Chapter 12
Designing with Distributed Objects

As businesses grow, they often set up operations over large geographic areas that may span multiple states or even countries and often find it desirable to process data at their point of origin or create results at the location where they are needed. As a consequence, businesses usually install multiple computer systems that are interconnected by communication links, and applications run across a network of computers rather than on a single machine. Such systems are called distributed systems.

Distributed processing offers a number of advantages. It is more economical and efficient to process data at the point of origin. Distributed systems make it easier for users to access and share resources. They also offer higher reliability and availability: failure of a single computer does not cripple the system as a whole. It is also more cost effective to add more computing power.

Distributed computing is not without its share of drawbacks. First, the software for implementing them is complex. Although a distributed system is made up of multiple computers, its design must somehow ensure that users, for the most part, are able to view it as a centralised system; it must coordinate actions between a number of possibly heterogeneous computer systems; if data is replicated, the copies must be made mutually consistent. Second, data access may be slow because information may have to be transferred across communication links. Third, securing the data is a challenge. As data is distributed over multiple systems and transported over communication links, care must be taken to guarantee that it is not lost, corrupted, or stolen.

As the final, major topic of the book, we address the process of designing and implementing a distributed, object-oriented application system. We present two approaches to building a such systems. The first mechanism uses *Java Remote Method Invocation (Java RMI)*, which is a piece of software, generally called middleware, that helps mask heterogeneity. The second approach uses the world-wide web itself to access data processed at remote sites.

B. Dathan and S. Ramnath, *Object-Oriented Analysis, Design and Implementation*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-24280-4_12

12.1 Client/Server Systems

Distributed systems can be classified into peer-to-peer systems and client-server systems. In the former, every computer system (or node) in the distributed system runs the same set of algorithms; they are all equals, in some sense. The latter, the client/server approach, is more popular in the commercial world. In client/server systems, there are two types of nodes: clients and servers. A client machine sends requests to one or more servers, which process the requests, and return the results to the client. Many applications can use this model and these days the software at many clients are web browsers.

In this chapter, we look at the implementation of object-oriented systems that use the client/server paradigm. We look at the architecture itself in Sect. 12.1.1.

12.1.1 Basic Architecture of Client/Server Systems

To keep matters simple, we assume that although the client/server systems we build may have multiple clients, they will have just one server. It is not difficult to extend the techniques to multiple servers, so this is not a serious restriction. Figure 12.1 shows a system with one server and three clients. Each client runs a program that provides a user interface, which may or not be a GUI. The server hosts an object-oriented system. Like any other client/server system, clients send requests to the server, these requests are processed by the object-oriented system at the server, and the results are returned. The results are then shown to end-users via the user interface at the clients.

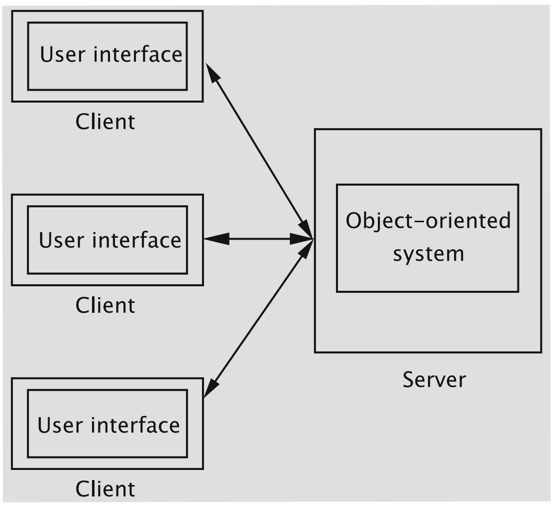


Fig. 12.1 Client/Server systems

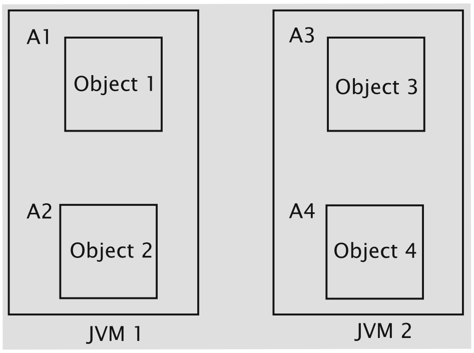


Fig. 12.2 Difficulty in accessing objects in a different JVM

There is a basic difficulty in accessing objects running in a different Java Virtual Machine (JVM). Let us consider two JVMs hosting objects as in Fig. 12.2. A single JVM has an address space part of which is allocated to objects living in it. For example, objects `object 1` and `object 2` are created in JVM 1 and are allocated at addresses A1 and A2 respectively. Similarly, objects `object 3` and `object 4` live in JVM 2 and are respectively allocated addresses A3 and A4. Code within `Object 2` can access fields and methods in `object 1` using address A1 (subject, of course, to access specifiers). However, addresses A3 and A4 that give the addresses of objects `object 3` and `object 4` in JVM 2 are meaningless within JVM 1. To see this, suppose A1 and A3 are equal. Then, accessing fields using address given by A3 from code within JVM 1 will end up accessing memory locations within `object 1`.

This difficulty can be handled in one of two ways:

1. By using object-oriented support software: The software solves the problem by the use of proxies that receive method calls on 'remote' objects, ship these calls, and then collect and return the results to the object that invoked the call. The client could have a custom-built piece of software that interacts with the server software. This approach is the basis of Java Remote Method Invocation and is covered in Sect. 12.2.
2. By avoiding direct use of remote objects by using the Hyper Text Transfer Protocol (HTTP). The system sends requests and collects responses via encoded text messages. The object(s) to be used to accomplish the task, the parameters, etc., are all transmitted via these messages. This approach has the client employ an Internet browser, which is, of course, a piece of general purpose software for accessing documents on the world-wide web. In this case, the client software is ignorant of the application structure and communicates to the server via text messages that include HTML code and data. This is the technique used for hosting a system on the Web; it is definitely more popular and we cover it in Sect. 12.3.

12.2 Java Remote Method Invocation

The goal of Java RMI is to support the building of Client/Server systems where the server hosts an object-oriented system that the client can access programmatically. The objects at the server maintained for access by the client are termed **remote objects**. A client accesses a remote object by getting what is called a **remote reference** to the remote object. After that the client may invoke methods of the object.

The basic idea behind RMI is to employ the **proxy** design pattern. This pattern is used when it is inefficient or inconvenient (even impossible, perhaps) to use the actual object. (Please refer to Fig. 12.3 for a description of the proxy pattern.) In the current context, the object is only available at a remote site. If the same object is to be available at multiple client sites, one option is to download a copy of the object to all client sites, but such replication of objects introduces synchronisation issues when the object is to be updated. Instead, the proxy pattern creates a *proxy* object

Using a proxy

Where do we employ this? This pattern is used when it is inefficient or inconvenient (even impossible, perhaps) to create/use an actual object. Perhaps creating an object may be too time consuming, in which case the use of this pattern lets postponement of this creation until the actual object is needed.

Examples of its use include distributed systems in which we need to access objects remotely. Another example would be opening a document that embeds graphical objects that are too time consuming to create.

How have we solved it? In the distributed systems case, the use of a proxy allows us to access the remote object by reference. That way, updates from multiple clients are made directly to the object. In contrast, we may choose to copy the object to the point of use; however, replication always introduces consistency issues that need the employment of expensive protocols.

In the example of documents referring to graphical objects, the proxy creates the actual image object only when so asked by the document object itself. After the image is created, all requests sent to the proxy are directed to the actual image itself.

In both examples, notice that the client always maintains a reference to the proxy, which delegates the responsibility of carrying out the operations to the actual object.

How have we employed it? Java RMI employs proxies to stand in for remote objects. All operations exported to remote sites (remote operations) are implemented by the proxy. Proxies are termed *stubs* in Java RMI. These stubs are created by the RMI compiler.

Fig. 12.3 Using a proxy

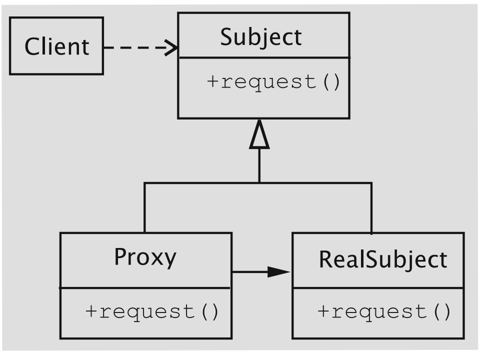


Fig. 12.4 Client/Server systems

at each client site that accesses the remote object. The proxy object implements all of the remote object's operations that the remote object wants to be available to the client. The set up is shown in Fig. 12.4. When the client calls a remote method, the corresponding method of the proxy object is invoked. The proxy object then assembles a message that contains the remote object's identity, method name, and parameters. This assembly is called **marshalling**. In this process, the method call must be represented with enough information so that the remote site knows the object to be used, the method to be invoked, and the parameters to be supplied. When the message is received by it, the server performs **demarshalling**, whereby the process is reversed. The actual call on the remote method of the remote object is made, and any return value is returned to the client via a message shipped from the server to the proxy object.

