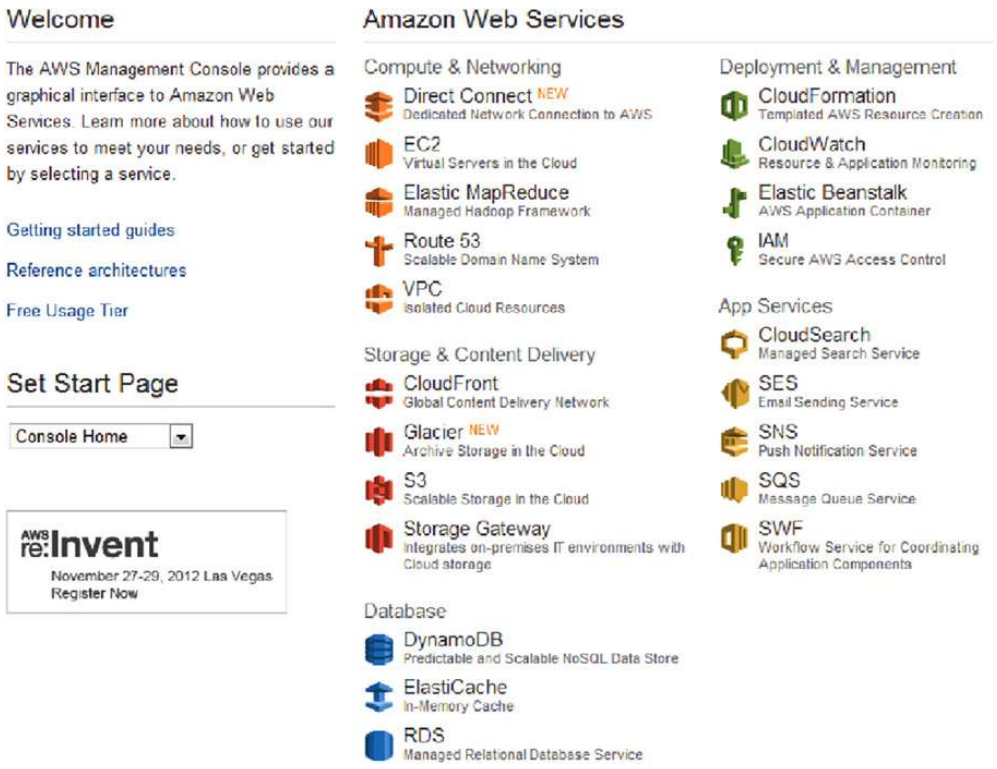

FIGURE 11.2

Amazon Web Services accessible from the *Amazon Management Console*.

filled out. The local instance store persists only for the duration of an instance; the data will persist if an instance is started using the Amazon Elastic Block Storage (EBS) and then the instance can be restarted at a later time.

Once an instance is created, the user can perform several actions – for example, connect to the instance, launch more instances identical to the current one, or create an EBS AMI. The user can also terminate, reboot, or stop the instance (see Figure 11.4). The *Network & Security* panel allows the creation of *Security Groups, Elastic IP addresses, Placement Groups, Load Balancers*, and *Key Pairs* (see the discussion in Section 11.3), whereas the EBS panel allows the specification of volumes and the creation of snapshots.

11.2 Connecting clients to cloud instances through firewalls

A firewall is a software system based on a set of rules for filtering network traffic. Its function is to protect a computer in a local area network from unauthorized access. The first generation of firewalls, deployed in the late 1980s, carried out *packet filtering*; they discarded individual packets that did not match a set of acceptance rules. Such firewalls operated below the transport layer and discarded packets based on the information in the headers of physical, data link, and transport layer protocols.

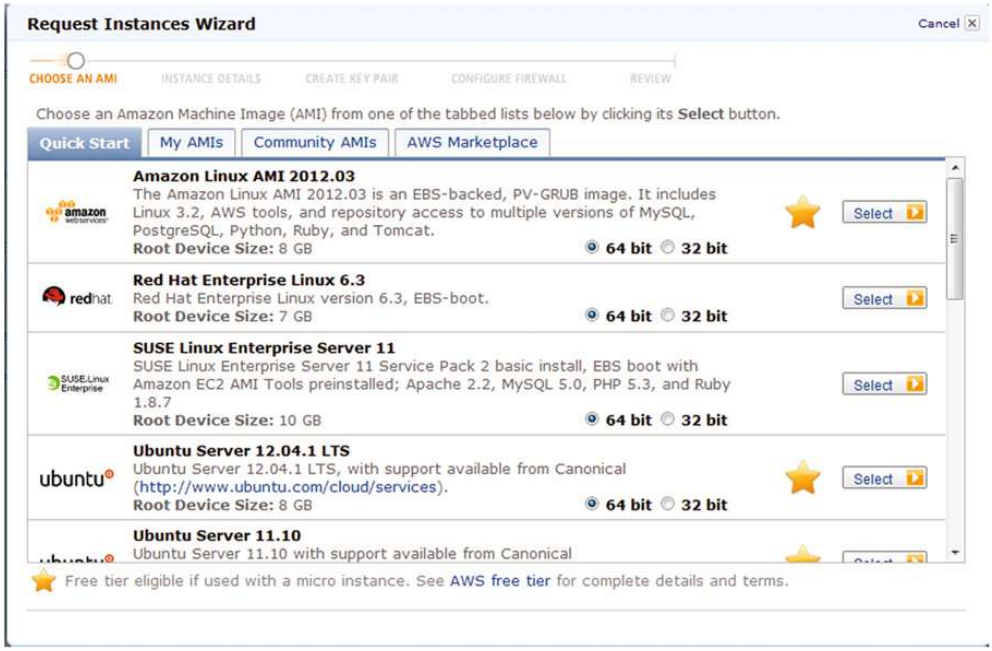

FIGURE 11.3

The Instance menu allows the user to select from existing AMIs.

The second generation of firewalls operate at the transport layer and maintain the state of all connections passing through them. Unfortunately, this traffic-filtering solution opened the possibility of *denial-of-service (DoS) attacks*. A DoS attack targets a widely used network service and forces the operating system of the host to fill the connection tables with illegitimate entries. DoS attacks prevent legitimate access to the service.

The third generation of firewalls "understand" widely used application layer protocols such as *FTP, HTTP, TELNET, SSH,* and *DNS*. These firewalls examine the header of application layer protocols and support *intrusion detection systems* (IDSs).

Firewalls screen incoming traffic and sometimes filter outgoing traffic as well. A first filter encountered by the incoming traffic in a typical network is a firewall provided by the operating system of the router; the second filter is a firewall provided by the operating system running on the local computer (see Figure 11.5).

Typically, the local area network (LAN) of an organization is connected to the Internet via a router. A router firewall often hides the true address of hosts in the local network using the Network Address Translation (NAT) mechanism. The hosts behind a firewall are assigned addresses in a "private address range," and the router uses the NAT tables to filter the incoming traffic and translate external IP addresses to private ones.[2]

[2]The mapping between the (*external address, external port*) pair and the (*internal address, internal port*) tuple carried by the Network Address Translation function of the router firewall is also called a *pinhole*.

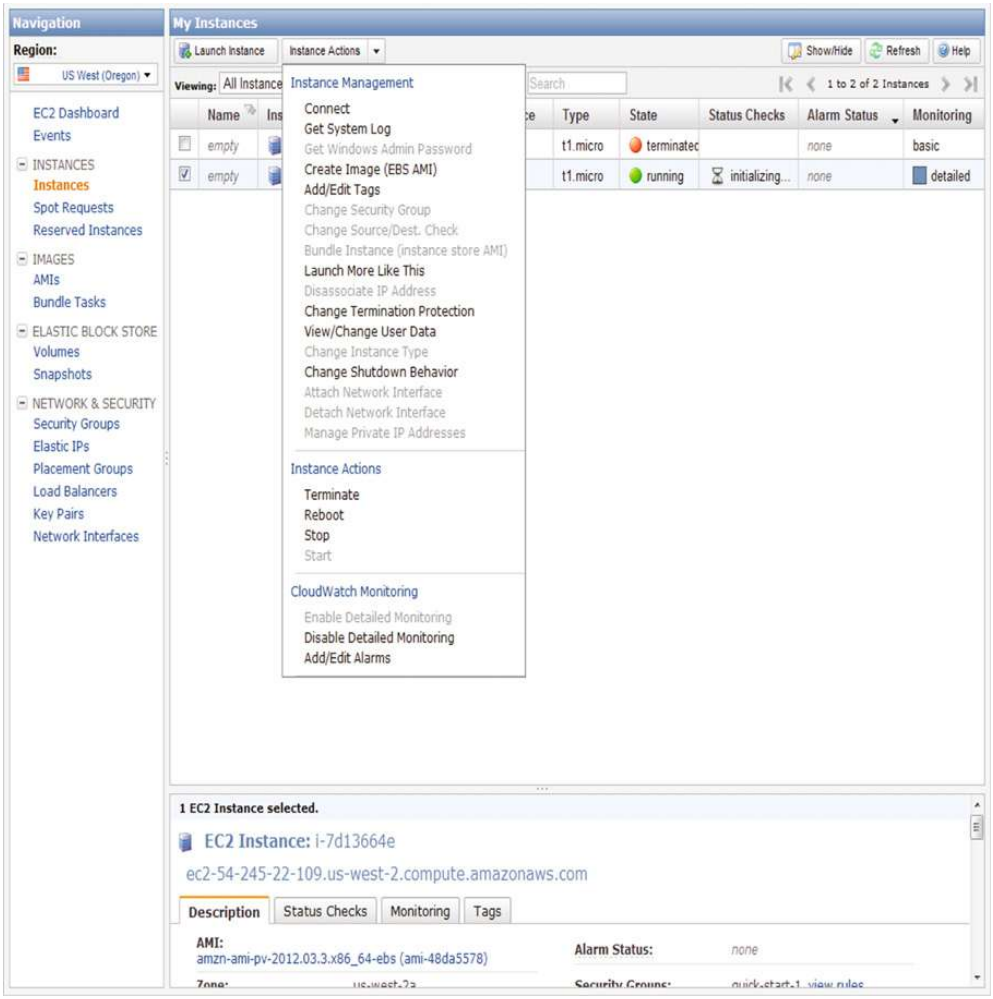

FIGURE 11.4

The *Instance Action* pull-down menu of the *Instances* panel of the *AWS Management Console* allows the user to interact with an instance, e.g., *Connect, Create an EBS AMI Image*, and so on.

If one tests a client-server application with the client and the server in the same LAN, the packets do not cross a router. Once a client from a different LAN attempts to use the service, the packets may be discarded by the router's firewall. The application may no longer work if the router is not properly configured.

Now let's examine the firewall support in several operating systems. Table 11.1 summarizes the options supported by various operating systems running on a host or on a router.

A *rule* specifies a filtering option at (i) the network layer, when filtering is based on the destination/source IP address; (ii) the transport layer, when filtering is based on destination/source port number; or (iii) the MAC layer, when filtering is based on the destination/source MAC address.

In *Linux* or *Unix* systems the firewall can be configured only as a *root* using the *sudo* command. The firewall is controlled by a kernel data structure, the *iptables*. The *iptables* command is used to set up,

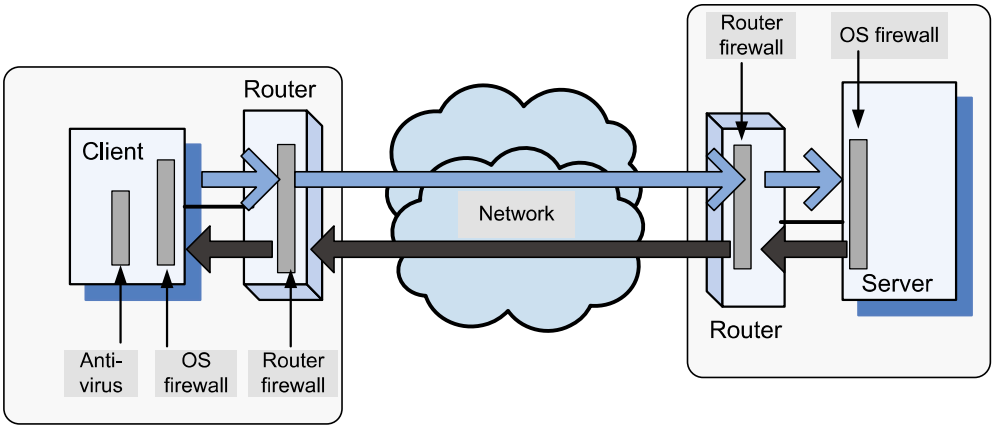

FIGURE 11.5

Firewalls screen incoming and sometimes outgoing traffic. The first obstacle encountered by the inbound or outbound traffic is a router firewall. The next one is the firewall provided by the host operating system. Sometimes the antivirus software provides a third line of defense.

Table 11.1 Firewall rule setting. The first column entries indicate whether a feature is supported by an operating system; the second column, a single rule can be issued to accept/reject a default policy; the third and fourth columns, filtering based on IP destination and source address, respectively; the fifth and sixth columns, filtering based on TCP/UDP destination and source ports, respectively; the seventh and eighth columns, filtering based on ethernet MAC destination and source address, respectively; the ninth and tenth columns, inbound (ingress) and outbound (egress) firewalls, respectively.