The system maintains a separate proxy for remote object. When an object asks for a reference to a remote object, it is handed a reference to the object's proxy instead. Setting up remote object access in RMI, even for a small application, involves a number of steps. In fact, as we shall see, once we learn to set up a simple application system, we will have learned the tools and techniques for creating almost any client/server system using this technology. As we discuss each major concept, we will illustrate it using a running example.

Setting up a remote object system is accomplished by the following steps:

1. Define the functionality that must be made available to clients. This is accomplished by creating **remote interfaces**.
2. Implement the remote interfaces via **remote classes**.
3. Create a server that serves the remote objects.
4. Set up the client.

These are elaborated in the following sections.

12.2.1 Remote Interfaces

The first step in implementing a remote object system is to define the system functionality that will be exported to clients, which implies the creation of a Java interface. In the case of RMI, the functionality exported of a remote object is defined via what is called a *remote interface*. A remote interface is a Java interface that extends the interface `java.rmi.Remote`, which contains no methods and hence simply serves as a marker. Clients are restricted to accessing methods defined in the remote interface. We call such method calls **remote method invocations.**

Remote method invocations can fail due to a number of reasons: the remote object may have crashed, the server may have failed, or the communication link between the client and the server may not be operational, etc. Java RMI encapsulates such failures in the form of an object of type `java.rmi.RemoteException`; as a result, all remote methods must be declared to throw this exception.

In summary, a remote interface must extend `java.rmi.Remote` and every method in it must declare to throw `java.rmi.RemoteException`. These concepts are shown in the following example.

```
import java.rmi.*;
public interface BookInterface extends Remote {
  public String getAuthor() throws RemoteException;
  public String getTitle() throws RemoteException;
  public String getId() throws RemoteException;
}
```

Remote objects implement remote interfaces. They may implement more methods, but clients are restricted to accessing methods declared in the remote interfaces.

12.2.2 Implementing a Remote Interface

After the remote interfaces are defined, the next step is to implement them via *remote classes*. Parameters to and return values from a remote method may be of primitive type, of remote type, or of a local type. All arguments to a remote object and all return values from a remote object must be serializable. Thus, in addition to the requirement that remote classes implement remote interfaces, we require that they also implement the `java.io.Serializable` interface. Parameters of non-remote types are passed by copy; they are serialized using the object serialization mechanism, so they too must implement the `Serializable` interface.

Intuitively, remote objects must somehow be capable of being transmitted over networks. A convenient way to accomplish this is to extend the class `java.rmi.server.UnicastRemoteObject`.

Thus, the implementation of `BookInterface` is as below.

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
```

```
public class Book extends UnicastRemoteObject implements
             BookInterface, Serializable {
  private String title;
  private String author;
  private String id;
  public Book(String title1, String author1, String id1)
         throws RemoteException {
    title = title1;
    author = author1;
    id = id1;
  }
  public String getAuthor()  throws RemoteException {
    return author;
  }
  public String getTitle()  throws RemoteException {
    return title;
  }
  public String getId()  throws RemoteException {
    return id;
  }
}
```

Since it is a remote class, `Book` must be compiled using the RMI compiler by invoking the command `rmic` as below.

```
rmic Book
```

The compiler produces a file named `Book_Stub.class`, which acts as a proxy for calls to the methods of `BookInterface`. When the constructor for the remote class (`Book` in the above case) is invoked, the constructor for `UnicastRemoteObject` *exports* the remote object. When an exported remote object is passed as a parameter or returned from a remote method call, the stub for that remote object is passed instead of the object itself. The stub itself contains a reference to the serialized object and implements all of the remote interfaces that the remote object implements. All calls to the remote interface go through the stub to the remote object.

Remote objects are thus passed by reference. This is depicted in Fig. 12.5, where we have a single remote object that is being accessed from two clients. Both clients maintain a reference to a stub object that points to the remote object that has a field named `a`. Suppose now that Client 1 invokes the method `setA` with parameter 5. As we have seen earlier, the call goes through the stub to the remote object and gets executed changing the field `a` to 5. The scheme has the consequence that any changes made to the state of the object by remote method invocations are reflected in the original remote object. If the second client now invokes the method `getA`, the updated value `5` is returned to it.

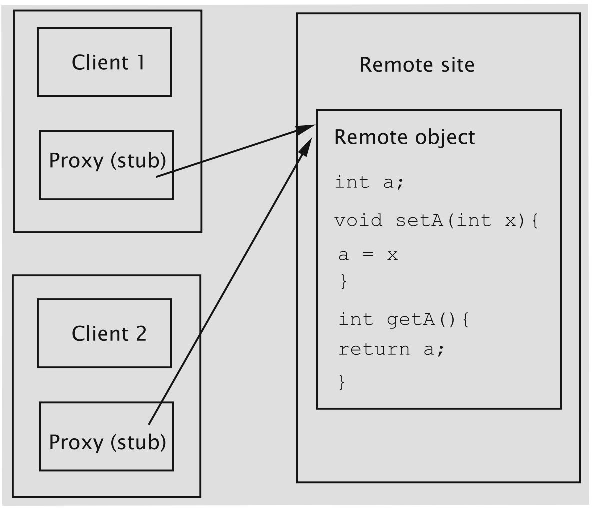


Fig. 12.5 Passing of remote objects as references

In contrast, parameters or return values that are *not* remote objects are passed by value. Thus, any changes to the object's state by the client are reflected only in the client's copy, not in the server's instance. Similarly, if the server updates its instance, the changes are not reflected in the client's copy.

12.2.3 Creating the Server

Before a remote object can be accessed, it must be instantiated and stored in an object registry, so that clients can obtain its reference. Such a registry is provided in the form of the class `java.rmi.Naming`. The method `bind` is used to register an object and has the following signature:

```
public static void bind(String nameInURL, Remote object)
      throws AlreadyBoundException, MalformedURLException,
      RemoteException
```

The first argument takes the form `//host:port/name` and is the URL of the object to be registered; `host` refers to the machine (remote or local) where the registry is located, `port` is the port number on which the registry accepts calls, and `name` is a simple string for distinguishing the object from the other objects in the registry. Both `host` and `port` may be omitted in which case they default to the local host and the port number of 1099, respectively.

The process of creating and binding the name is given below.

```
try {
  <interface-name> object = new <class-name>(parameters);
  Naming.rebind("//localhost:1099/SomeName", object);
} catch (Exception e) {
  System.out.println("Exception " + e);
}
```

The complete code for activating and storing the `Book` object is shown below.

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class BookServer {
  public static void main(String[] s) {
    String name = "//localhost:1099/" + s[0];
    try {
      BookInterface book = new Book("t1", "a1", "id1");
      Naming.rebind(name, book);
    } catch (Exception e) {
      System.out.println("Exception " + e);
    }
  }
}
```

In the above code, we assume that when the server code is executed, it is provided with the name that should be associated with the `Book` object.

12.2.4 The Client

A client may get a reference to the remote object it wants to access in one of two ways:

1. It can obtain a reference from the `Naming` class using the method `lookup`.
2. It can get a reference as a return value from another method call.

Let us see how the first of these approaches can be accomplished. In the following we assume that an object of type `SomeInterface` has been entered into the local registry under the name `SomeName`.

```
SomeInterface object = (SomeInterface) Naming.lookup
                          ("//localhost:1099/SomeName");
```

After the above step, the client can invoke remote methods on the object. In the following code, the getters of the `BookInterface` object are called and displayed.

```
import java.util.*;
import java.rmi.*;
import java.net.*;
import java.text.*;
import java.io.*;
public class BookUser {
  public static void main(String[] s) {
    try {
      String name = "//localhost/" + s[0];
      BookInterface book = (BookInterface) Naming.lookup(name);
      System.out.println(book.getTitle() + " " + book.getAuthor()
                                  + " " + book.getId());
    } catch (Exception e) {
      System.out.println("Book RMI exception: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

Just as in the case of the server, the client needs to know the name that is bound to the object, so it must be started with that name as the parameter.

12.2.5 Setting up the System

To run the system, create two directories, say `server` and `client`, and copy the files `BookInterface.java`, `Book.java`, and `BookServer.java` into `server` and the file `BookUser.java` into `client`. Then compile the three Java files in `server` and then invoke the command

```
rmic Book
```

while in the `server` directory. This command creates the stub file `Book_Stub.class`. Copy the client program into `client` and compile it.

Run RMI registry and the server program using the following commands (on Windows).

```
start rmiregistry
java -Djava.rmi.server.codebase=file:C:\Server\BookServer
BookServer MyBook
```

The first command starts the registry and the second causes the `Book` instance to be created and registered with the name `MyBook`.

Finally, run the client as below from the `client` directory.

```
java -Djava.rmi.server.codebase=file:C:\Client\BookUser
BookUser MyBook
```

The client code starts, looks up the object with the name `MyBook`, calls the object's getter methods, and displays the values.

12.3 Implementing an Object-Oriented System on the Web

Without doubt, the world-wide web is the most popular medium for hosting distributed applications. Increasingly, people are using the web to book airline tickets, purchase a host of consumer goods, make hotel reservations, and so on. The browser acts as a general purpose client that can interact with any application that talks to it using the Hyper Text Transfer Protocol (HTTP).

One major characteristic of a web-based application system is that the client (the browser), being a general-purpose program, typically does no application-related computation at all. Of course, it is possible to ship a Java applet with a web page and have the applet do some computation, but this is not hugely popular. All business logic and data processing take place at the server. Typically, the browser receives web pages from the server in HTML and displays the contents according to the format, a number of tags and values for the tags, specified in it. In this sense, the browser simply acts as a 'dumb' program displaying whatever it gets from the application and transmitting user data from the client site to the server.

The HTML program shipped from a server to a client often needs to be customised: the code has to suit the context. For example, when we make a reservation on a flight, we expect the system to display the details of the flight on which we made the reservation. This requires that HTML code for the screen be dynamically constructed. This is done by code at the server.

For server-side processing, there are competing technologies such as Java Server Pages and Java Servlets, Active Server Pages (ASP), and PHP. In this book we study Java Servlets.

12.3.1 HTML and Java Servlets

As we have stated earlier, any system that ultimately displays web pages via a browser has to create HTML code. HTML code displays text, graphics such as images, links that users can click to move to other web pages, and *forms* for the user to enter data. We will now describe the essential code for doing these.

An HTML program can be thought of as containing a header, a body, and a trailer. The header contains code like the following:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta content="text/html; charset=ISO-8859-1"
   http-equiv="content-type">
  <title>A Web Page</title>
</head>
```

The first four lines are usually written as given for any HTML file. We do not elaborate on these, but observe words such as `html` and `head` that are enclosed between angled brackets (< and >). They are called **tags**. HTML tags usually occur in pairs: start tag that begins an entry and end tag that signals the entry's end. For example, the tag `<head>` begins the header and is ended by `</head>`. The text between the start and end tags is the element content.

In the fifth line we see the tag `title`, which defines the string that is displayed in the title bar. The idea is that the string `A Web Page` will be displayed in the title bar of the browser when this page is displayed.

As a sample body, let us consider the following.

```
<body>
<h1>
    <span style="color: rgb(0, 0, 255);">
    <span style="font-family: lucida bright;">
    <span style="font-style: italic;">
    <span style="font-weight: bold;">
       An Application
    </span>
    </span>
    </span>
    </span>
</h1>
</body>
```

The body contains code that determines what gets displayed in the browser's window. Some tags may have attributes, which provide additional information. For example, see the line

```
    <span style="color: rgb(0, 0, 255);">
```

where the tag `span` has its attribute `style` modified, so that the text will be in blue colour: the reader may have guessed that `rgb` stands for the amount of red, green, and blue in the color, whose the values can range from 0 to 255. As the examples suggest, attributes always come in name/value pairs of the form `name="value"`. They are always specified in the start tag of an HTML element.

The body contains code to display the string `An Application` in the font Lucida bright, bolded, italicised, and in blue color.
The last line of the file is

```
</html>
```

Obviously, it ends the HTML file.

Entering and Processing Data

The reader is probably familiar with web pages that allow the user to enter information that the system processes. For example, a search engine provides a field in which we type in some search terms. When an accompanying button is clicked, the system transfers control to the search engine that displays results of the search.

This is accomplished by using what is called a **form** tag in HTML. The complete code that allows us to enter some piece of text in the web page is given below.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
 <meta content="text/html; charset=ISO-8859-1"
 http-equiv="content-type">
 <title>Sample Form</title>
</head>
<body>
  <form action="/servlet/apackage.ProcessInput" method="post">
    <table>
       <tr>
           <td align="right">Enter Data:</td>
           <td><input type="text" name="userInput"></td>
       </tr>
       <tr>
           <td><input type="submit" value="Process"></td>
       </tr>
    </table>
  </form>
</body>
</html>
```

Let us get a general understanding of the above piece of code. Consider the code that begins with the line

```
  <form action="/servlet/apackage.ProcessInput" method="post">
```

The tag `form` begins the specification of a set of elements that allow the user to enter information. The `action` attribute specifies that the information entered by the user is to be processed by a Java class called `ProcessInput.class`, which resides in the package `apackage`.

There are two primary ways in which form data is encoded by the browser: one is GET and the other is POST. GET means that form data is to be encoded into a URL while POST makes data appear within the message itself. See Fig. 12.6 for the considerations in deciding which of these methods should be used.

The tag `<table>` begins the creation of a table. Each row of the table is described using the tag `<tr>`, and the tag `<td>` defines a cell in the table. The line

```
<td><input type="text" name="userInput"></td>
```

GET or POST?

While considering the question of which of the two methods, GET or POST, should be employed to transmit form data, it is helpful to remember that GET inserts the data in the URL itself whereas POST includes the data as part of the message. As a consequence, the URLs created for the POST and GET methods differ in that the latter completely identifies the server resource. This implies that the resource from the URL of the GET method can be used from other web pages to access the same resource, a capability that is not possible with the URL of POST.

A section in the HTTP/1.1 specifications talks about a kind of client/server interactions called **safe interactions**. In a safe interaction, users are not responsible for the result of the interaction, and GET is the appropriate method to use in such situations. To understand the concept of safe interactions, consider a web page (call it page 1) that asks the user to agree to some conditions by checking a box before allowing him/her to download a piece of software from a second page (say, page 2). Suppose that the form data, which includes the checkbox, from page 1 is transmitted using GET. Clearly, the URL completely identifies page 2. This URL can then be used to provide a link, called a deep link, to Page 2 from an unrelated web page (say, page 3), and any use of this link from page 3 is an unsecure way of accessing the resource.

It should be noted, however, that trying to hide the resource location is not a foolproof mechanism: one could look at the source file of the web page to craft a link to the resource.

One of the HTTP usage recommendations is that the GET method should be used only when the form processing is **idempotent**, that is, the result is the same whether the form is processed once or multiple times. This definition, however, should not be taken too literally. Generally speaking, if resubmitting a form does not change the application data stored at the server (even if it changes other entities such as log files), it is appropriate to use GET. In other circumstances, the POST method should be used to transmit form data.

Generally speaking, results from the GET method are cached, but data obtained from POST are not. As a consequence, GET method may execute faster than POST.

Fig. 12.6 Get and post: a brief comparison

thus creates a cell, which is actually an input field where the user can enter data. This is indicated by the `<input>` tag two attributes of which, `type` and `name`, are modified in this example. One attribute is `type`, which specifies what the type of input is: here we have `"text"`, which means plain text. (Some other possibilities such as `"password"`, which makes the entry unreadable on the screen, will be covered later in the chapter.) The second attribute, `name`, must be given a unique value: the value names the input element, somewhat like an identifier.

Next, look at the line

```
<td><input type="submit" value="Process"></td>
```

This line creates a button of type `"submit"`, which when clicked causes the form data to be sent to the server. The button has the label `Process`.

The server-side code `ProcessInput` is an example of a **servlet,** which uses the request-response paradigm. Servlets can process data sent using the HTTP protocol via a form. They can handle multiple requests concurrently. We create a servlet by extending the class `HttpServlet` as below.

```
public class ProcessInput extends HttpServlet {
```

Since we transmitted form data using the POST method, we need to override a method called `doPost`. This method has two parameters, `request` and `response` that respectively encapsulate the data sent by the client and the response to the client.

The header of the `doPost` method is given below.

```
public void doPost(HttpServletRequest request,
                  HttpServletResponse response) throws
                  IOException, ServletException {
```

Data sent by the client through the form is retrieved using the `request` object as below.

```
String input = request.getParameter("userInput");
```

Note that `userInput` corresponds to the name of the field in the form.

After the data is captured and processed, the servlet creates an HTML page using the response object as below.

```
response.setContentType("text/html");
response.getWriter().println("<!DOCTYPE html PUBLIC \"-//W3C//DTD "+
                             "HTML 4.01 Transitional//EN\">");
```

The first line states that the data is HTML and the second line begins the HTML code. The complete code for the servlet is given below.