Operating System	Def Rule	IP Dest Addr	IP Src Addr	TCP/UDP Dest Port	TCP/UDP Src Port	Ether MAC Dest	Ether MAC Src	In-bound Fwall	Out-bound Fwall
Linux iptables	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
OpenBSD	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Windows 7	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Windows Vista	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes
Windows XP	No	No	Yes	Partial	No	No	No	Yes	No
Cisco Access List	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Juniper Networks	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

maintain, and inspect the tables of the *IPv4* packet filter rules in the *Linux* kernel. Several tables may be defined; each table contains a number of built-in chains and may also contain user-defined chains. A *chain* is a list of rules that can match a set of packets: The *INPUT* rule controls all incoming connections; the *FORWARD* rule controls all packets passing through this host; and the *OUTPUT* rule controls all

outgoing connections from the host. A *rule* specifies what to do with a packet that matches: *Accept*, let the packet pass; *Drop*, discharge the packet; *Queue*, pass the packet to the user space; or *Return*, stop traversing this chain and resume processing at the head of the next chain. For complete information on the *iptables*, see http://linux.die.net/man/8/iptables.

To get the status of the firewall, specify the L (List) action of the *iptables* command:

```
sudo iptables -L
```

As a result of this command the status of the *INPUT, FORWARD,* and *OUTPUT* chains will be displayed.

To change the default behavior for the entire chain, specify the action P (Policy), the chain name, and the target name; e.g., to allow all outgoing traffic to pass unfiltered, use

```
sudo iptables -P OUTPUT ACCEPT s
```

To add a new security rule, specify: the action, A (add), the chain, the transport protocol, TCP or UDP, and the target ports, as in:

```
sudo iptables -A INPUT -p -tcp -dport ssh -j ACCEPT
sudo iptables -A OUTPUT -p -udp -dport 4321 -j ACCEPT
sudo iptables -A FORWARD -p -tcp -dport 80 -j DROP
```

To delete a specific security rule from a chain, set the action D (Delete) and specify the chain name and the rule number for that chain. The top rule in a chain has number 1:

```
sudo iptables -D INPUT 1
sudo iptables -D OUTPUT 1
sudo iptables -D FORWARD 1
```

By default, the *Linux* virtual machines on Amazon's *EC2* accept all incoming connections.

The ability to access that virtual machine will be permanently lost when a user accesses an *EC2* virtual machine using *ssh* and then issues the following command:

```
sudo iptables -P INPUT DROP.
```

The access to the *Windows 7* firewall is provided by a GUI accessed as follows:

```
Control Panel -> System & Security -> Windows Firewall -> Advanced
Settings
```

The default behavior for incoming and/or outgoing connections can be displayed and changed from the window *Windows Firewall with Advanced Security on Local Computer*.

Access to the *Windows XP* firewall is provided by a graphical user interface (GUI) accessed by selecting *Windows Firewall* in the *Control Panel*. If the status is *ON*, incoming traffic is blocked by default and a list of Exceptions (as noted on the *Exceptions* tab) defines the connections allowed. The user can only define exceptions for TCP on a given port, UDP on a given port, and a specific program. *Windows XP* does not provide any control over outgoing connections.

Antivirus software running on a local host may provide an additional line of defense. For example, the *Avast* antivirus software (see www.avast.com) supports several real-time shields. The *Avast*

network shield monitors all incoming traffic; it also blocks access to known malicious Web sites. The *Avast Web shield* scans the HTTP traffic and monitors all Web browsing activities. The antivirus also provides statistics related to its monitoring activities.

11.3 Security rules for application and transport layer protocols in *EC2*

A client must know the IP address of a virtual machine in the cloud to be able to connect to it. The Domain Name Service (DNS) is used to map human-friendly names of computer systems to IP addresses in the Internet or in private networks. DNS is a hierarchical distributed database and plays a role reminiscent of a phone book on the Internet. In late 2010 Amazon announced a DNS service called *Route 53* to route users to *AWS* services and to infrastructure outside of *AWS*. A network of DNS servers is scattered across the globe, which enables customers to gain reliable access to *AWS* and place strict controls over who can manage their DNS system by allowing integration with *AWS* Identity and Access Management (IAM).

For several reasons, including security and the ability of the infrastructure to scale up, the IP addresses of instances visible to the outside world are mapped internally to private IP addresses. A virtual machine running under Amazon's *EC2* has several IP addresses:

1. *EC2 Private IP Address.* The internal address of an instance; it is only used for routing within the *EC2* cloud.
2. *EC2 Public IP Address.* Network traffic originating outside the *AWS* network must use either the public IP address or the elastic IP address of the instance. The public IP address is translated using Network Address Translation (NAT) to the private IP address when an instance is launched and it is valid until the instance is terminated. Traffic to the public address is forwarded to the private IP address of the instance.
3. *EC2 Elastic IP Address.* The IP address allocated to an *AWS* account and used by traffic originated outside *AWS*. NAT is used to map an elastic IP address to the private IP address. Elastic IP addresses allow the cloud user to mask instance or availability zone failures by programmatically remapping public IP addresses to any instance associated with the user's account. This allows fast recovery after a system failure. For example, rather than waiting for a cloud maintenance team to reconfigure or replace the failing host or waiting for DNS to propagate the new public IP to all of the customers of a Web service hosted by *EC2*, the Web service provider can remap the elastic IP address to a replacement instance. Amazon charges a fee for unallocated Elastic IP addresses.

Amazon Web Services use *security groups* to control access to users' virtual machines. A virtual machine instance belongs to one and only one security group, which can only be defined before the instance is launched. Once an instance is running, the security group the instance belongs to cannot be changed. However, more than one instance can belong to a single security group.

Security group rules control inbound traffic to the instance and have no effect on outbound traffic from the instance. The inbound traffic to an instance, either from outside the cloud or from other instances running on the cloud, is blocked unless a rule stating otherwise is added to the security group of the instance. For example, assume a client running on instance A in the security group Σ_A is to connect to a server on instance B listening on TCP port P, where B is in security group Σ_B. A new rule must be

added to security group Σ_B to allow connections to port P; to accept responses from server B, a new rule must be added to security group Σ_A.

The following steps allow the user to add a security rule:

1. Sign in to the *AWS* Management Console at `http://aws.amazon.com` using your email address and password and select *EC2* service.
2. Use the *EC2 Request Instance Wizard* to specify the instance type, whether it should be monitored, and specify a key/value pair for the instance to help organize and search (see Figures 11.6 and 11.7).
3. Provide a name for the key pair. Then on the left-side panel, choose *Security Groups* under *Network & Security*, select the desired security group, and click on the *Inbound* tab to enter the desired rule (see Figure 11.6).

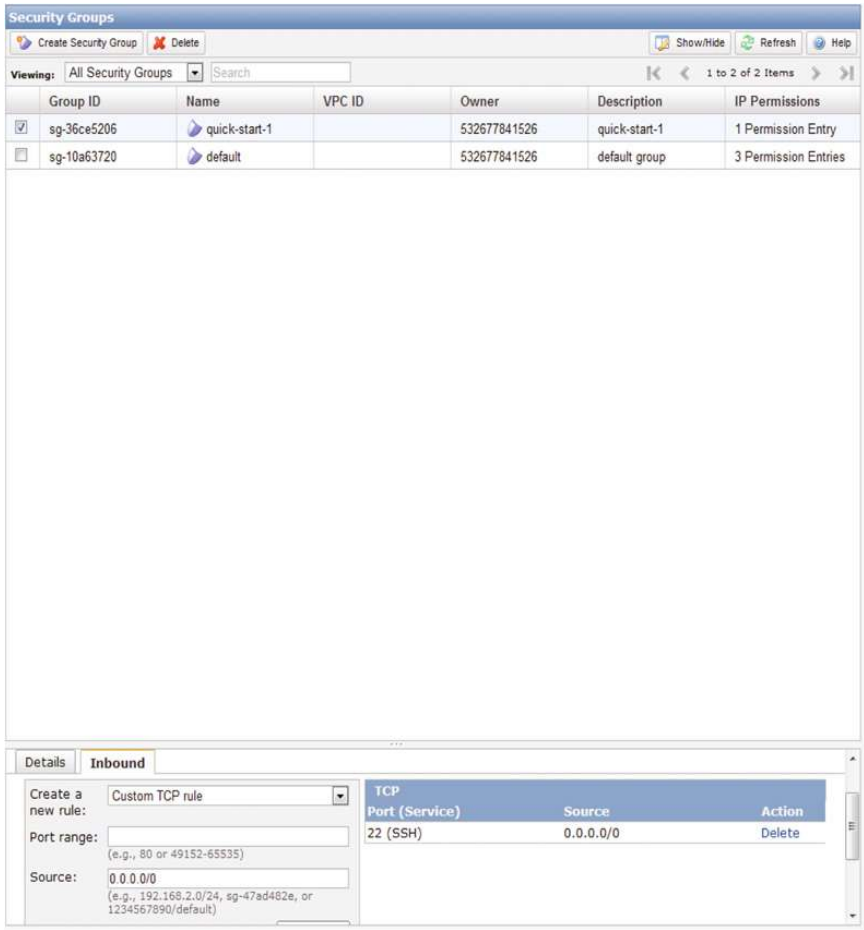

FIGURE 11.6

AWS security. Choose *Security Groups* under *Network & Security*, select the desired security group, and click on the *Inbound* tab to enter the desired rule.

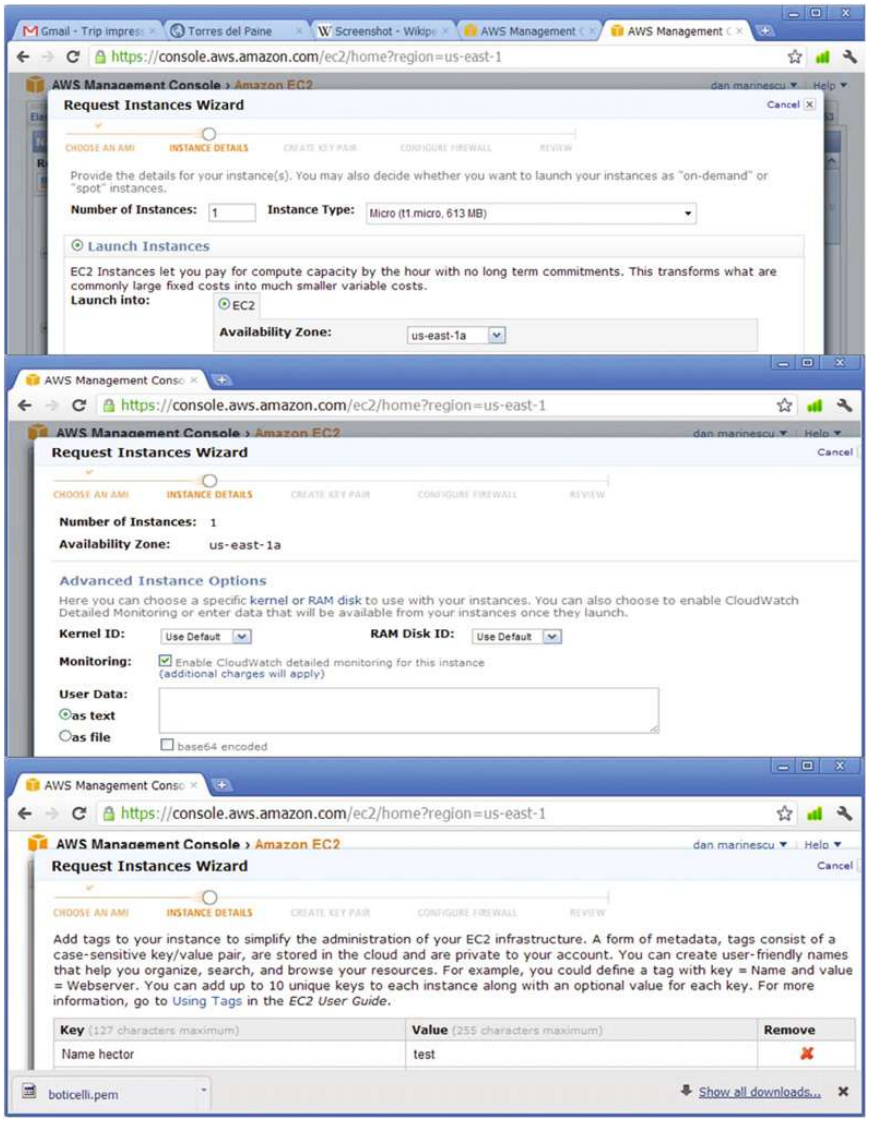

FIGURE 11.7

EC2 Request Instance Wizard is used to (a) specify the number and type of instances and the zone; (b) specify the kernelId and the RAM diskId and enable the *CloudWatch* service to monitor the *EC2* instance; (c) add tags to the instance. A tag is stored in the cloud and consists of a case-sensitive key/value pair private to the account.

To allocate an Elastic IP address, use the *Elastic IPs* tab of the *Network & Security* left-side panel.

On *Linux* or *Unix* systems the port numbers below 1,024 can only be assigned by the *root*. The plain ASCII file called *services* maps friendly textual names for Internet services to their assigned port numbers and protocol types, as in the following example:

```
netstat 15/tcp
ftp 21/udp
ssh 22/tcp
telnet 23/tcp
http 80/tcp
```

11.4 How to launch an *EC2 Linux* instance and connect to it

This section gives a step-by-step process to launch an *EC2 Linux* instance from a *Linux* platform.
A. Launch an instance

1. From the *AWS Management Console*, select *EC2* and, once signed in, go to *Launch Instance Tab*.
2. To determine the processor architecture when you want to match the instance with the hardware, enter the command

   ```
   uname -m
   ```

   and choose an appropriate *Amazon Linux AMI* by pressing *Select*.
3. Choose *Instance Details* to control the number, size, and other settings for instances.
4. To learn how the system works, press *Continue* to select the default settings.
5. Define the instance's security, as discussed in Section 11.3: In the *Create Key Pair* page enter a name for the pair and then press *Create and Download Key Pair*.
6. The key-pair file downloaded in the previous step is a *.pem* file, and it *must* be hidden to prevent unauthorized access. If the file is in the directory *awcdir/dada.pem* enter the commands

   ```
   cd awcdir
   chmod 400 dada.pem
   ```

7. Configure the firewall. Go to the page *Configure firewall*, select the option *Create a New Security Group*, and provide a *Group Name*. Normally we use `ssh` to communicate with the instance; the default port for communication is port 8080, and we can change the port and other rules by creating a new rule.
8. Press *Continue* and examine the review page, which gives a summary of the instance.
9. Press *Launch* and examine the confirmation page, then press *Close* to end the examination of the confirmation page.
10. Press the *Instances* tab on the navigation panel to view the instance.
11. Look for your *Public DNS* name. Because by default some details of the instance are hidden, click on the *Show/Hide* tab on the top of the console and select *Public DNS*.
12. Record the *Public DNS* as *PublicDNSname*; it is needed to connect to the instance from the *Linux* terminal.
13. Use the *ElasticIP* panel to assign an Elastic IP address if a permanent IP address is required.

B. Connect to the instance using *ssh* and the TCP transport protocol.

1. Add a rule to the *iptables* to allow `ssh` traffic using the TCP *protocol*. Without this step, either an *access denied* or *permission denied* error message appears when you're trying to connect to the instance.

```
sudo iptables -A iptables -p -tcp -dport ssh -j ACCEPT
```

2. Enter the *Linux* command:

```
ssh -i abc.pem ec2-user@PublicDNSname
```

   If you get the prompt *You want to continue connecting?* respond *Yes*. A warning that the DNS name was added to the list of known hosts will appear.
3. An icon of the `Amazon Linux AMI` will be displayed.

C. Gain root access to the instance

By default the user does not have *root* access to the instance; thus, the user cannot install any software. Once connected to the *EC2* instance, use the following command to gain *root* privileges:

```
sudo -i
```

Then use `yum` install commands to install software, e.g., *gcc* to compile C programs on the cloud.

D. Run the service ServiceName

If the instance runs under *Linux* or *Unix*, the service is terminated when the *ssh* connection is closed. To avoid the early termination, use the command

```
nohup ServiceName
```

To run the service in the background and redirect *stdout* and *stderr* to files *p.out* and *p.err*, respectively, execute the command

```
nohup ServiceName > p.out 2 > p.err &
```

11.5 How to use *S3* in Java

The Java API for Amazon Web Services is provided by the *AWS* SDK.[3]
Create an S3 client. S3 access is handled by the class *AmazonS3Client* instantiated with the account credentials of the *AWS* user:

```
AmazonS3Client s3 = new AmazonS3Client(
new BasicAWSCredentials("your_access_key", "your_secret_key"));
```

[3] A software development kit (SDK) is a set of software tools for the creation of applications in a specific software environment. Java Development Kit (JDK) is an SDK for Java developers available from Oracle; it includes a set of programming tools such as: *javac*, the Java compiler that converts Java source code into Java bytecode; *java*, the loader for Java applications, which can interpret the class files generated by the Java compiler; *javadoc*, the documentation generator; *jar*, the archiver for class libraries; *jdb*, the debugger; *JConsole*, the monitoring and management console; *jstat*, for JVM statistics monitoring; *jps*, a JVM process status tool; *jinfo*, the utility to get configuration information from a running Java process; *jrunscript*, the command-line script shell for Java; the *appletviewer* tool to debug Java applets without a Web browser; and *idlj*, the IDL-to-Java compiler. The *Java Runtime Environment* is also a component of the JDK, consisting of a Java Virtual Machine (JVM) and libraries.

The access and the secret keys can be found on the user's *AWS* account homepage, as mentioned in Section 11.3.

Buckets. An *S3 bucket* is analogous to a file folder or directory, and it is used to store *S3 objects*. Bucket names must be *globally unique*; hence, it is advisable to check first to see whether the name exists:

```
s3.doesBucketExist("bucket_name");
```

This function returns "true" if the name exists and "false" otherwise. Buckets can be created and deleted either directly from the *AWS* Management Console or programmatically as follows:

```
s3.createBucket("bucket_name");
s3.deleteBucket("bucket_name");
```

S3 objects. An *S3 object* stores the actual data and it is indexed by a key string. A single key points to only one *S3* object in one bucket. Key names do not have to be globally unique, but if an existing key is assigned to a new object, the original object indexed by that key is lost. To upload an object in a bucket, we can use the *AWS Management Console* or, programmatically, a file *local_file_name* can be uploaded from the local machine to the bucket *bucket_name* under the key *key* using

```
File f = new File("local_file_name");
s3.putObject("bucket_name", "key", f);
```

A versioning feature for the objects in *S3* was made available recently; it allows us to preserve, retrieve, and restore every version of an *S3* object. To avoid problems in uploading large files, e.g., dropped connections, use the *.initiateMultipartUpload()* with an API described at the *AmazonS3Client*. To access this object with key *key* from the bucket *bucket_name* use:

```
S3Object myFile = s3.getObject("bucket_name", "key");
```

To `read` this file, you must use the S3Object's *InputStream*:

```
InputStream in = myFile.getObjectContent();
```

The *InputStream* can be accessed using *Scanner, BufferedReader*, or any other supported method. Amazon recommends closing the stream as early as possible, since the content is not buffered and it is streamed directly from the *S3*. An open *InputStream* means an open connection to *S3*. For example, the following code will `read` an entire object and print the contents to the screen:

```
AmazonS3Client s3 = new AmazonS3Client(
    new BasicAWSCredentials("access_key", "secret_key"));
    InputStream input = s3.getObject("bucket_name", "key")
      .getObjectContent();
    Scanner in = new Scanner(input);
    while (in.hasNextLine())
```

```
        {
         System.out.println(in.nextLine());
        }
in.close();
input.close();
```

Batch upload/download. Batch upload requires repeated calls of *s3.putObject()* while iterating over local files.

To view the keys of all objects in a specific bucket, use

```
ObjectListing listing = s3.listObjects("bucket_name");
```

ObjectListing supports several useful methods, including *getObjectSummaries(). S3ObjectSummary* encapsulates most of an *S3* object properties (excluding the actual data), including the key to access the object directly,

```
List<S3ObjectSummary> summaries = listing.getObjectSummaries();
```

For example, the following code will create a list of all keys used in a particular bucket and all of the keys will be available in string form in *List < String >allKeys*:

```
AmazonS3Client s3 = new AmazonS3Client(
   new BasicAWSCredentials("access_key", "secret_key"));
   List<String> allKeys = new ArrayList<String>();
   ObjectListing listing = s3.listObjects("bucket_name");
   for (S3ObjectSummary summary:listing.getObjectSummaries())
    {
     allKeys.add(summary.getKey());
   }
```

Note that if the bucket contains a very large number of objects, then *s3.listObjects()* will return a truncated list. To test if the list is truncated, we could use *listing.isTruncated()*; to get the next batch of objects, use

```
s3.listNextBatchOfObjects(listing)};
```

To account for a large number of objects in the bucket, the previous example becomes

```
AmazonS3Client s3 = new AmazonS3Client(
new BasicAWSCredentials("access_key", "secret_key"));
List<String> allKeys = new ArrayList<String>();
ObjectListing listing = s3.listObjects("bucket_name");
while (true)
  {
   for (S3ObjectSummary summary :
    listing.getObjectSummaries())
```

```
    {
    allKeys.add(summary.getKey());
    }
  if (!listing.isTruncated())
    {
    break;
    }
  listing = s3.listNextBatchOfObjects(listing);
}
```

11.6 How to manage *SQS* services in C#

Recall from Section 3.1 that *SQS* is a system for supporting automated workflows; multiple components can communicate with messages sent and received via *SQS* . An example showing the use of message queues is presented in Section 4.7. Figure 11.8 shows the actions available for a given queue in *SQS*.

The following steps can be used to create a queue, send a message, receive a message, and delete a message, and delete the queue in C#:

1. Authenticate an *SQS* connection:

```
NameValueCollection appConfig =
   ConfigurationManager.AppSettings;
AmazonSQS sqs = AWSClientFactory.CreateAmazonSQSClient
  (appConfig["AWSAccessKey"], appConfig["AWSSecretKey"]);
```

2. Create a queue:

```
CreateQueueRequest sqsRequest = new CreateQueueRequest();
sqsRequest.QueueName = "MyQueue";
CreateQueueResponse createQueueResponse =
   sqs.CreateQueue(sqsRequest);
String myQueueUrl;
```

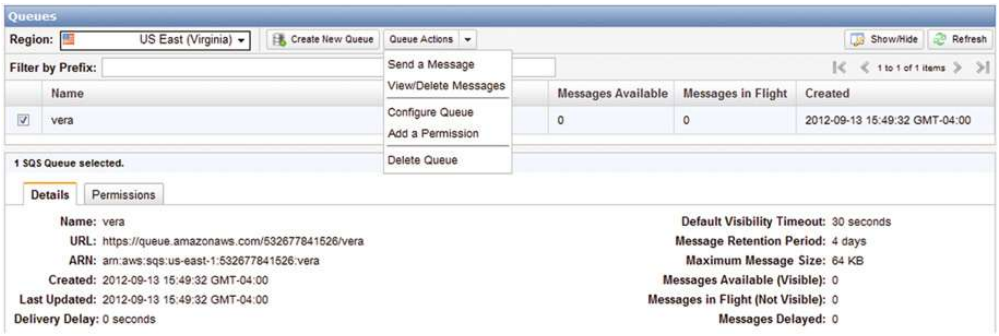

FIGURE 11.8

Queue actions in *SQS*.

```
myQueueUrl = createQueueResponse.CreateQueueResult.QueueUrl;
```

3. Send a message:

```
SendMessageRequest sendMessageRequest =
   new SendMessageRequest();
sendMessageRequest.QueueUrl =
   myQueueUrl; //URL from initial queue
sendMessageRequest.MessageBody = "This is my message text.";
sqs.SendMessage(sendMessageRequest);
```

4. Receive a message:

```
ReceiveMessageRequest receiveMessageRequest =
   new ReceiveMessageRequest();
receiveMessageRequest.QueueUrl = myQueueUrl;
ReceiveMessageResponse receiveMessageResponse =
   sqs.ReceiveMessage(receiveMessageRequest);
```

5. Delete a message:

```
DeleteMessageRequest deleteRequest =
   new DeleteMessageRequest();
deleteRequest.QueueUrl = myQueueUrl;
deleteRequest.ReceiptHandle = messageRecieptHandle;
DeleteMessageResponse DelMsgResponse =
   sqs.DeleteMessage(deleteRequest);
```

6. Delete a queue:

```
DeleteQueueRequest sqsDelRequest = new DeleteQueueRequest();
sqsDelRequest.QueueUrl =
   CreateQueueResponse.CreateQueueResult.QueueUrl;
DeleteQueueResponse delQueueResponse =
sqs.DeleteQueue(sqsDelRequest);
```

11.7 How to install the *Simple Notification Service* on *Ubuntu* 10.04

Ubuntu is an open-source operating system for personal computers based on Debian *Linux* distribution; the desktop version of *Ubuntu*[4] supports the Intel *x86* 32-bit and 64-bit architectures.

The *Simple Notification Service (SNS)* is a Web service for: monitoring applications, workflow systems, time-sensitive information updates, mobile applications, and other event-driven applications that require a simple and efficient mechanism for message delivery. SNS "pushes" messages to clients rather than requiring a user to periodically poll a mailbox or another site for messages.

[4] *Ubuntu* is an African humanist philosophy; "*Ubuntu*" is a word in the Bantu language of South Africa that means "humanity toward others."

SNS is based on the publish/subscribe paradigm; it allows a user to define the topics, the transport protocol used (HTTP/HTTPS, email, SMS, *SQS*), and the endpoint (URL, email address, phone number, *SQS* queue) for notifications to be delivered. It supports the following actions:

- Add/Remove Permission.
- Confirm Subscription.
- Create/Delete Topic.
- Get/Set Topic Attributes.
- List Subscriptions/Topics/Subscriptions by Topic.
- Publish/Subscribe/Unsubscribe.

The document at `http://awsdocs.s3.amazonaws.com/SNS/latest/sns-qrc.pdf` provides detailed information about each one of these actions.

To install the *SNS* client the following steps must be taken:

1. Install Java in the *root* directory and then execute the commands:

```
deb http://archive.canonical.com/lucidpartner
update
install sun-java6-jdk
```

   Then change the default Java settings:

```
update-alternatives -config java
```

2. Download the *SNS* client, unzip the file, and change permissions:

```
wget http://sns-public-resources.s3.amazonaws.com/
    SimpleNotificationServiceCli-2010-03-31.zip
chmod 775 /root/ SimpleNotificationServiceCli-1.0.2.3/bin
```

3. Start the *AWS* Management Console and go to *Security Credentials*. Check the *Access Key ID* and the *Secret Access Key* and create a text file */root/credential.txt* with the following content:

```
AWSAccessKeyId= your_Access_Key_ID
AWSSecretKey= your_Secret_Access_Key
```

4. Edit the *.bashrc* file and add:

```
export AWS_SNS_HOME=~/SimpleNotificationServiceCli-1.0.2.3/
export AWS_CREDENTIAL_FILE=$HOME/credential.txt
export PATH=$AWS_SNS_HOME/bin
export JAVA_HOME=/usr/lib/jvm/java-6-sun/
```

5. Reboot the system.
6. Enter on the command line:

```
sns.cmd
```

   If the installation was successful, the list of *SNS* commands will be displayed.