```
package apackage;
import javax.servlet.*;
import javax.servlet.http.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
public class ProcessInput extends HttpServlet {
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response) throws IOException,
                     ServletException {
    String input = request.getParameter("userInput");
    response.setContentType("text/html");
    response.getWriter().println("<!DOCTYPE html PUBLIC \"-//W3C//DTD " +
                                 "HTML 4.01 Transitional//EN\">");
    response.getWriter().println("<html>");
    response.getWriter().println("<head>");
    response.getWriter().println("<meta content=\"text/html;" +
                                 " charset=ISO-8859-1\"" +
                                 "http-equiv=\"content-type\">");
    response.getWriter().println("<title>Response to Input</title>");
    response.getWriter().println("</head>");
    response.getWriter().println("<body>");
    response.getWriter().println("You entered " + input);
    response.getWriter().println("</body>");
    response.getWriter().println("</html>");
  }
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response) throws IOException,
                    ServletException {
    doPost(request, response);
  }
}
```

Although we do not use the GET method, we have overridden it, so that in case the form is changed to use the GET method, the system will continue to work.

The architecture for serving web pages is depicted in Fig. 12.7. Assume that an HTML page is displayed on the client's browser. The page includes, among other things, a form that allows the user to enter some data. The client makes some entries in the form's fields and submits them, say, by clicking a button. The data in the form is then transmitted to the server and given to a Java servlet, which processes the data and generates HTML code that is then transmitted to the client's browser, which displays the page.

12.3.2 Deploying the Library System on the World-Wide Web

We now undertake the task of designing and developing a web-based version of the library system. Of course, we cannot do everything exactly as in a real library: in particular, we do not have machines that scan bar codes on books, but we will do as close a job as possible as a real system.

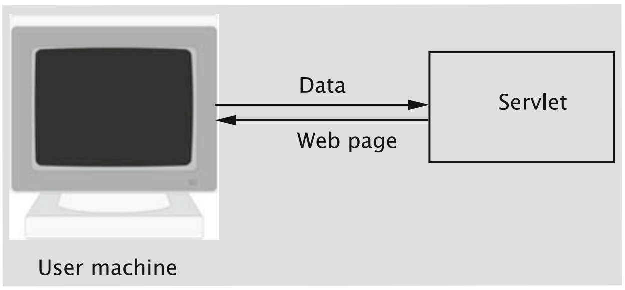


Fig. 12.7 How servlets and HTML cooperate to serve web pages

Developing User Requirements

As in any system, the first task is to determine the system requirements. We will, as has been the case throughout the book, restrict the functionality so that the system's size is manageable.

1. The user must be able to type in a URL in the browser and connect to the library system.
2. Users are classified into two categories: *superusers* and *ordinary members*. Superusers are essentially designated library employees, and ordinary members are the general public who borrow library books. The major difference between the two groups of users is that superusers can execute any command when logged in from a terminal in the library, whereas ordinary members cannot access some 'privileged commands'. In particular, the division is as follows:
   (a) Only superusers can issue the following commands: add a member, add a book, return a book, remove a book, process holds, save data to disk, and retrieve data from disk.
   (b) Ordinary members and superusers may invoke the following commands: issue and renew books, place and remove holds, and print transactions.
   (c) Every user eventually issues the exit command to terminate his/her session.
3. Some commands can be issued from the library only. These include all of the commands that only the superuser has access to and the command to issue books.
4. A superuser cannot issue any commands from outside of the library. They can log in, but the only command choice will be to exit the system.
5. Superusers have special user ids and corresponding password. For regular members, their library member id will be their user id and their phone number will be the password.

Interface requirements It turns out that due to the nature of the graphical user interface, an arbitrarily large number of sequences of interactions are possible between the user and the interface. Employing the use-case model alone to determine the requirements would necessitate the use of too many conditionals, and the resulting sequence diagrams are not easily understood. Suppose that for their convenience, users be able to abandon most operations in the middle; for example, the user may decide to place a hold on a book, but when the screen to enter the book id and duration of the hold pops up, the user may change her mind and decide not to place a hold. For a second example, a book may be self issued or the member may ask a library staff member to check out the book. In the latter case, the member id needs to be input, whereas in the former case, that information is already available to the system.

We thus depict the requirements mostly through state transition diagrams. (A little later, we will depict the flow using a sequence diagram as well.) However, a single transition diagram is too large and unwieldy. Therefore, we split the state transition diagram into a number of smaller ones.

Logging in and the Initial Menu

In Fig. 12.8, we show the process of logging in to the system. When the user types in the URL to access the library system, the log in screen that asks for the user id and password is displayed on the browser. If a valid combination is typed in, an appropriate menu is displayed. What is in the menu depends on whether the user is an ordinary member or a superuser and whether the terminal is in the library or is outside.

1. The Issue Book command is available only if the user logs in from a terminal in the library.
2. Commands to place a hold, remove a hold, print transactions, and renew books are available to members of the library (not superusers) from anywhere.
3. Certain commands are available only to superusers who log in from a library terminal: these are for returning or deleting books, adding members and books, processing holds, and saving data to and retrieving data from disk.

A superuser has to be logged in from a terminal in the library, or the menu will simply contain the command to exit the system. It may seem somewhat strange that

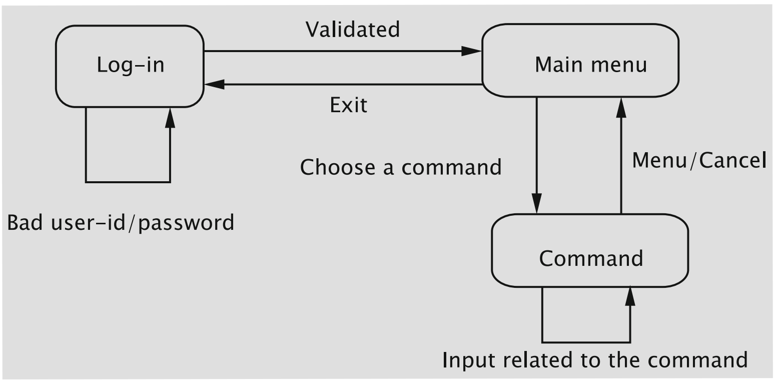


Fig. 12.8 State transition diagram for logging in

a superuser has no access to the library system from outside of the library whereas an ordinary member has several commands at her disposal. But note that it makes not much sense to allow a superuser to issue most of the commands from outside the library: a superuser cannot deal with a library member from the outside, so commands to add members, issue, return, and renew books, place and remove holds and print transactions are not applicable. Also, a superuser cannot be reasonably expected to add or remove books from the outside. One could make a case that a superuser be allowed to process holds and save and retrieve data from the outside, but it is hard to see why a superuser would work from the outside just for issuing these commands.

When the user types in the URL for the library, the system presents a log-in screen for entering the user id and password. If the user types in a bad user id/password combination, the system presents the log in screen again with an error message.

On successful validation, the system displays a menu that contains clickable options. The Command State in Fig. 12.8 denotes the general flow of a command. When a certain command is chosen, we enter a state that represents the command. How the transitions take place within a command obviously depends on what the command is. All screens allow an option to cancel and go back to the main menu. If this option is chosen, the system goes on to display the main menu awaiting the next command.

When the exit command is chosen, the system logs the user out and presents the log in screen again.

Add Book

The flow is shown in Fig. 12.9. When the command to add a book is chosen, the system constructs the initial screen to add a book, which should contain three fields for entering the title, author, and id of the book, and then display it and enter the Add Book state. By clicking on a button, it should be possible for the user to submit these values to system. The system must then call the appropriate method in the `Library` class to create a `Book` object and enter it into the catalog. The result of the operation is displayed in the Command Completed state.

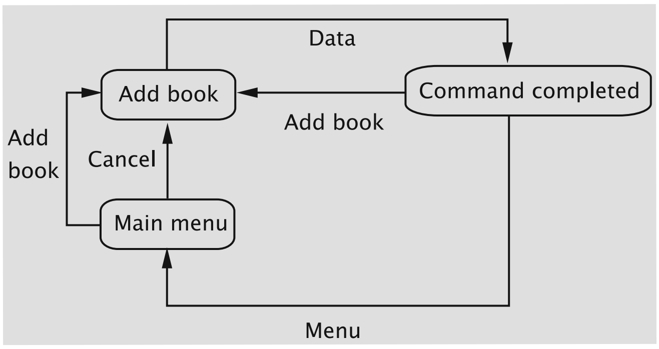


Fig. 12.9 State transition diagram for add book

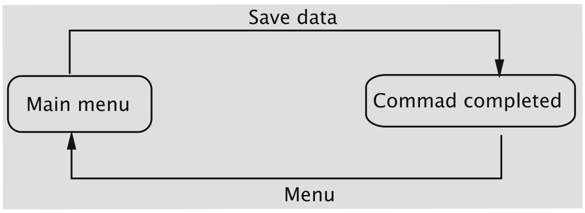


Fig. 12.10 State transition diagram for saving data

From the Command Completed state, the system must allow the user to add another book or go back to the menu. In the Add Book state, the user has the option to cancel the operation and go back to the main menu.

Add Member, Return Book, Remove Book

The requirements are similar to the ones for adding books. We need to accept some input (member details or book id) from the user, access the `Library` object to invoke one of its methods, and display the result. So we do not describe them here nor do we give the corresponding state transition diagrams.

Save Data

When the data is to be written to disk, no further input is required from the user. The system should carry out the task and print a message about the outcome. The state transition diagram is given in Fig. 12.10.

Retrieve Data

The requirements are similar to those for saving data.

Issue Book

This is one of the more complicated commands. As shown in the state transition diagram in Fig. 12.11, a book may be checked out in two different ways: First, a member is allowed to check it out himself/herself. Second, he/she may give the book to a library staff member, who checks out the book for the member. In the first case, the system already has the user's member id, so that should not be asked again. In the second case, the library staff member needs to input the member id to the system followed by the book id.

After receiving a book id, the system must attempt to check out the book. Whether the operation is successful or not, the system enters the Book Id Processed state.

A second reason for the complexity arises from the fact that any number of books may be checked out. Thus, after each book is checked out, the system must ask if more books need to be issued or not. The system must either go to the Get Book Id state for one more book id or to the Main Menu state.

As usual, it should be possible to cancel the operation at any time.

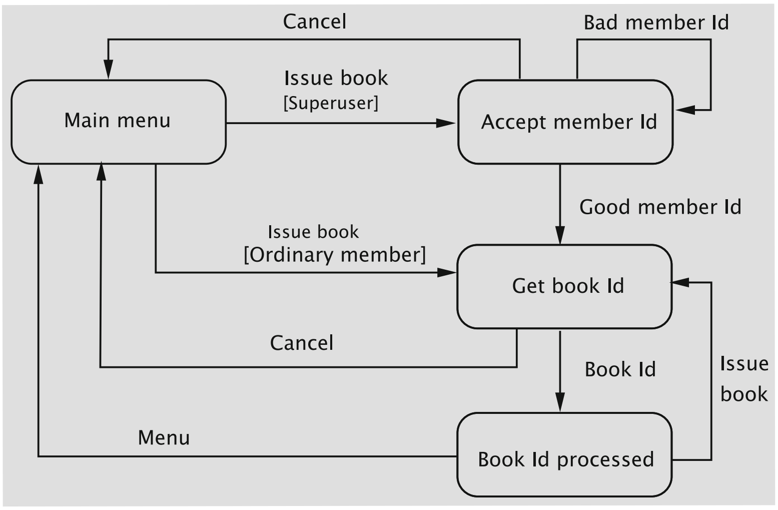


Fig. 12.11 State transition diagram for issuing books

Place Hold, Remove Hold, Print Transactions

The requirements for these are similar to those for issuing a book, so we omit their description.

Renew Books

The system must list the title and due date of all the books loaned to the member. For each book, the system must also present a choice to the user to renew the book. After making the choices, the member clicks a button to send any renew requests to the system. For every book renewal request, the system must display the title, the due date (possibly changed because of renewal), and a message that indicates whether the renewal request was honoured. After viewing the results, the member uses a link on the page to navigate to the main menu.

The state transition diagram is given in Fig. 12.12.

Design and Implementation

To deploy the system on the web, we need the following:

1. Classes associated with the library, which we developed in Chap. 7; you will recall that this includes classes such as `Library`, `Member`, `Book`, `Catalog`, and so on.
2. Permanent data (created by the save command) that stores information about the members, books, who borrowed what, holds, etc.
3. HTML files that support a GUI for displaying information on a browser and collecting data entered by the user. For example, when a book is to be returned, a screen that asks for the book id should pop up on the browser. This screen will

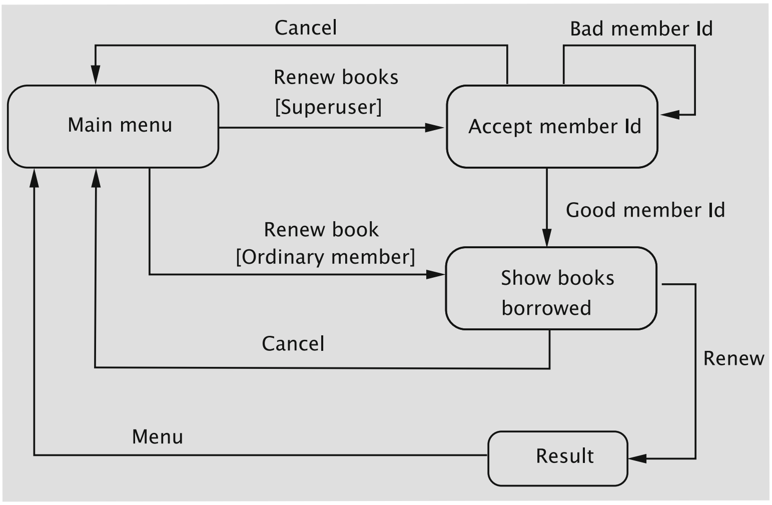


Fig. 12.12 State transition diagram for renewing books

have a prompt to enter the book id, a space for typing in the same, and a button to submit the data to the system.

4. A set of files that interface between the GUI ((3) above) and the objects that actually do the processing ((1) above). Servlets will be used to accomplish this task.

Structuring the files HTML code for delivery to the browser can be generated in one of two ways:

1. Embed the HTML code in the servlets. This has the disadvantage of making the servlets hard to read, but more dynamic code can be produced.
2. Read the HTML files from disk as a string and send the string to the browser. This is less flexible because the code remains static.

We attempt to combine the two approaches so as to utilise the advantages of both approaches without sacrificing either flexibility or cleanliness of code. Almost all HTML code is generated by reading files from disk; where needed, simple changes are applied to these files, so the desired functionality is achieved.

Having made the decision that most of the HTML code will be stored in files, the next question is how to set them up.

1. Create a separate HTML file for every type of page that needs to be displayed. For example, create a file for entering the id of the book to be returned, a second file for displaying the result of returning the book, a third file for inputting the id of the book to be removed, a fourth one for displaying the result of removing the book, etc.

2. Exploit the commonalities between the commands and create a number of HTML code fragments, a subset of which can be assembled to form an HTML file suitable for a specific context.

The first option has the advantage of simplicity. However, the reader can probably guess that the number of HTML files could be a problem. A rough calculation shows that at least 28 files are needed even without considering the intricacies associated with some of the commands such as Issue Book and Renew Book.

Although the second option is more involved because of the need to assemble a big file from several fragments, we find that it presents some advantages over the first. First, it reduces the number of files somewhat. More importantly, however, there is a great deal of duplication in the files in the first approach; duplication brings with it the problem of inconsistency. For example, to change the way the library's name is displayed in the screens, every one of the HTML files will need to be updated!

We thus opt for the second choice.

Examples of HTML file fragments To show how this approach works in practice, consider the two commands, one for returning and the other for removing books. In both, the user must be presented with a web page that asks him/her to enter a book id. We have just one file that displays this page. However, the servlet that needs to be invoked will change depending on the context. Therefore, we code the servlet name as below.