11.8 How to create an *EC2 Placement Group* and use *MPI*

An *EC2 Placement Group* is a logical grouping of instances that allows the creation of a virtual cluster. When several instances are launched as an *EC2 Placement Group*, the virtual cluster has a high-bandwidth interconnect system suitable for network-bound applications. The cluster computing instances require a hardware virtual machine (HVM) ECB-based machine image, whereas other instances use a paravirtual machine (PVM) image. Such clusters are particularly useful for high-performance computing when most applications are communication intensive.

Once a placement group is created, *MPI* (Message Passing Interface) can be used for communication among the instances in the placement group. *MPI* is a de facto standard for parallel applications using message passing, designed to ensure high performance, scalability, and portability; it is a language-independent "message-passing application programmer interface, together with a protocol and the semantic specifications for how its features must behave in any implementation" [146]. *MPI* supports point-to-point as well as collective communication; it is widely used by parallel programs based on the same program multiple data (SPMD) paradigm.

The following *C* code [146] illustrates the startup of *MPI* communication for a process group *MPI_COM_PROCESS_GROUP* consisting of a number of *nprocesses*; each process is identified by its *rank*. The run-time environment *mpirun* or *mpiexec* spawns multiple copies of the program, with the total number of copies determining the number of process ranks in *MPI_COM_PROCESS_GROUP*.

```
#include <mpi.h>
#include <stdio.h>
#include <string.h>

#define TAG 0
#define BUFSIZE 128

int main(int argc, char *argv[])
{
 char idstr[32];
 char buff[BUFSIZE];
 int nprocesses;
 int my_processId;
 int i;
 MPI_Status stat;

 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COM_PROCESS_GROUP,&nprocesses);
 MPI_Comm_rank(MPI_COM_PROCESS_GROUP,&my_processId);
```

MPI_SEND and *MPI_RECEIVE* are blocking send and blocking receive, respectively; their syntax is:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag,MPI_Comm comm)
```

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
          int source, int tag, MPI_Comm comm, MPI_Status *status)
```

with

`buf`	— initial address of send buffer (choice).
`count`	— number of elements in send buffer (nonnegative integer).
`datatype`	— data type of each send buffer element (handle).
`dest`	— rank of destination (integer).
`tag`	— message tag (integer).
`comm`	— communicator (handle).

Once started, every process other than the coordinator, the process with $rank = 0$, sends a message to the entire group and then receives a message from all the other members of the process group.

```
if(my_processId == 0)
{
 printf("%d: We have %d processes\n", my_processId, nprocesses);
 for(i=1;i<nprocesses;i++)
 {
  sprintf(buff, "Hello %d! ", i);
  MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_PROCESS_
  GROUP);
 }
 for(i=1;i<nprocesses;i++)
 {
  MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_PROCESS_
  GROUP,&stat);
  printf("%d: %s\n", my_processId, buff);
 }
}
else
{
 /* receive from rank 0: */
 MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_PROCESS_
 GROUP, &stat);
 sprintf(idstr, "Processor %d ", my_processId);
 strncat(buff, idstr, BUFSIZE-1);
 strncat(buff, "reporting for duty\n' BUFSIZE-1);
 /* send to rank 0: */
 MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COM_PROCESS
 _GROUP);
}
MPI_Finalize();
return 0;
}
```

An example of cloud computing using the *MPI* is described in [119]. An example of *MPI* use on *EC2* is located at http://rc.fas.harvard.edu/faq/amazonec2.

11.9 How to install *Hadoop* on *Eclipse* on a *Windows* system

The software packages used are:

- *Eclipse* (www.eclipse.org) is a software development environment that consists of an integrated development environment (IDE) and an extensible plug-in system. It is written mostly in Java and can be used to develop applications in Java and, by means of various plug-ins, in C, C++, Perl, PHP, Python, R, Ruby, and several other languages. The IDE is often called Eclipse CDT for C/C++, Eclipse JDT for Java, and Eclipse PDT for PHP.
- Apache *Hadoop* is a software framework that supports data-intensive distributed applications under a free license. *Hadoop* was inspired by Google's *MapReduce*. See Section 4.6 for a discussion of *MapReduce* and Section 4.7 for an application using *Hadoop*.
- *Cygwin* is a *Unix*-like environment for *Microsoft Windows*. It is open-source software released under the GNU General Public License version 2. The *cygwin* environment consists of (1) a dynamic-link library (DLL) as an API compatibility layer providing a substantial part of the POSIX API functionality; and (2) an extensive collection of software tools and applications that provide a *Unix*-like look and feel.

A. Prerequisites

- *Java 1.6*; set JAVA_Home = path where *JDK* is installed
- *Eclipse Europa 3.3.2*
  Note: the *Hadoop* plugin was specially designed for Europa and newer releases of *Eclipse* might have some issues with the *Hadoop* plugin.

B. SSH Installation

1. Install *cygwin* using the installer downloaded from www.cygwin.com. From the *Select Packages* window, select the *openssh* and *openssl* under Net.
   Note: Create a desktop icon when asked during installation.
2. Display the "Environment Variables" panel:

   ```
   Computer -> System Properties -> Advanced System Settings
   -> Environment Variables
   ```

   Click on the variable named *Path* and press *Edit*; append the following value to the path variable:

   ```
   ;c:\cygwin\bin;c:\cygwin\usr\bin
   ```

3. Configure the *ssh daemon* using *cygwin*. Left-click on the *cygwin* icon on the desktop and click "Run as Administrator." Type in the command window of *cygwin*:

   ```
   ssh-host-config.
   ```

4. Answer "Yes" when prompted with *sshd should be installed as a service*; answer "No" to all other questions.
5. Start the *cygwin* service by navigating to:

   ```
   Control Panel -> Administrative Tools -> Services
   ```

   Look for *cygwin sshd* and start the service.
6. Open the *cygwin* command prompt and execute the following command to generate keys:

   ```
   ssh-keygen
   ```

7. When prompted for filenames and passphrases, press Enter to accept default values. After the command has finished generating keys, enter the following command to change into your *.ssh* directory:

   ```
   cd~.ssh
   ```

8. Check to see whether the keys were indeed generated:

   ```
   ls -l
   ```

9. The two files *id_rsa.pub* and *id_rsa* with recent creation dates contain authorization keys.
10. To register the new authorization keys, enter the following command (note: the sharply-angled double brackets are very important):

    ```
    cat id_rsa.pub ≫ authorized_keys
    ```

11. Check to see whether the keys were set up correctly:

    ```
    ssh localhost
    ```

12. Since it is a new *ssh* installation, you will be warned that authenticity of the host could not be established and will be asked whether you really want to connect. Answer Yes and press Enter. You should see the `cygwin` prompt again, which means that you have successfully connected.
13. Now execute again the command:

    ```
    ssh localhost
    ```

    This time no prompt should appear.

C. Download Hadoop

1. Download *Hadoop 0.20.1* and place it in a directory such as:

   ```
   C:Java
   ```

2. Open the *cygwin* command prompt and execute:

   ```
   cd
   ```

3. Enable the home directory folder to be shown in the *Windows* Explorer window:

   ```
   explorer
   ```

4. Open another *Windows* Explorer window and navigate to the folder that contains the downloaded *Hadoop* archive.
5. Copy the *Hadoop* archive into the home directory folder.

D. Unpack Hadoop

1. Open a new *cygwin* window and execute:

```
tar -xzf hadoop-0.20.1.tar.gz
```

2. List the contents of the home directory:

```
ls -l
```

You should see a newly created directory called *Hadoop-0.20.1*. Execute:

```
cd hadoop-0.20.1
ls -l
```

You should see the files listed in Figure 11.9.

E. Set properties in configuration file

1. Open a new *cygwin* window and execute the following commands:

```
cd hadoop-0.20.1
cd conf
explorer
```

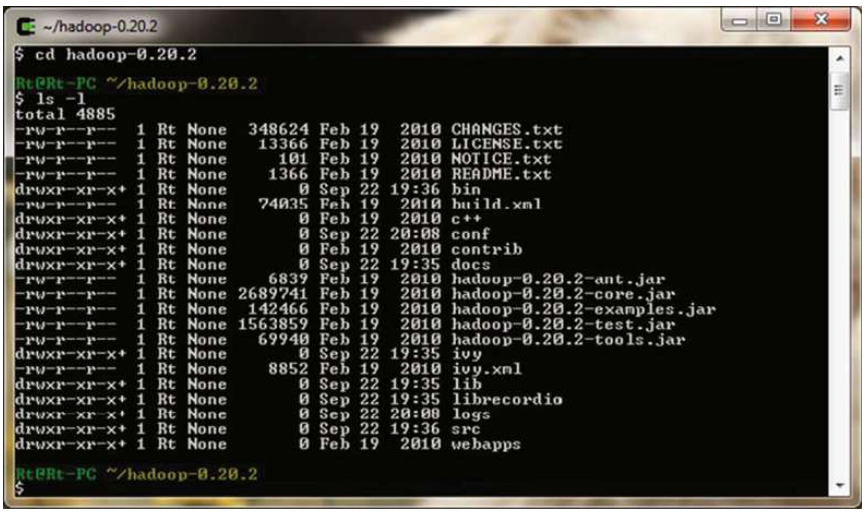

FIGURE 11.9

The result of unpacking *Hadoop*.

FIGURE 11.10

The creation of a *Hadoop* Distributed File System (HDFS).

2. The last command will cause the Explorer window for the *conf* directory to pop up. Minimize it for now or move it to the side.
3. Launch *Eclipse* or a text editor such as *Notepad* ++ and navigate to the *conf* directory. Open the *Hadoop*-site file to insert the following lines between the <*configuration*> and <*/configuration*> tags:

```
<property>
<name>fs.default.name</name>
<value>hdfs://localhost:9100</value>
</property> <property>
<name>mapred.job.tracker</name>
<value>localhost:9101</value>
</property> <property>
<name>dfs.replication</name>
<value>1</value>
</property>
```

F. Format the Namenode

Format the *namenode* to create a *Hadoop* Distributed File System (HDFS). Open a new *cygwin* window and execute the following commands:

```
cd hadoop-0.20.1
mkdir logs
bin/hadoop namenode -format
```

When the formatting of the *namenode* is finished, the message in Figure 11.10 appears.

11.10 Cloud-based simulation of a distributed trust algorithm

Mobile wireless applications are likely to benefit from cloud computing, as we discussed in Chapter 4. This expectation is motivated by several reasons:

- The convenience of data access from any site connected to the Internet.
- The data transfer rates of wireless networks are increasing; the time to transfer data to and from a cloud is no longer a limiting factor.

- Mobile devices have limited resources; whereas new generations of smartphones and tablet computers are likely to use multicore processors and have a fair amount of memory, power consumption is, and will continue to be, a major concern in the near future. Thus, it seems reasonable to delegate compute- and data-intensive tasks to an external entity, e.g., a cloud.

The first application we discuss is a cloud-based simulation for trust evaluation in a Cognitive Radio Network (CRN) [52]. The available communication spectrum is a precious commodity, and the objective of a CRN is to use the communication bandwidth effectively while attempting to avoid interference with licensed users. Two main functions necessary for the operation of a CRN are spectrum sensing and spectrum management. The former detects unused spectrum and the latter decides the optimal use of the available spectrum. Spectrum sensing in CRNs is based on information provided by the nodes of the network. The nodes compete for the free channels, and some may supply deliberately distorted information to gain advantage over the other nodes; thus, trust determination is critical for the management of CRNs.

Cognitive Radio Networks. Research over the last decade reveals a significant temporal and spatial underutilization of the allocated spectrum. Thus, there is a motivation to opportunistically harness the vacancies of spectrum at a given time and place.

The original goal of cognitive radio, first proposed at Bell Labs [246,247], was to develop a software-based radio platform that allows a reconfigurable wireless transceiver to automatically adapt its communication parameters to network availability and to user demands. Today the focus of cognitive radio is on spectrum sensing [58,161].

We recognize two types of devices connected to a CRN: primary and secondary. *Primary* nodes/devices have exclusive rights to specific regions of the spectrum; *secondary* nodes/devices enjoy dynamic spectrum access and are able to use a channel, provided that the primary, licensed to use that channel, is not communicating. Once a primary starts its transmission, the secondary using the channel is required to relinquish it and identify another free channel to continue its operation. This mode of operation is called an *overlay mode.*

CRNs are often based on a *cooperative spectrum-sensing* strategy. In this mode of operation, each node determines the occupancy of the spectrum based on its own measurements, combined with information from its neighbors, and then shares its own spectrum occupancy assessment with its neighbors [129,339,340].

Information sharing is necessary because a node alone cannot determine the true spectrum occupancy. Indeed, a secondary node has a limited transmission and reception range; node mobility combined with typical wireless channel impairments, such as multipath fading, shadowing, and noise, add to the difficulty of gathering accurate information by a single node.

Individual nodes of a centralized or infrastructure-based CRN send the results of their measurements regarding spectrum occupancy to a central entity, whether a base station, an access point, or a cluster head. This entity uses a set of *fusion rules* to generate the spectrum occupancy report and then distributes it to the nodes in its jurisdiction. The area covered by such networks is usually small since global spectrum decisions are affected by the local geography.

There is another mode of operation based on the idea that a secondary node operates at a much lower power level than a primary one. In this case the secondary can share the channel with the primary as

long as its transmission power is below a threshold, μ, that has to be determined periodically. In this scenario the receivers that want to listen to the primary are able to filter out the "noise" caused by the transmission initiated by secondaries if the signal-to-noise ratio, (S/N), is large enough.

We are only concerned with the overlay mode whereby a secondary node maintains an *occupancy report*, which gives a snapshot of the current status of the channels in the region of the spectrum it is able to access. The occupancy report is a list of all the channels and their state, e.g., 0 if the channel is free for use and 1 if the primary is active. Secondary nodes continually sense the channels available to them to gather accurate information about available channels.

The secondary nodes of an ad hoc CRN compete for free channels, and the information one node may provide to its neighbors could be deliberately distorted. Malicious nodes will send false information to the fusion center in a centralized CRN. Malicious nodes could attempt to deny the service or to cause other secondary nodes to violate spectrum allocation rules. To *deny the service*, a node will report that free channels are used by the primary. To entice the neighbors to commit Federal Communication Commission (FCC) violations, the occupancy report will show that channels used by the primary are free. This attack strategy is called a *secondary spectrum data falsification (SSDF)*, or Byzantine, attack.[5] Thus, trust determination is a critical issue for CR networks.

Trust. The actual meaning of *trust* is domain and context specific. Consider, for example, networking; at the MAC layer the multiple-access protocols assume that all senders follow the channel access policy, e.g., in Carrier Sense Multiple Access with Collision Detection (CSMA-CD) a sender senses the channel and then attempts to transmit if no one else does. In a store-and-forward network, trust assumes that all routers follow a best-effort policy to forward packets toward their destination.

In the context of cognitive radio, trust is based on the quality of information regarding the channel activity provided by a node. The status of individual channels can be assessed by each node based on the results of its own measurements, combined with the information provided by its neighbors, as is the case of several algorithms discussed in the literature [68,339].

The alternative discussed in Section 11.11 is to have a cloud-based service that collects information from individual nodes, evaluates the state of each channel based on the information received, and supplies this information on demand. Evaluation of the trust and identification of untrustworthy nodes are critical for both strategies [284].

A Distributed Algorithm for Trust Management in Cognitive Radio. The algorithm computes the trust of node $1 \leqslant i \leqslant n$ in each node in its vicinity, $j \in V_i$, and requires several preliminary steps. The basic steps executed by a node i at time t are:

1. Determine node i's version of the occupancy report for each one of the K channels:

$$S_i(t) = \{s_{i,1}(t), s_{i,2}(t), \ldots, s_{i,K}(t)\} \tag{11.1}$$

   In this step node i measures the power received on each of the K channels.
2. Determine the set $V_i(t)$ of the nodes in the vicinity of node i. Node i broadcasts a message and individual nodes in its vicinity respond with their `NodeId`.
3. Determine the distance to each node $j \in V_i(t)$ using the algorithm described in this section.

[5] See Section 2.11 for a brief discussion of Byzantine attacks.

4. Infer the power as measured by each node $j \in V_i(t)$ on each channel $k \in K$.
5. Use the location and power information determined in the previous two steps to infer the status of each channel:
$$s_{i,k,j}^{infer}(t), \quad 1 \leqslant k \leqslant K, \quad j \in V_i(t). \tag{11.2}$$
A secondary node j should have determined 0 if the channel is free for use, 1 if the primary node is active, and X if it cannot be determined.
$$s_{i,k,j}^{infer}(t) = \begin{cases} 0 & \text{if secondary node } j \text{ decides that channel } k \text{ is free.} \\ 1 & \text{if secondary node } j \text{ decides that channel } k \text{ is used by the primary.} \\ X & \text{if no inference can be made.} \end{cases} \tag{11.3}$$
6. Receive the information provided by neighbor $j \in V_i(t)$, $S_{i,k,j}^{recv}(t)$.
7. Compare the information provided by neighbor $j \in V_i(t)$:
$$S_{i,k,j}^{recv}(t) = \left\{ s_{i,1,j}^{recv}(t),\ s_{i,2,j}^{recv}(t), \ldots,\ s_{i,K,j}^{recv}(t) \right\} \tag{11.4}$$
with the information inferred by node i about node j:
$$S_{i,k,j}^{infer}(t) = \left\{ s_{i,1,j}^{infer}(t),\ s_{i,2,j}^{infer}(t), \ldots,\ s_{i,K,j}^{infer}(t) \right\}. \tag{11.5}$$
8. Compute the number of matches, mismatches, and cases when no inference is possible, respectively,
$$\alpha_{i,j}(t) = \mathcal{M}\left[S_{i,k,j}^{infer}(t), \quad S_{i,k,j}^{recv}(t) \right], \tag{11.6}$$
with $\mathcal{M}$ the number of matches between the two vectors and
$$\beta_{i,j}(t) = \mathcal{N}\left[S_{i,k,j}^{infer}(t), \quad S_{i,k,j}^{recv}(t) \right], \tag{11.7}$$
with $\mathcal{N}$ the number of mismatches between the two vectors, and $X_{i,j}(t)$ the number of cases where no inference could be made.
9. Use the quantities $\alpha_{i,j}(t)$, $\beta_{i,j}(t)$, and $X_{i,j}(t)$ to assess the trust in node j. For example, compute the trust of node i in node j at time t as
$$\zeta_{i,j}(t) = \left[1 + X_{i,j}(t)\right] \frac{\alpha_{i,j}(t)}{\alpha_{i,j}(t) + \beta_{i,j}(t)}. \tag{11.8}$$

Simulation of the Distributed Trust Algorithm. The cloud application is a simulation of a CRN to assess the effectiveness of a particular trust assessment algorithm. Multiple instances of the algorithm run concurrently on an *AWS* cloud. The area where the secondary nodes are located is partitioned into several overlapping subareas, as shown in Figure 11.11. The secondary nodes are identified by an instance Id, *iId*, as well as a global Id, *gId*. The simulation assumes that the primary nodes cover the entire area; thus, their position is immaterial.

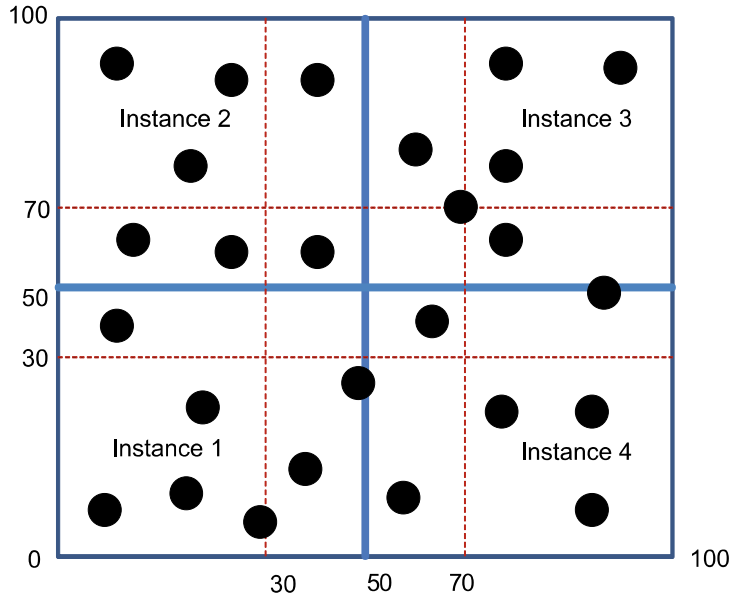

FIGURE 11.11

Data partitioning for the simulation of a trust algorithm. The area covered is of size 100×100 units. The nodes in the four subareas of size 70×70 units are processed by an instance of the cloud application. The subareas allocated to an instance overlap to allow an instance to have all the information about a node in its coverage area.

The simulation involves a controller and several cloud instances. In its initial implementation, the controller runs on a local system under *Linux Ubuntu* 10.04 LTS. The controller supplies the data, the trust program, and the scripts to the cloud instances; the cloud instances run under the Basic 32-bit *Linux* image on *AWS*, the so-called `t1.micro`. The instances run the actual trust program and compute the instantaneous trust inferred by a neighbor; the results are then processed by an *awk*[6] script to compute the average trust associated with a node as seen by all its neighbors. On the next version of the application the data is stored on the cloud using the *S3* service, and the controller also runs on the cloud.

In the simulation discussed here, the nodes with

$$gId = \{1, 3, 6, 8, 12, 16, 17, 28, 29, 32, 35, 38, 39, 43, 44, 45\} \tag{11.9}$$

were programmed to be dishonest. The results show that the nodes programmed to act maliciously have a trust value lower than that of the honest nodes; their trust value is always lower than 0.6 and, in many instances, lower than 0.5 (see Figure 11.12). We also observe that the node density affects the accuracy of the algorithm; the algorithm predicts more accurately the trust in densely populated areas. As expected, nodes with no neighbors are unable to compute the trust.

[6]The *awk* utility is based on a scripting language and used for text processing; in this application it is used to produce formatted reports.

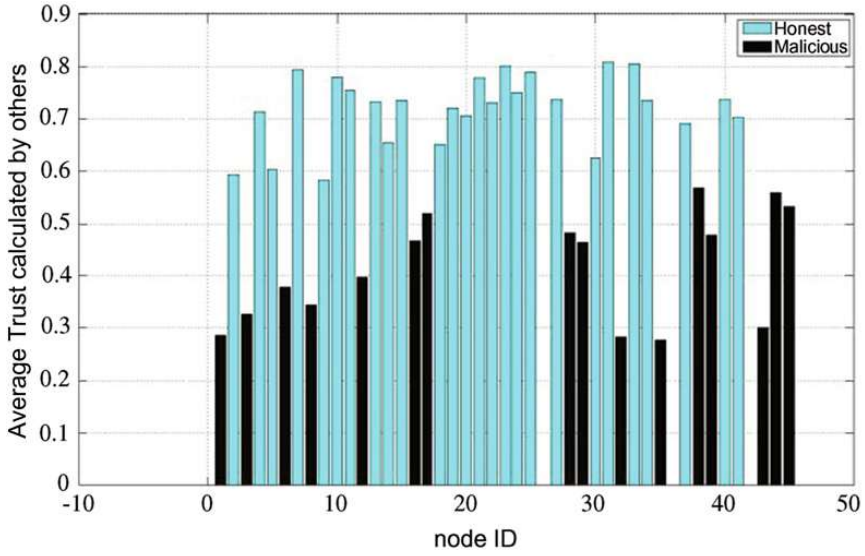

FIGURE 11.12

The trust values computed using the distributed trust algorithm. The secondary nodes programmed to act maliciously have a trust value less than 0.6 and many less than 0.5, lower than that of the honest nodes.

In practice the node density is likely to be nonuniform, high in a crowded area such as a shopping mall, and considerably lower in surrounding areas. This indicates that when the trust is computed using the information provided by all secondary nodes, we can expect higher accuracy of the trust determination in higher density areas.

11.11 A trust management service

The cloud service discussed in this section, see also [52], is an alternative to the distributed trust management scheme analyzed in Section 11.10. mobile devices are ubiquitous nowadays and their use will continue to increase. Clouds are emerging as the computing and storage engines of the future for a wide range of applications. There is a symbiotic relationship between the two; mobile devices can consume as well as produce very large amounts of data, whereas computer clouds have the capacity to store and deliver such data to the user of a mobile device. To exploit the potential of this symbiotic relationship, we propose a new cloud service for the management of wireless networks.

Mobile devices have limited resources; new generations of smartphones and tablet computers are likely to use multicore processors and have a fair amount of memory, but power consumption is still and will continue to be a major concern; thus, it seems reasonable to delegate and data-intensive tasks to the cloud. The motivation for this application is to reduce the power consumption of the mobile devices.

Transferring computations related to CRN management to a cloud supports the development of new, possibly more accurate, resource management algorithms. For example, algorithms to discover

communication channels currently in use by a primary transmitter could be based on past history but are not feasible when the trust is computed by the mobile device. Such algorithms require massive amounts of data and can also identify malicious nodes with high probability.

Mobile devices such as smartphones and tablets are able to communicate using two networks: (i) a cellular wireless network; and (ii) a Wi-Fi network. The service we propose assumes that a mobile device uses the cellular wireless network to access the cloud, whereas the communication over the Wi-Fi channel is based on cognitive radio (CR). The amount of data transferred using the cellular network is limited by the subscriber's data plan, but no such limitation exists for the Wi-Fi network. The cloud service, discussed next, will allow mobile devices to use the Wi-Fi communication channels in a cognitive radio network environment and will reduce the operating costs for end users.

Although the focus of our discussion is on trust management for CRNs, the cloud service we propose can be used for tasks other than bandwidth management; for example, routing in mobile ad hoc networks, detection and isolation of noncooperative nodes, and other network management and monitoring functions could benefit from the identification of malicious nodes.

Model Assumptions. The cognitive radio literature typically analyzes networks with a relatively small number of nodes and active in a limited geographic area; thus, all nodes in the network sense the same information on channel occupancy. Channel impairments, such as signal fading, noise, and so on cause errors and lead trustworthy nodes to report false information. We consider networks with a much larger number of nodes distributed over a large geographic area; as the signal strength decays with the distance, we consider several rings around a primary tower. We assume a generic fading model given by the following expression:

$$\gamma_k^i = T_k \times \frac{A^2}{s_{ik}^{\alpha}} \tag{11.10}$$

where γ_k^i is the received signal strength on channel k at location of node i, A is the frequency constant, $2 \leqslant \alpha \leqslant 6$ is path loss factor, s_{ik}^{α} is the distance between primary tower P_k and node i, and T_k is the transition power of primary tower P_k transmitting on channel k.

In our discussion we assume that there are K channels labeled $1, 2, \ldots, K$ and a primary transmitter P^k transmits on channel k. The algorithm is based on several assumptions regarding the secondary nodes, the behavior of malicious nodes, and the geometry of the system. First, we assume that the secondary nodes:

- Are mobile devices; some are slow-moving, others are fast-moving.
- Cannot report their position because they are not equipped with a global positioning system (GPS).
- The clocks of the mobile devices are not synchronized.
- The transmission and reception range of a mobile device can be different.
- The transmission range depends on the residual power of each mobile device.