```
<form action="GOTO_WITH_BOOKID" method="post">
```

By simply changing the string `GOTO_WITH_BOOKID` in the servlet, we can use the same HTML file in multiple situations.

A similar approach is taken for accepting member ids.

For every web page, the header should display a title that depends on the context. We maintain just one file for the header. This file has a string `TITLE` that stands for the title of the web page. Depending on which page is being displayed, `TITLE` is replaced by an appropriate string, which gets displayed in the title bar.

When a command is completed, we need to display a web page. For most commands, the data to be displayed is small enough that it can be thought of as a simple string. We, therefore, employ just one file, `commandCompleted.html`, to carry out this task. This file is adapted, however, in two different ways.

1. The result to be displayed will vary on the command as well as whether the operation was successful. To take care of this, the file has a string called `RESULT`.

   ```
   <h3> RESULT <br></h3>
   ```

   This may be replaced by strings such as `Book not found` and `Member added`. Once the file is read into a string, the `RESULT` string is replaced by the appropriate result of executing the command. The following pseudocode gives the idea.

```
String result;
Member member;
String htmlFile = getFile("commandCompleted.html"):
if ((member = library.addMember(name, address, phone)) == null) {
  htmlFile = htmlFile.replace("TITLE", "Member not Added");
  result = "Member could not be added";
} else {
  htmlFile = htmlFile.replace("TITLE", "Member Added");
  result = member.getName() + " ID: " + member.getId() + " added";
}
htmlFile = htmlFile.replace("RESULT", result);
```

2. To reduce the number of mouse clicks, the user may be given the option to repeat the command whose result is displayed by the `commandCompleted.html` file. For example, after completing the `Add Book` command, we need to give an option to issue the command once again so that the user can add another book. Since the code where control should go to depends on the command that was just executed, some adaptation is in order. This is facilitated by having the line

   ```
   <a href="REPLACE_JS">REPLACE_COMMAND</a><br>
   ```

   in the HTML file.
   In the case of `Add Member`, we substitute `REPLACE_COMMAND` by `Add Book`, which provides a link that the user can click, and `REPLACE_JS` by `addmemberinitialization`, which locates the Java class that is given the control when the link is clicked.

   ```
   htmlFile = htmlFile.replace("REPLACE_J,S", "addmemberinitialization");
   htmlFile = htmlFile.replace("REPLACE_COMMAND", "Add Member");
   ```

How to remember a user Servlets typically deal with multiple users. When a servlet receives data from a browser, it must somehow figure out which user sent the message, what the user's privileges are, etc. Each request from the browser to the server starts a new connection, and once the request is served, the connection is torn down. However, typical web transactions involve multiple request–response pairs. This makes the process of remembering the user associated with a connection somewhat difficult without extra support from the system.

The system provides the necessary support by means of what are known as **sessions**, which are of type `HttpSession`. When it receives a request from a browser, the servlet may call the method `getSession()` on the `HttpServletRequest` object to create a **session object**, or if a session is already associated with the request, to get a reference to it. To check if a session is associated with the request and to optionally create one, a variant of this method `getSession(boolean create)` may be used. If the value `false` is passed to this method and the request has no valid `HttpSession`, this method returns `null`.

When a user logs in, the system creates a session object as below.