We assume that the malicious nodes in the network are a minority and their behavior is captured by the following assumptions:

- The misbehaving nodes are malicious rather than selfish; their only objective is to hinder the activity of other nodes whenever possible, a behavior distinct from the one of selfish nodes motivated to gain some advantage.

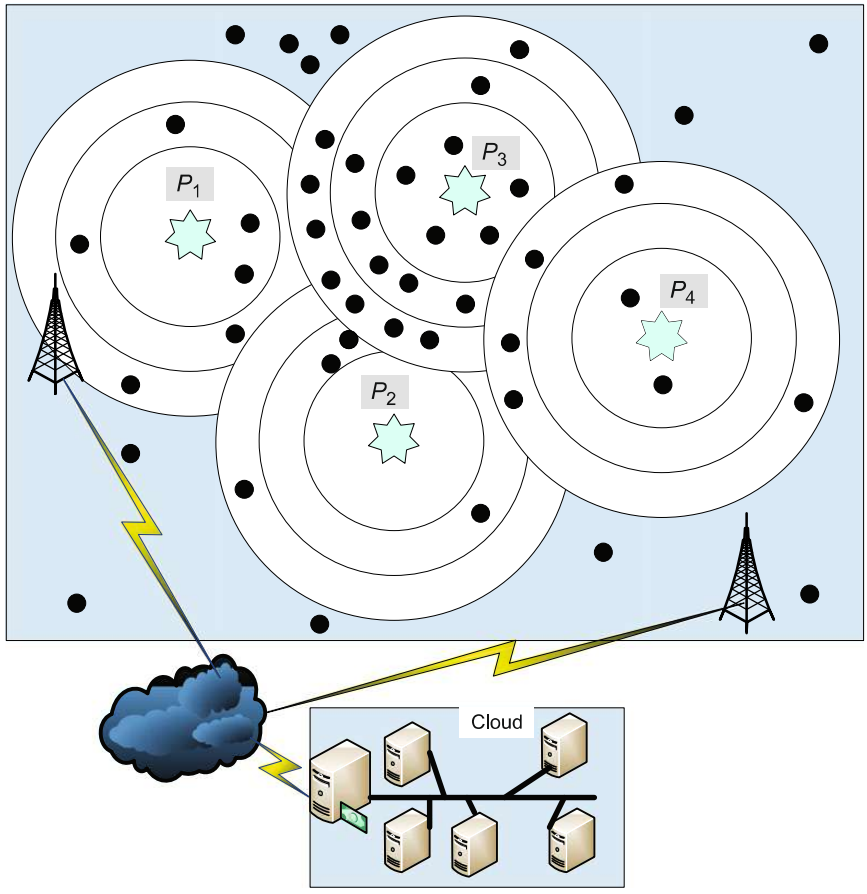

FIGURE 11.13

Schematic representation of a CRN layout: four primary nodes, P_1–P_4, a number of mobile devices, two towers for a cellular network and a cloud are shown. Not shown are the hotspots for the Wi-Fi network.

- The malicious nodes are uniformly distributed in the area we investigate.
- The malicious nodes do not collaborate in their attack strategies.
- The malicious nodes change the intensity of their Byzantine attack in successive time slots. Similar patterns of malicious behavior are easy to detect, and an intelligent attacker is motivated to avoid detection.

The geometry of the system is captured by Figure 11.13. We distinguish primary and secondary nodes and the cell towers used by the secondary nodes to communicate with service running on the cloud.

We use a majority voting rule for a particular ring around a primary transmitter. The global decision regarding the occupancy of a channel requires a majority of the votes. Since the malicious nodes are a minority and they are uniformly distributed, the malicious nodes in any ring are also a minority; thus, a ring-based majority fusion is a representative of accurate occupancy for the channel associated with the ring.

All secondary nodes are required to register first and then to transmit periodically their current power level, as well as their occupancy report for each one of the K channels. As mentioned in the introductory discussion, the secondary nodes connect to the cloud using the cellular network. After a mobile device is registered, the cloud application requests the cellular network to detect its location. The towers of the cellular network detect the location of a mobile device by triangulation with an accuracy that is a function of the environment and is of the order of 10 meters. The location of the mobile device is reported to the cloud application every time it provides an occupancy report.

The nodes that do not participate in the trust computation will not register in this cloud-based version of the resource management algorithm; thus, they do not get the occupancy report and cannot use it to identify free channels. Obviously, if a secondary node does not register, it cannot influence other nodes and prevent them from using free channels, or tempt them to use busy channels.

In the registration phase a secondary node transmits its MAC address and the cloud responds with the tuple (Δ, δ_s). Here, Δ is the time interval between two consecutive reports, chosen to minimize the communication as well as the overhead for sensing the status of each channel. To reduce the communication overhead, secondary nodes should only transmit the changes from the previous status report. $\delta_s < \Delta$ is the time interval to the first report expected from the secondary node. This scheme provides a pseudo-synchronization so that the data collected by the cloud, and used to determine the trust is, based on observations made by the secondary nodes at about the same time.

An Algorithm for Trust Evaluation Based on Historical Information. The cloud computes the probable distance d_i^k of each secondary node i from the known location of a primary transmitter, P^k. Based on signal attenuation properties we conceptualize N circular rings centered at the primary, where each ring is denoted by $\mathcal{R}_r^k$, with $1 \leqslant r \leqslant N$ the ring number. The radius of a ring is based on the distance d_r^k to the primary transmitter P^k. A node at a distance $d_i^k \leqslant d_1^k$ is included in the ring $\mathcal{R}_1^k$, nodes at distance $d_1^k < d_i^k \leqslant d_2^k$ are included in the ring $\mathcal{R}_2^k$, and so on. The closer to the primary, the more accurate the channel occupancy report of the nodes in the ring should be. Call n_r^k the number of nodes in ring $\mathcal{R}_r^k$.

At each report cycle at time t_q, the cloud computes the occupancy report for channel $1 \leqslant k \leqslant K$ used by primary transmitter P^k. The status of channel k reported by node $i \in \mathcal{R}_r^k$ is denoted as $s_i^k(t_q)$. Call $\sigma_{one}^k(t_q)$ the count of the nodes in the ring $\mathcal{R}_r^k$ reporting that the channel k is not free (reporting $s_i^k(t_q) = 1$) and $\sigma_{zero}^k(t_q)$ the count of those reporting that the channel is free (reporting $s_i^k(t_q) = 0$):

$$\sigma_{one}^k(t_q) = \Sigma_{i=1}^{n_r^k} s_i^k(t_q) \quad \text{and} \quad \sigma_{zero}^k(t_q) = n_r^k - \sigma_{one}^k(t_q). \tag{11.11}$$

Then, the status of channel k reported by the nodes in the ring R_r^k is determined by majority voting as

$$\sigma_{R_r}^k(t_q) \begin{cases} = 1 \text{ when } \sigma_{one}^k(t_q) \geqslant \sigma_{zero}^k(t_q). \\ = 0 \text{ otherwise.} \end{cases} \tag{11.12}$$

To determine the trust in node i we compare $s_i^k(t_q)$ with $\sigma_{R_r}^k(t_q)$; call $\alpha_{i,r}^k(t_q)$ and $\beta_{i,r}^k(t_q)$ the number of matches and, respectively, mismatches in this comparison for each node in the ring R_r^k. We repeat this procedure for all rings around P^k and construct

$$\alpha_i^k(t_q) = \Sigma_{r=1}^{n_r^k} \alpha_{i,r}^k(t_q) \quad \text{and} \quad \beta_i^k(t_q) = \Sigma_{r=1}^{n_r^k} \beta_{i,r}^k(t_q). \tag{11.13}$$

Node i will report the status of the channels in the set $C_i(t_q)$, the channels with index $k \in C_i(t_q)$; then, the quantities $\alpha_i(t_q)$ and $\beta_i(t_q)$ with $\alpha_i(t_q) + \beta_i(t_q) = |C_i(t_q)|$ are

$$\alpha_i(t_q) = \Sigma_{k \in C_i} \alpha_i^k(t_q) \quad \text{and} \quad \beta_i(t_q) = \Sigma_{k \in C_i} \beta_i^k(t_q). \tag{11.14}$$

Finally, the global trust in node i is

$$\zeta_i(t_q) = \frac{\alpha_i(t_q)}{\alpha_i(t_q) + \beta_i(t_q)}. \tag{11.15}$$

The trust in each node at each iteration is determined using a similar strategy to the one discussed earlier. Its status report, $S_j(t)$, contains only information about the channels it can report on, and only if the information has changed from the previous reporting cycle.

Then, a statistical analysis of the random variables for a window of time W, $\zeta_j(t_q)$, $t_q \in W$ allows us to compute the moments as well as a 95% confidence interval. Based on these results we assess whether node j is trustworthy and eliminate the untrustworthy nodes when we evaluate the occupancy map at the next cycle. We continue to assess the trustworthiness of all nodes and may accept the information from node j when its behavior changes.

Let's now discuss the use of historical information to evaluate trust. We assume a sliding window $W(t_q)$ consists of n_w time slots. Given two decay constants $k_1$and k_2, with $k_1 + k_2 = 1$, we use an exponential averaging that gives decreasing weight to old observations. We choose $k_1 \ll k_2$ to give more weight to the past actions of a malicious node. Such nodes attack only intermittently and try to disguise their presence with occasional good reports; the misbehavior should affect the trust more than the good actions. The history-based trust requires the determination of two quantities:

$$\alpha_i^H(t_q) = \Sigma_{i=0}^{n_w-1} \alpha_i(t_q - i\tau) k_1^i \quad \text{and} \quad \beta_i^H(t_q) = \Sigma_{i=0}^{n_w-1} \beta_i(t_q - i\tau) k_2^i. \tag{11.16}$$

Then, the history-based trust for node i valid only at times $t_q \geqslant n_w\tau$ is:

$$\zeta_i^H(t_q) = \frac{\alpha_i^H(t_q)}{\alpha_i^H(t_q) + \beta_i^H(t_q)}. \tag{11.17}$$

For times $t_q < n_w\tau$ the trust will be based only on a subset of observations rather than a full window on n_w observations.

This algorithm can also be used in regions in which the cellular infrastructure is missing. An ad hoc network could allow the nodes that cannot connect directly to the cellular network to forward their information to nodes closer to the towers and then to the cloud-based service.

Simulation of the History-Based Algorithm for Trust Management. The aim of the history-based trust evaluation is to distinguish between trustworthy and malicious nodes. We expect the ratio of malicious to trustworthy nodes as well as node density to play an important role in this decision. The node density ρ is the number of nodes per unit of the area. In our simulation experiments the size of the area is constant but the number of nodes increases from 500 to 2,000; thus, the node density increases by a factor of four. The ratio of the number of malicious to the total number of nodes varies between $\alpha = 0.2$ and a worst case of $\alpha = 0.6$.

The performance metrics we consider are as follows: the average trust for all nodes, the average trust of individual nodes, and the error of honest/trustworthy nodes. We want to see how the algorithm behaves when the density of the nodes increases, so we consider four cases with 500, 1,000, 1,500, and 2,000 nodes on the same area. Thus, we allow the density to increase by a factor of four. We also investigate the average trust when α, the ratio of malicious nodes to the total number of nodes, increases from $\alpha = 0.2$ to $\alpha = 0.4$ and, finally, to $\alpha = 0.6$.

This straightforward data-partitioning strategy for the distributed trust management algorithm is not a reasonable one for the centralized algorithm, because it would lead to excessive communication among the cloud instances. Individual nodes may contribute data regarding primary transmitters in a different subarea; to evaluate the trust of each node, the cloud instances would have to exchange a fair amount of information. This data partitioning would also complicate our algorithm, which groups together secondary nodes based on their distance from the primary one.

Instead, we allocate to each instance a number of channels, and all instances share the information about the geographic position of each node. The distance of a secondary node to any primary one can then be easily computed. This data-partitioning strategy scales well in the number of primaries. Thus, it is suitable for simulation in large metropolitan areas, but may not be able to accommodate cases when the number of secondaries is on the order of 10^8–10^9.

The objective of our studies is to understand the limitations of the algorithm; the aim of the algorithm is to distinguish between trustworthy and malicious nodes. We expect that the ratio of malicious to trustworthy nodes, as well as the node density should play an important role in this decision. The measures we examine are the average trust for all nodes, as well as the average trust of individual nodes.

The effect of the malicious versus trustworthy node ratio on the average trust. We report the effect of the malicious versus trustworthy node ratio on the average trust when the number of nodes increases. The average trust is computed separately for the two classes of nodes and allows us to determine whether the algorithm is able to clearly separate them.

Recall that the area is constant; thus, when the number of nodes increases, so does the node density. First we consider two extreme cases: the malicious nodes represent only 20% of the total number of nodes and an unrealistically high presence, 60%. Then we report on the average trust when the number of nodes is fixed and the malicious nodes represent an increasing fraction of the total number of nodes.

Results reported in [52] show that when the malicious nodes represent only 20% of all nodes, there is a clear distinction between the two groups. The malicious nodes have an average trust of 0.28 and trustworthy nodes have an average trust index of 0.91, regardless of the number of nodes.

When the malicious nodes represent 60% of all the nodes, the number of nodes plays a significant role; when the number of nodes is small, the two groups cannot be distinguished, so their average trust index is almost equal, 0.55, although the honest nodes have a slightly larger average trust value. When the number of nodes increases to 2,000 and node density increases fourfold, the average trust of the malicious group decreases to 0.45 and for the honest group it increases to about 0.68.