```
HttpSession session = request.getSession();
```

When the user logs out, the session is removed as below.

```
session.invalidate();
```

Requests other than log in requires the user to be logged in. The following code evaluates to true if the user does not have a session: that is, the user has not logged in.

```
request.getSession(false) == null
```

A session object can be used to store information about the session. In the library system, we would like to store the user id, the type of terminal from which the user has logged in, and some additional information related to the user. The methods for this are

1. `void setAttribute(String name, Object value)`
   This command binds `value`, the object given in the second parameter, to the attribute specified in `name`. By setting the second parameter to `null`, the attribute can be removed.
2. `Object getAttribute(String name)`
   The attribute value associated with `name` is returned.
3. `void removeAttribute(String name)`
   This method deletes the specified attribute from this session.

Configuration The server runs with the support of Apache Tomcat, which is a **servlet container**. A servlet container is a program that supports servlet execution. The servlets themselves are registered with the servlet container. URL requests made by a user are converted to specific servlet requests by the servlet container. The servlet container is responsible for initialising the servlets and delivering requests made by the client browser to the appropriate servlet.

The directory structure is as in Fig. 12.13. We store the HTML files in a directory named `Library`, which is a subdirectory of `webapps`, which, in turn, is a subdirectory of the home directory of Tomcat. The servlets are in the package `library`, which is stored in `Library/WEB-INF/classes`. The implementation of the backend classes such as `Member`, `Catalog`, etc. is in the package `basicImplementation`.

Our implementation requires that the user create an environment variable named `LIBRARY-HOME` that has as value the absolute path name of the directory that houses the HTML files.

The deployment descriptor elements are defined in a file called `web.xml`. While this file permits a large number of tags, our use of them is limited to mapping the URLs to servlets. To understand how this is done, first examine the following lines of XML code.

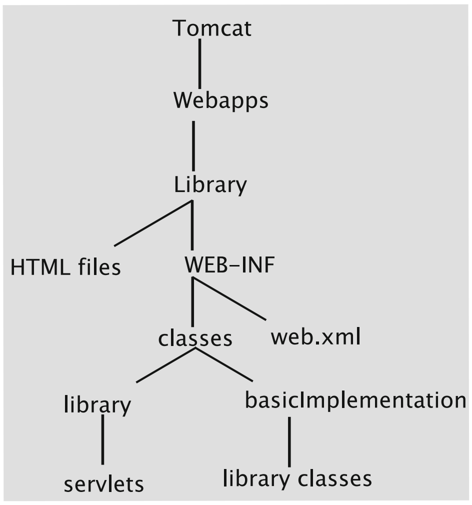


Fig. 12.13 Directory structure for the servlets

```
<servlet-mapping>
    <servlet-name>LoginServlet</servlet-name>
    <url-pattern>/login</url-pattern>
</servlet-mapping>
```

Thus when we write code such as

```
URL=login
```

in the HTML file, the string `login` is mapped to the servlet name `LoginServlet`.

But the servlet name given by the tag `<servlet-name>` is just a name that is mapped to the fully-qualified class name of the servlet as below.

```
<servlet>
    <servlet-name>LoginServlet</servlet-name>
    <servlet-class>library.Login</servlet-class>
</servlet>
```

In the `web.xml` file for our application, a servlet such as `library. IssueBook Initialization` (`<servlet-class>`) is mapped from the `<servlet-name>` of `IssueBookInitializationServlet`, which, in turn, is mapped from the URL pattern of `issuebookinitialization`, that is, the name of the original servlet in lower-case.

The list of superusers and their passwords is stored in a file named `Privileged Users`. The IP addresses of all client machines located in the library are listed in a file named `IPAddresses`. Both files are to be stored in the same directory that has the HTML files.

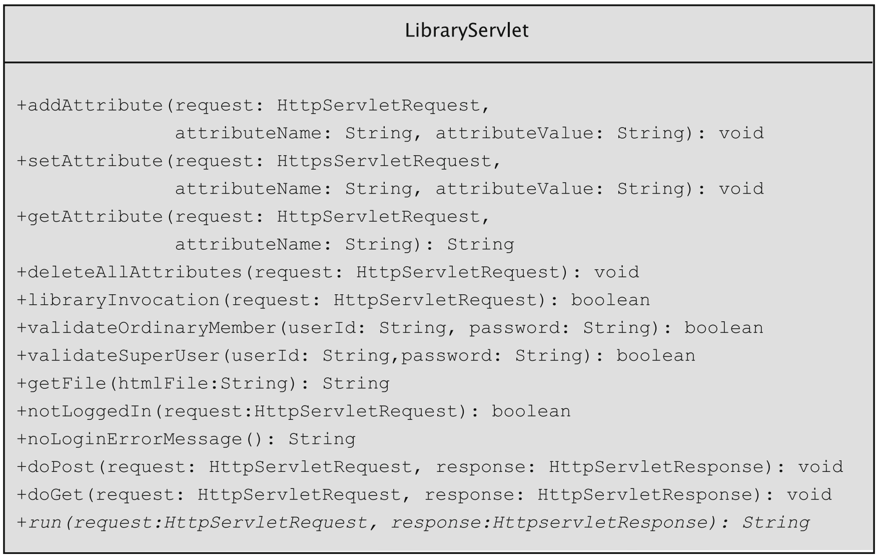
To run the system, first Tomcat needs to be started and then the library system needs to be accessed from a browser by typing in the URL of the Tomcat home concatenated with `/Library`. The file `index.html` in the `library` directory is then accessed; this file directs the request to the servlet `Login`.

Structure of servlets in the web-based library system A servlet receives data from a browser through a `HttpServletRequest` object. This involves parameter names and their values, IP address of the user, and so on. For example, when the form to add book is filled and the `Add` button is clicked, the servlet's `doPost` method is invoked. As we have seen earlier, this method has two parameters: a request parameter of type `HttpServletRequest` and a response parameter of type `HttpServletResponse`.

Each command is organised as a combination of one to three servlets. They need a number of common utility functions during the course of processing. These methods and `doPost` and `doGet` are collected into a class named `LibraryServlet`. This class has the structure shown in Fig. 12.14.

Most of the methods of `LibraryServlet` fall into one of five categories:

1. One group contains methods that store information about the user. This information includes the user id, the type of terminal from which the user has logged in, etc. and are stored in attributes associated with the session object. The methods are `addAttribute`, `setAttribute`, `getAttribute`, and `deleteAllAttributes`.
2. Methods to validate users and help assess access rights. The `validateSuper User` method checks whether the user is a superuser and `validateOrdinary`



```
LibraryServlet

+addAttribute(request: HttpServletRequest,
              attributeName: String, attributeValue: String): void
+setAttribute(request: HttpsServletRequest,
              attributeName: String, attributeValue: String): void
+getAttribute(request: HttpServletRequest,
              attributeName: String): String
+deleteAllAttributes(request: HttpServletRequest): void
+libraryInvocation(request: HttpServletRequest): boolean
+validateOrdinaryMember(userId: String, password: String): boolean
+validateSuperUser(userId: String,password: String): boolean
+getFile(htmlFile:String): String
+notLoggedIn(request:HttpServletRequest): boolean
+noLoginErrorMessage(): String
+doPost(request: HttpServletRequest, response: HttpServletResponse): void
+doGet(request: HttpServletRequest, response: HttpServletResponse): void
+run(request:HttpServletRequest, response:HttpservletResponse): String
```

Fig. 12.14 Class diagram for Library servlet

   Member does the same job for ordinary members. The method libraryInvocation returns true if and only if the user has logged in from a terminal located within the library.
3. The getFile method reads an HTML file and returns its contents as a String object.
4. The fourth group of methods are used for handling users who may have invoked a command without actually logging in. The method notLoggedIn returns true if and only if the user has not currently logged in. The method noLoginErrorMessage returns HTML code that displays an error message when a person who has not logged in attempts to execute a command.
5. The final group of commands deal with processing the request and responding to it. The doGet message calls doPost, which does some minimal processing needed for all commands and then calls the abstract run method, which individual servlets override.

In our design, all servlets inherit from LibraryServlet and will override the run method. Some of these are simple while others are more involved. Except for a couple of servlets that deal with log in, the structure of the run method is as below.

```
if the user has not logged in
   return an html page that displays "Not logged in"
else
   return an html page that is the result of processing the request
```

Execution flow Processing a request sometimes involves simply generating an HTML page, which is quite straightforward. This is best understood by following a sample command. We choose as example, the command to remove a book. A somewhat simplified sequence of what takes place in the course of the execution of this command is shown in Fig. 12.15.

As in the case of any command, the command is issued from the main menu. The URL associated with the text is as below:

```
<a href="removebookinitialization">Remove book</a>
```

The URL for the servlet is removebookinitialization; recall that this corresponds to the class RemoveBookInitialization, so when the link is clicked, the doPost method of that servlet is invoked. The code for this method is in LibraryServlet and is as follows:

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
       throws IOException, ServletException {
    response.setContentType("text/html");
    String page = run(request, response);
    if (!notLoggedIn(request)) {
      setAttribute(request, "page", page);
    }
    response.getWriter().println(page);
}
```

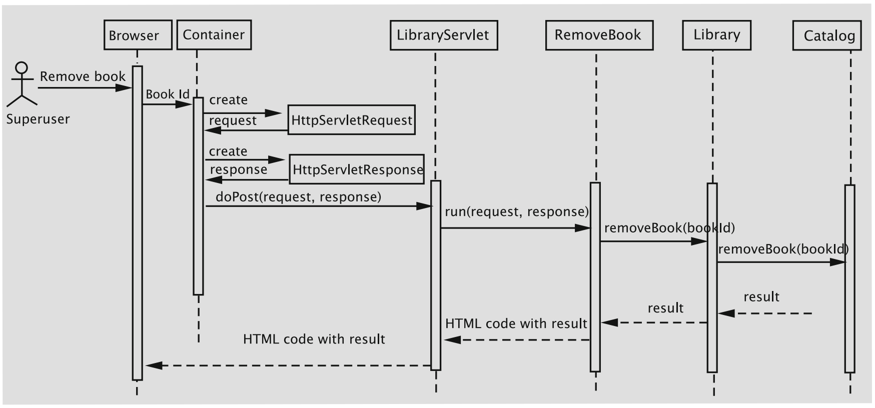


Fig. 12.15 Simplified sequence diagram for removing books

The first line in the method specifies the type of the file for the `response` object: whatever is written to the response object is treated as HTML. The `run` method is invoked, which is implemented within the subclass. This method returns HTML code as a `String` object and is saved in the attribute named `page` of the session. This helps in the following way. The system always remembers the last page displayed. If the user tries to log in from a different window of the browser, that page is redisplayed. It also helps when the user overwrites the current page by visiting some other site and wants to come back to the library system. Finally, the page is written out and gets displayed in the browser.

The check for whether the user has logged in was discussed before and is repeated for convenience.

```
public boolean notLoggedIn(HttpServletRequest request) {
  return request.getSession(false) == null;
}
```

The code for removing a book begins with the servlet `RemoveBook Initialization`, whose `run` method is given below.

```
package library;
import javax.servlet.*;
import javax.servlet.http.*;
public class RemoveBookInitialization extends LibraryServlet {
  public String run(HttpServletRequest request,
                    HttpServletResponse response) {
    if (notLoggedIn(request)) {
      return noLoginErrorMessage();
    }
    String htmlFile = getFile(HEADER);
    htmlFile = htmlFile.replace("TITLE", "Remove Book");
```

```
        htmlFile += getFile(GET_BOOK_ID);
        htmlFile = htmlFile.replace("GOTO_WITH_BOOKID", "removebook");
        htmlFile += getFile(CANCEL);
        htmlFile += getFile(END_PAGE);
        return htmlFile;
      }
    }
```

The first three lines in the `run` method check if the user has actually logged in and is not here via some other means. This can actually occur if the user has two windows connected to the library and the exit command is issued from one of the two. If that is indeed the case, the method `noLoginErrorMessage()` is called. This method simply generates an HTML page that displays ‘Not logged in’ and supplies a link to the log in screen.

In the case that the user is actually logged in, the HTML page is assembled. It includes reading four files: one to begin the HTML page and the other to end it. In between, a form to enter the book id and a link to cancel the command are inserted. As a consequence, the browser at the client displays a page that either requires the user to enter the id of a book that should be removed or click on a link to cancel the command and return to the main menu.

The process of ensuring that the user had logged in and using the header file to begin assembling the HTML file is common to all servlets, so we will not explain these actions in further discussion.

The HTML code for entering the book id is given below.

```
<form action="GOTO_WITH_BOOKID" method="post">
<table>
<tr>
  <td align="right">Id:</td>
  <td><input type="text" name="bookId"></td>
</tr>
<td><br><input type="submit" value="Enter Book Id"></td>
</tr>
</table>
</form>
```

In the normal course of action, the user would enter a book id and click the button labelled `Enter Book Id`. Notice the lines

```
<form action="GOTO_WITH_BOOKID" method="post">
```

in the HTML file and the line

```
htmlFile = htmlFile.replace("GOTO_WITH_BOOKID", "removebook");
```

in the servlet. The place holder GOTO_WITH_BOOKID is replaced by the URL removebook. Therefore, when the user submits the form, the RemoveBook servlet is initiated. The code for this class is given below.

```
package library;
import javax.servlet.*;
import javax.servlet.http.*;
public class RemoveBook extends LibraryServlet {
  public String run(HttpServletRequest request,
                    HttpServletResponse response) {
    if (notLoggedIn(request)) {
      return noLoginErrorMessage();
    }
    String id = request.getParameter("bookId");
    String htmlFile = getFile(HEADER);
    htmlFile += getFile(COMMAND_COMPLETED);
    htmlFile += getFile(END_PAGE);
    String result;
    if ((result = library.removeBook(id)) == null) {
      htmlFile = htmlFile.replace("TITLE", "Book not Removed");
      result = "Book could not be removed";
    } else {
      htmlFile = htmlFile.replace("TITLE", "Book Removed");
    }
    htmlFile = htmlFile.replace("RESULT", result);
    htmlFile = htmlFile.replace("REPLACE_JS", "removebookinitialization");
    htmlFile = htmlFile.replace("REPLACE_COMMAND", "Remove Book");
    return htmlFile;
  }
}
```

The path to the run method is once again via the doPost method in Library Servlet. The id of the book to be removed is retrieved by invoking the command getParameter on the request object. We then start assembling the HTML page to respond to the request. The removeBook method in the Library class is invoked and the result of the command is used to replace the place holder RESULT. The servlet provides two choices at this stage: the user may remove another book or go back to the main menu. The option to go back to the main menu is common to all commands, so it is hard-coded in the HTML file for command completion. However, the command to be repeated depends on what command we are in, so the place holders REPLACE_JS and REPLACE_COMMAND are replaced by the URL of the servlet and an appropriate piece of text that the user can click on.

The code for some other commands (returning a book, adding a book, adding a member, and processing holds) are quite similar and warrants no further explanation. The code for saving and retrieving data are simpler. The implementation for the other commands, issuing a book, placing a hold, removing a hold, and printing transactions are more complicated, but they are all similar. So we next explain the implementation for issuing a book.

Issuing books Issuing books is complicated by the fact that an ordinary member may self-issue a book or may ask a library staff member, a superuser, to issue the book for himself/herself. (See the state transition diagram in Fig. 12.11.) In the former case, we need to skip asking the member's id and in the latter case, the system must present a screen for entering the member id.

Like all other commands, the user clicks on a link to issue books; the HTML file contains the lines

```
<td valign="top" width="160">
<a href="issuebookinitialization">Issue book</a>
<br></td>
```

The click on `Issue book` causes the servlet `IssueBookInitialization` to execute. This servlet checks if the user is a superuser, and if so, a screen to accept the member id is displayed; otherwise, the member to whom the book should be issued is known and a screen to accept a book id is displayed. The code is given below.

```
public class IssueBookInitialization extends LibraryServlet {
  public String run(HttpServletRequest request, HttpServletResponse response) {
    if (notLoggedIn(request)) {
      return noLoginErrorMessage();
    }
    boolean privileged = getAttribute(request, "userType").equals("Privileged");
    String memberId = getAttribute(request, "currentUserId");
    String htmlFile = getFile(HEADER);
    htmlFile = htmlFile.replace("TITLE", "Issue Book");
    if (privileged) {
      htmlFile += getFile(GET_MEMBER_ID);
      htmlFile = htmlFile.replace("GOTO_WITH_MEMBERID", "issuebookgetmemberid");
    } else {
      htmlFile += getFile(GET_BOOK_ID);
      htmlFile = htmlFile.replace("GOTO_WITH_BOOKID", "issuebookgetbookid");
    }
    htmlFile += getFile(CANCEL);
    htmlFile += getFile(END_PAGE);
    return htmlFile;
  }
}
```

We now discuss how we remember the member for whom the book is to be issued. Recall that the session object can store attributes and that commands such as issuing a book and placing a hold are always carried out for a specific member. That member's id is stored in the attribute `currentUserId`. If the session was for an ordinary member, the value for this attribute is the member's id itself. Otherwise, when a superuser is logged in, the value changes depending on the member for whom the command is being carried out; when the command does not involve a member (for example, the superuser is adding books), the value of this attribute is the empty string (`""`).