This result is not unexpected; it only shows that the history-based algorithm is able to classify the nodes properly, even when the malicious nodes are a majority, a situation we do not expect to encounter in practice. This effect is somewhat surprising; we did not expect that under these extreme conditions the average of the trust of all nodes would be so different for the two groups. A possible explanation is that our strategy to reward constant good behavior rather than occasional good behavior, designed to mask the true intentions of a malicious node, works well.

Figures 11.14(a) and (b) shows the average trust function of α, the ratio of malicious versus total number of nodes. The results confirm the behavior discussed earlier. We see a clear separation of the two classes only when the malicious nodes are in the minority. When the density of malicious nodes approaches a high value so that they are in the majority, the algorithm still performs, as is evident from the figures. The average trust for honest nodes even at high value of α is larger than the trust of malicious nodes. Thus, the trusts allows the identification of malicious nodes. We also observe that the distinction between the two classes of nodes is more clear when the number of nodes in the network increases.

The benefits of a cloud-based service for trust management. A cloud service for trust management in cognitive networks can have multiple technical as well as economic benefits [74]. The service is likely to have a broader impact than the one discussed here, and it could be used to support a range of important policies in a wireless network where many decisions require the cooperation of all nodes. A history-based algorithm to evaluate the trust and detect malicious nodes with high probability is at the center of the solution we have proposed [52].

A centralized, history-based algorithm for bandwidth management in CRNs has several advantages over the distributed algorithms discussed in the literature:

- Drastically reduces the computations a mobile device is required to carry out to identify free channels and avoid penalties associated with interference with primary transmitters.
- Allows a secondary node to get information about channel occupancy as soon as it joins the system, and later on demand. This information is available even when a secondary node is unable to receive reports from its neighbors, or when it is isolated.
- Does not require the large number of assumptions critical to the distributed algorithms.
- The dishonest nodes can be detected with high probability and their reports can be ignored; thus, over time the accuracy of the results increases. Moreover, historic data could help detect a range of Byzantine attacks orchestrated by a group of malicious nodes.
- Is very likely to produce more accurate results than the distributed algorithm because the reports are based on information from all secondary nodes reporting on a communication channel used by a primary, not only those in its vicinity; a higher node density increases the accuracy of the predictions. The accuracy of the algorithm is a function of the frequency of the occupancy reports provided by the secondary nodes.

The centralized trust management scheme has several other advantages. First, it can be used not only to identify malicious nodes and provide channel occupancy reports, but also to manage the allocation of free channels. In the distributed case, two nodes may attempt to use a free channel and collide; this situation is avoided in the centralized case. At the same time, malicious nodes can be identified with high probability and be denied access to the occupancy report.

The server could also collect historic data regarding the pattern of behavior of the primary nodes and use this information for the management of free channels. For example, when a secondary requests access for a specific length of time, the service may attempt to identify a free channel likely to be available for that time.

The trust management may also be extended to other network operations such as routing in a mobile ad hoc network; the strategy in this case would be to avoid routing through malicious nodes.

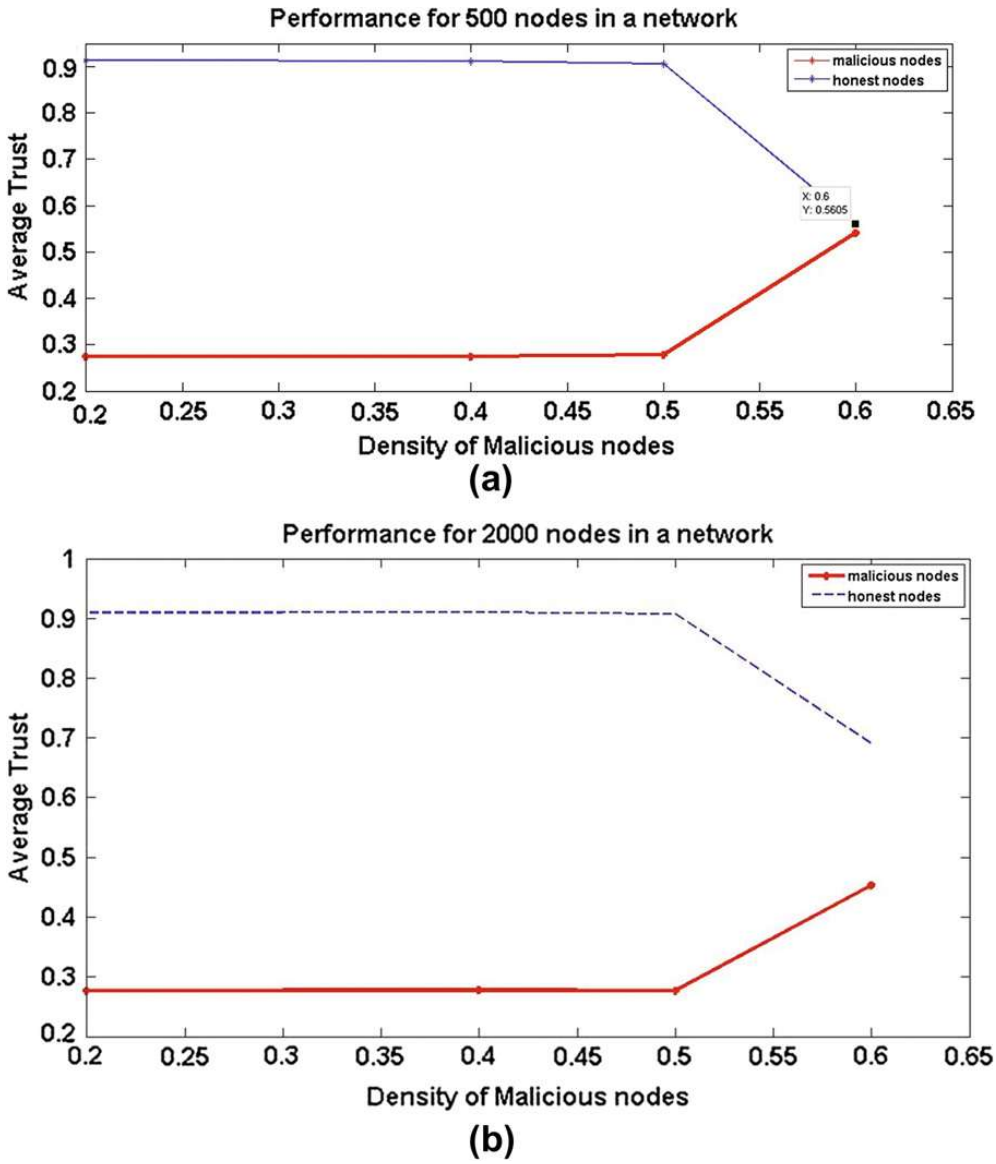

FIGURE 11.14

The average trust function of α when the population size increases: (a) 500; (b) 2,000 nodes. As long as malicious nodes represent 50% or less of the total number of nodes, the average trust of malicious nodes is below 0.3, whereas the average trust of trustworthy nodes is above 0.9 in a scale of 0 to 1.0. As the number of nodes increases, the distance between the average trust of the two classes becomes larger, and even larger when $\alpha > 0.5$., i.e., when the malicious nodes are in the majority.

11.12 A cloud service for adaptive data streaming

In this section we discuss a cloud application related to data streaming [288]. Data streaming is the name given to the transfer of data at a high rate with real-time constraints. Multimedia applications such as music and video streaming, high-definition television (HDTV), scientific applications that process a continuous stream of data collected by sensors, the continuous backup copying to a storage medium of the data flow within a computer, and many other applications require the transfer of real-time data at a high rate. For example, to support real-time human perception of the data, multimedia applications have to make sure that enough data is being continuously received without any noticeable time lag.

We are concerned with the case when data streaming involves a multimedia application connected to a service running on a computer cloud. The stream could originate from the cloud, as is the case of the iCloud service provided by Apple, or could be directed toward the cloud, as in the case of a real-time data collection and analysis system.

Data streaming involves three entities: the sender, a communication network, and a receiver. The resources necessary to guarantee the timing constraints include CPU cycles and buffer space at the sender and the receiver, as well as network bandwidth. Adaptive data streaming determines the data rate based on the available resources. Lower data rates imply lower quality, but they reduce the demands for system resources.

Adaptive data streaming is possible only if the application permits tradeoffs between quantity and quality. Such tradeoffs are feasible for audio and video streaming, which allow lossy compression, but are not acceptable for many applications that process a continuous stream of data collected by sensors.

Data streaming requires accurate information about all resources involved, and this implies that the network bandwidth has to be constantly monitored; at the same time, the scheduling algorithms should be coordinated with memory management to guarantee the timing constraints. Adaptive data streaming poses additional constraints because the data flow is dynamic. Indeed, once we detect that the network cannot accommodate the data rate required by an audio or video stream, we have to reduce the data rate; thus, to convert to a lower quality audio or video. Data conversion can be done on the fly and, in this case, the data flow on the cloud has to be changed.

Accommodating dynamic data flows with timing constraints is nontrivial; only about 18% of the top 100 global video Web sites use *adaptive bit rate* (ABR) technologies for streaming [336].

This application stores the music files in *S3* buckets, and the audio service runs on the *EC2* platform. In *EC2* each virtual machine functions as a virtual private server and is called an *instance;* an instance specifies the maximum amount of resources available to an application, the interface for that instance, and the cost per hour.

EC2 allows the import of virtual machine images from the user environment to an instance through a facility called *VM import.* It also distributes automatically the incoming application traffic among multiple instances using the *elastic load-balancing* facility. *EC2* associates an *elastic IP address* with an account; this mechanism allows a user to mask the failure of an instance and remap a public IP address to any instance of the account, without the need to interact with the software support team.

Adaptive audio streaming involves a multi-objective optimization problem. We want to convert the highest-quality audio file stored on the cloud to a resolution corresponding to the rate that can be sustained by the available bandwidth; at the same time, we want to minimize the cost on the cloud site

and minimize the buffer requirements for the mobile device to accommodate the transmission jitter. Finally, we want to reduce to a minimum the startup time for the content delivery.

A first design decision is whether data streaming should only begin after the conversion from the WAV to MP3 format has been completed or it should proceed concurrently with conversion – in other words, start as soon as several MP3 frames have been generated. Another question is whether the converted music file should be saved for later use or discarded.

To answer these questions, we experimented with conversion from the highest-quality audio files, which require a 320 Kbps data rate, to lower-quality files corresponding to 192, 128, 64, 32, and finally 16 Kbps. If the conversion time is small and constant there is no justification for pipelining data conversion and streaming, a strategy that complicates the processing flow on the cloud. It makes sense to cache the converted copy for a limited period of time with the hope that it will be reused in the future.

Another design decision is how the two services should interact to optimize performance. Two alternatives come to mind:

1. The audio service running on the *EC2* platform requests the data file from the *S3*, converts it, and, eventually, sends it back. The solution involves multiple delays and it is far from optimal.
2. Mount the *S3 bucket* as an *EC2* drive. This solution reduces considerably the start-up time for audio streaming.

The conversion from a high-quality audio file to a lower-quality, thus a lower-bit-rate, file is performed using the LAME library.

The conversion time depends on the desired bitrate and the size of the original file. Tables 11.2, 11.3, 11.4, and 11.5 show the conversion time in seconds when the source MP3 files are of 320 Kbps and 192 Kbps, respectively. The sizes of the input files are also shown.

The platforms used for conversion are (a) the *EC2 t1.micro* server for the measurements reported in Tables 11.2 and 11.3 and (b) the *EC2 c1.medium* for the measurements reported in Tables 11.4 and 11.5. The instances run the *Ubuntu Linux* operating system.

The results of our measurements when the instance is the *t1.micro* server exhibit a wide range of conversion times, 13–80 seconds, for the large audio file of about 6.7 MB when we convert from 320 to 192 Kbps. A wide range, 13–64 seconds, is also observed for an audio file of about 4.5 MB when

Table 11.2 Conversion time in seconds on a *EC2 t1.micro* server platform. The source file is of the highest audio quality, 320 Kbps. The individual conversions are labeled $C1$ to $C10$; $\overline{T}_c$ is the mean conversion time.

Bit Rate (Kbps)	Audio File Size (MB)	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	$\overline{T}_c$
192	6.701974	73	43	19	13	80	42	33	62	66	36	46.7
128	4.467982	42	46	64	48	19	52	52	48	48	13	43.2
64	2.234304	9	9	9	9	10	26	43	9	10	10	14.4
32	1.117152	7	6	14	6	6	7	7	6	6	6	7.1
16	0.558720	4	4	4	4	4	4	4	4	4	4	4

Table 11.3 Conversion time in seconds on a *EC2 t1.micro* server platform. The source file is of high audio quality, 192 Kbps. The individual conversions are labeled $C1$ to $C10$; $\overline{T}_c$ is the mean conversion time.

Bit Rate (Kbps)	Audio File Size (MB)	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	$\overline{T}_c$
128	4.467982	14	15	13	13	73	75	56	59	72	14	40.4
64	2.234304	9	9	9	32	44	9	23	9	45	10	19.9
32	1.117152	6	6	6	6	6	6	20	6	6	6	7.4
16	0.558720	6	6	6	6	6	6	20	6	6	6	5.1

Table 11.4 Conversion time T_c in seconds on a *EC2 c1.medium* platform. The source file is of the highest audio quality, 320 Kbps. The individual conversions are labeled $C1$ to $C10$; $\overline{T}_c$ is the mean conversion time.

Bit Rate (Kbps)	Audio File Size (MB)	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	$\overline{T}_c$
192	6.701974	15	15	15	15	15	15	15	15	15	15	15
128	4.467982	15	15	15	15	15	15	15	15	15	15	15
64	2.234304	11	11	11	11	11	11	11	11	11	11	11
32	1.117152	7	7	7	7	7	7	7	7	7	7	7
16	0.558720	4	4	4	4	4	4	4	4	4	4	4

Table 11.5 Conversion time in seconds on a *EC2 c1.medium* platform. The source file is of high audio quality, 192 Kbps. The individual conversions are labeled $C1$ to $C10$; $\overline{T}_c$ is the mean conversion time.

Bit Rate (Kbps)	Audio File Size (MB)	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	$\overline{T}_c$
128	4.467982	15	15	15	15	15	15	15	15	15	15	15
64	2.234304	10	10	10	10	10	10	10	10	10	10	10
32	1.117152	7	7	7	7	7	7	7	7	7	7	7
16	0.558720	4	4	4	4	4	4	4	4	4	4	4

we convert from 320 to 128 Kbps. For poor-quality audio the file size is considerably smaller, about 0.56 MB, and the conversion time is constant and small, 4 seconds. Figure 11.15 shows the average conversion time for the experiments summarized in Tables 11.2 and 11.3. It is somewhat surprising that

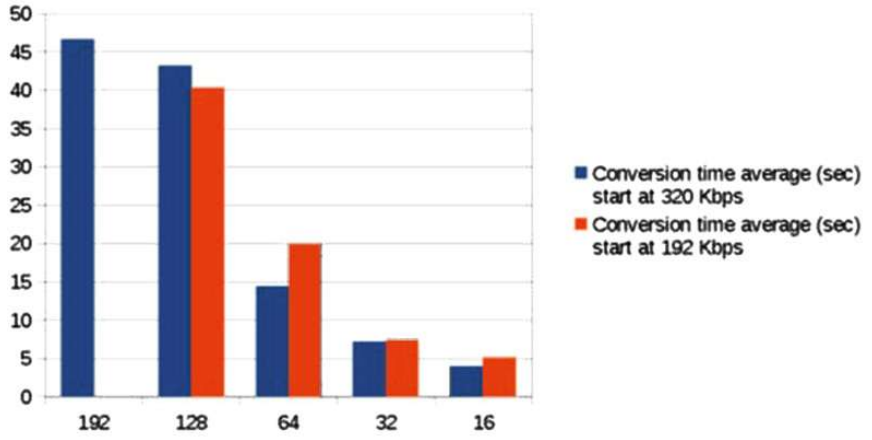

FIGURE 11.15

The average conversion time on a *EC2 t1.micro* platform. The bars at left and right correspond to the original file at the highest resolution (320 Kbps data rate) and next highest resolution (192 Kbps data rate), respectively.

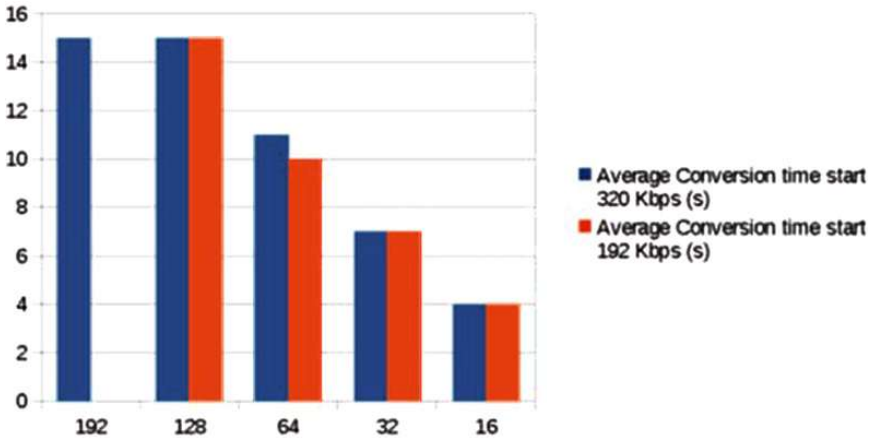

FIGURE 11.16

The average conversion time on a *EC2 c1.medium* platform. The bars at left and right correspond to the original file at the highest resolution (320 Kbps data rate) and next highest resolution (192 Kbps data rate), respectively.

the average conversion time is larger when the source file is smaller, as is the case when the target bit rates are 64, 32, and 16 Kbps.

Figure 11.16 shows the average conversion time for the experiments summarized in Tables 11.4 and 11.5.

The results of our measurements when the instance runs on the *EC2 c1.medium* platform show consistent and considerably lower conversion times; Figure 11.16 presents the average conversion time.

To understand the reasons for our results, we took a closer look at the two types of *AWS EC2* instances, "micro" and "medium," and their suitability for the adaptive data-streaming service. The *t1.micro* supports bursty applications, with a high average-to-peak ratio for CPU cycles, e.g., transaction-processing systems. The Amazon Elastic Block Store (EBS) provides block-level storage volumes; the "micro" instances are only EBS-backed.

The "medium" instances support compute-intensive application with a steady and relatively high demand for CPU cycles. Our application is compute-intensive; thus, there should be no surprise that our measurements for the *EC2 c1.medium* platform show consistent and considerably lower conversion times.

11.13 Cloud-based optimal FPGA synthesis

In this section we discuss another class of application that could benefit from cloud computing. In Chapter 4 we discussed cloud applications in computational science and engineering. The benchmarks presented in Section 4.9 compared the performance of several codes running on a cloud with runs on supercomputers; as expected, the results showed that a cloud is not an optimal environment for applications exhibiting fine- or medium-grained parallelism. Indeed, the communication latency is considerably larger on a cloud than on a supercomputer with a more expensive, custom interconnect. This means that we have to identify cloud applications that do not involve extensive communication or applications exhibiting coarse-grained parallelism.

A cloud is an ideal running environment for scientific applications that involve model development. In this case, multiple cloud instances could concurrently run slightly different models of the system. When the model is described by a set of parameters, the application can be based on the SPMD paradigm combined with an analysis phase when the results from the multiple instances are ranked based on a well-defined metric. In this case there is no communication during the first phase of the application, when partial results are produced and then written to the storage server. Then individual instances signal the completion and a new instance to carry out the analysis and display the results is started. A similar strategy can be used by engineering applications of mechanical, civil, electrical, electronic, or any other system design area. In this case, the multiple instances run concurrent design for different sets of parameters of the system.

A cloud application for optimal design of Field-Programmable Gate Arrays (FPGAs) is discussed next. As the name suggests, an FPGA is an integrated circuit designed to be configured, adapted, or programmed in the field to perform a well-defined function [311]. Such a circuit consists of *logic blocks* and *interconnects* that can be "programmed" to carry out logical and/or combinatorial functions (see Figure 11.17).

The first commercially viable FPGA, the XC2064, was produced in 1985 by Xilinx. Today FPGAs are used in many areas, including digital signal processing, CRNs, aerospace, medical imaging, computer vision, speech recognition, cryptography, and computer hardware emulation. FPGAs are less energy efficient and slower than application-specific integrated circuits (ASICs). The widespread use of FPGAs is due to their flexibility and the ability to reprogram them.

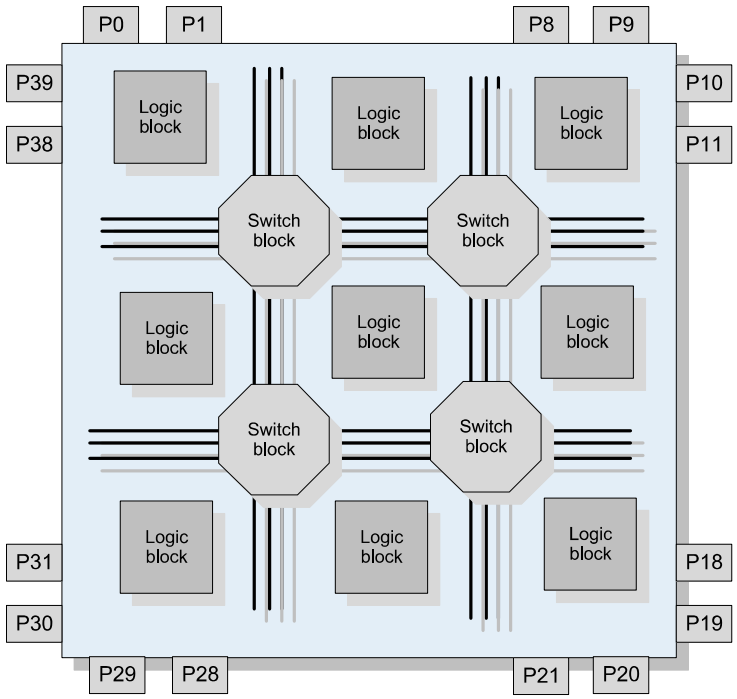

FIGURE 11.17

The structure of an FPGA with 30 pins, P0–P29; nine logic blocks; and four switch blocks.

Hardware description languages (HDLs) such as VHDL and Verilog are used to program FPGAs. HDLs are used to specify a register-transfer level (RTL) description of the circuit. Multiple stages are used to synthesize FPGAs.

A cloud-based system was designed to optimize the routing and placement of components. The basic structure of the tool is shown in Figure 11.18. The system uses the PlanAhead tool from Xilinx (see `www.xilinx.com`) to place system components and route chips on the FPGA logical fabric. The computations involved are fairly complex and take a considerable amount of time; for example, a fairly simple system consisting of a software core processor (Microblaze), a block random access memory (BRAM), and a couple of peripherals can take up to 40 minutes to synthesize on a powerful workstation. Running *N* design options in parallel on a cloud speeds up the optimization process by a factor close to N.

11.14 Exercises and problems

Problem 1. Establish an *AWS* account. Use the *AWS* Management Console to launch an *EC2* instance and connect to it.

Problem 2. Launch three *EC2* instances. The computations carried out by the three instances should consist of two phases, and the second phase should be started only after all instances

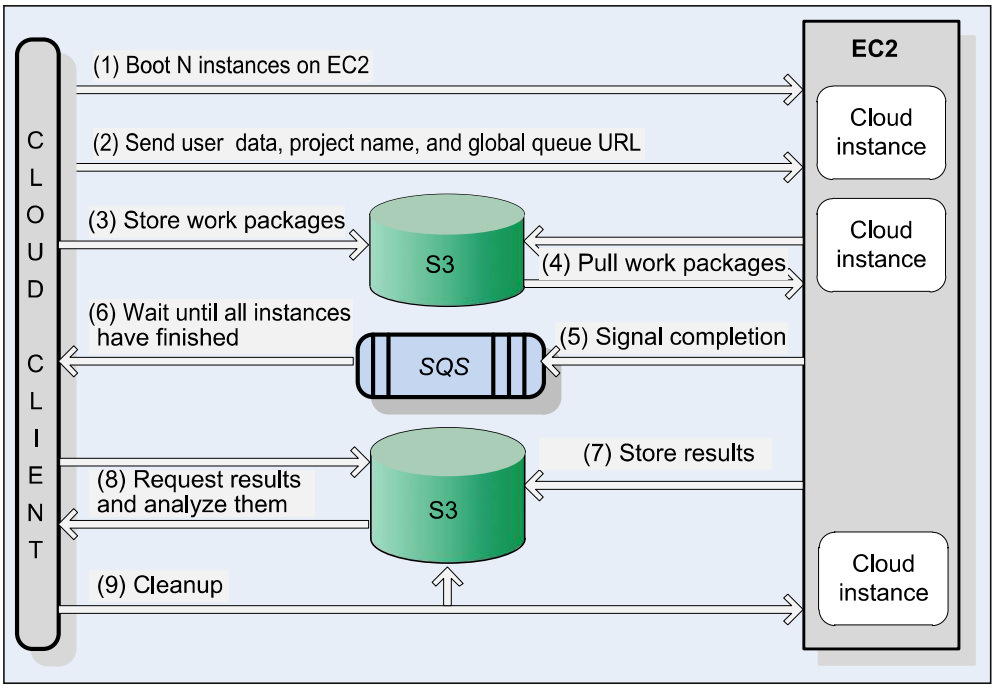

FIGURE 11.18

The architecture of a cloud-based system to optimize the routing and placement of components on an FPGA.

have finished the first stage. Design a protocol and use *Simple Queue Service (SQS)* to implement the barrier synchronization after the first phase.

Problem 3. Use the *Zookeeper* to implement the coordination model in Problem 2.

Problem 4. Use the *Simple Workflow Service (SWF)* to implement the coordination model in Problem 2. Compare the three methods.

Problem 5. Upload several (10 – 20) large image files to an *S3* bucket. Start an instance that retrieves the images from the *S3* bucket and compute the retrieval time. Use the *ElastiCache* service and compare the retrieval time for the two cases.

Problem 6. Numerical simulations are ideal applications for cloud computing. Output data analysis of a simulation experiment requires the computation of confidence intervals for the mean for the quantity of interest [210]. This implies that one must run multiple batches of simulation, compute the average value of the quantity of interest for each batch, and then calculate, say, 95% confidence intervals for the mean. Use the *CloudFormation* service to carry out a simulation using multiple cloud instances that store partial results in *S3* and then another instance that computes the confidence interval for the mean.

Problem 7. Run an application that takes advantage of the *Autoscaling* service.

Problem 8. Use the *Elastic Beanstalk* service to run an application and compare it with the case when the *Autoscaling* service was used.

Problem 9. Design a cloud service and a testing environment. Use the *Elastic Beanstalk* service to support automatic scaling up and down, and use the *Elastic Load Balancer* to distribute the incoming service request to different instances of the application.