From the above discussion, clearly,

```
String memberId = getAttribute(request, "currentUserId");
```

would be the empty string if the user is a superuser and the logged-in-member's id otherwise.

The servlet IssueBookGetMemberId retrieves the id of the member to whom books should be issued:

```
String memberId = request.getParameter("memberId");
```

If the member id is invalid, the HTML file consists of an error message and a form to accept the member id. In this case, note that control will come back to the same servlet.

```
if (!library.searchMembership(memberId)) {
  htmlFile +=  getFile(COMMAND_COMPLETED);
  htmlFile = htmlFile.replace("RESULT", "Could not locate member");
  htmlFile += getFile(GET_MEMBER_ID);
  htmlFile = htmlFile.replace("GOTO_WITH_MEMBERID", "issuebookgetmemberid");
  htmlFile = htmlFile.replace("REPLACE_JS", "");
  htmlFile = htmlFile.replace("REPLACE_COMMAND", "");
}
```

If the member id is valid, it is remembered in the attribute currentUserId and a form for capturing the book id is created.

```
setAttribute(request, "currentUserId", memberId);
htmlFile += getFile(GET_BOOK_ID);
htmlFile = htmlFile.replace("GOTO_WITH_BOOKID", "issuebookgetbookid");
htmlFile += getFile(CANCEL);
```

The IssueBookGetBookId servlet gets the book id from the form, retrieves the value of the attribute currentUserId to get the member id and calls the issueBook method of Library. The result is then used to replace the string RESULT in the commandCompleted HTML file.

```
String bookId = request.getParameter("bookId");
String memberId = getAttribute(request, "currentUserId");
Book book;
String result;
if ((book = library.issueBook(memberId, bookId)) == null) {
  result = "Book could not be issued";
} else {
  result = book.getTitle() + "issued.";
}
htmlFile = htmlFile.replace("RESULT", result);
htmlFile += getFile(GET_BOOK_ID);
htmlFile = htmlFile.replace("GOTO_WITH_BOOKID", "issuebookgetbookid");
```

The result of invoking issueBook is stored in the string RESULT as has been the case for other commands. (This part of the HTML code also has the option to return to the main menu.) We then concatenate the HTML file that contains the form to enter a book id, so the user has the option to enter another book id. The system continues executing this servlet until the user decides to go back to the main menu.

Renewing books Book renewal begins in the same manner as book issuing: the member id needs to be accepted if the user is a superuser; otherwise, that step can be bypassed.

To allow renewal, the title and due date all of the books checked out to the user must be displayed. Also, for each book a checkbox needs to be shown, so the user can check it if he/she wants the book to be renewed. The HTML code, stored in the file `renewBook.html`, for this part of the process is given below.

```
<tr>
 <td> TITLE </td>
 <td> DUE_DATE </font> </td>
 <td> <input type="checkbox" name="RENEW" /> </td>
</tr>
```

The type `checkbox` denotes a checkbox control, which the user can click to indicate that a book should be renewed. The three strings, `TITLE`, `DUE_DATE`, and `RENEW` are placeholders for the book title, book id, and the name of the checkbox control. The idea is that the above five lines of code will be replicated as many times as the number of books checked out. The names of the possibly multiple checkboxes need to be different, which is the task of the servlets.

The list of books must be assembled from two servlets: `RenewBooks Initialization` if the user is an ordinary member and `RenewBooksGet MemberId` if the user is a superuser. Since the code to perform this task is a bit lengthy, it is extracted into `LibraryServlet`.

The code, given below, first gets an iterator for the books checked out. The HTML file is built up from the file `renewBook.html` we described earlier. The strings `TITLE` and `DUE_DATE` are respectively replaced by the book's title and due date. A unique name for the checkbox is generated by replacing the string `RENEW` by the concatenation of `renew` and a counter that is incremented once per loop iteration.

Now, for some slightly more complicated task. The `RenewBooks` servlet must somehow discover the book id and other details of the books that are to be renewed. Also, we list the title and due date (possibly changed) of each book to be renewed and a status message that says whether the book was renewed or not. This demands that we remember the details of all the books in the order we displayed them. These are stored in the attributes `bookId`, `title`, and `dueDate`, each concatenated with the value of the counter. Also, the number of books displayed is also stored in the attribute `numberOfBooks`.

```
protected String assembleBooks(HttpServletRequest request, String memberId) {
  int counter = 0;
  String htmlFile = "";
  for (Iterator issuedBooks = library.getBooks(memberId);
              issuedBooks.hasNext(); counter++) {
```

```
        Book book = (Book) (issuedBooks.next());
        htmlFile += getFile(RENEW_BOOKS);
        htmlFile = htmlFile.replace("TITLE", book.getTitle());
        htmlFile = htmlFile.replace("DUE_DATE", book.getDueDate());
        setAttribute(request, "bookId" + counter, book.getId());
        setAttribute(request, "title" + counter, book.getTitle());
        setAttribute(request, "dueDate" + counter, book.getDueDate());
        htmlFile = htmlFile.replace("RENEW", "renew" + counter);
      }
      setAttribute(request, "numberOfBooks", counter + "");
      return htmlFile;
    }
```

As we mentioned above, when the user responds, the servlet `RenewBooks` comes into play. The servlet enters a loop iterating as many times as there are number of books checked out. If the checkbox is clicked, the condition `request.getParameter("renew" + counter) != null` evaluates to `true`. The servlet retrieves the book's title from the attribute `bookTitle` and replaces the TITLE in the HTML file `renewedBook.html`. An attempt is made to renew the book using the member id and book id obtained from stored attributes. If the renewal is successful, the string `DUE_DATE` in `renewedBook.html` is replaced by the new due date. Otherwise, the old due date replaces the string.

Notice also that we delete all of the attributes created for the renewal process.

```
String memberId = getAttribute(request, "currentUserId");
int numberOfBooks = Integer.parseInt(getAttribute(request, "numberOfBooks"));
for (int counter = 0; counter < numberOfBooks; counter++) {
  if (request.getParameter("renew" + counter) != null) {
    htmlFile += getFile(RENEWED_BOOKS);
    String bookId = getAttribute(request, "bookId" + counter);
    String title = getAttribute(request, "title" + counter);
    String dueDate = getAttribute(request, "dueDate" + counter);
    htmlFile = htmlFile.replace("TITLE", title);
    Book book = library.renewBook(bookId, memberId);
    if (book != null) {
      htmlFile = htmlFile.replace("RENEWED", "renewed");
      htmlFile = htmlFile.replace("DUE_DATE", book.getDueDate());
    } else {
      htmlFile = htmlFile.replace("RENEWED", "book not renewable");
      htmlFile = htmlFile.replace("DUE_DATE", dueDate);
    }
    request.getSession(false).removeAttribute("bookId" + counter);
    request.getSession(false).removeAttribute("title" + counter);
    request.getSession(false).removeAttribute("dueDate" + counter);
 }
}
request.getSession(false).removeAttribute("numberOfBooks");
```

Logging in and logging out When the class `LibraryServlet` is loaded, it reads the files `PrivilegedUsers` and `IPAddresses` and copies the information to main memory. When a user logs in, we have seen that control goes to the `Login` servlet. It assembles the log in screen for display by the browser.

Assume now that the user types in a user id and password and sends them to the server. The `Index` servlet reads in the user id and password and calls a method named `getMenu` in the class `MenuBuilder`. This class is responsible for checking the validity of the user and returning the appropriate menu. The class `MenuBuilder` itself is not a servlet, so to utilise the methods of `LibraryServlet`, it needs the reference to the `Index` servlet. To call some of these methods, `MenuBuilder` also needs the request object. For uniformity, we also pass the response object, although it is not currently used. The method thus has 5 parameters: a reference to the `Index` servlet, the request and response objects, and the user-id and password.

First, the code checks if the user is a superuser by calling the method `validate SuperUser` of `LibraryServlet`, and if so, the attribute `userType` is given the value `Privileged`. Otherwise, the `LibraryServlet` class's `validate OrdinaryMember` method is called to see if the user is a member of the library; in that case, the `userType` attribute is set as `Ordinary`. Also, note the use of the boolean variables `privileged` and `validated`. In the event of an invalid user-id–password combination, a `null` value is returned to the `Index` servlet, which redisplays the log-in screen with an error message.

```
if (servlet.validateSuperUser(userId, password)) {
  servlet.setAttribute(request, "userType", "Privileged");
  validated = true;
} else if (servlet.validateOrdinaryMember(userId, password)) {
  servlet.setAttribute(request, "userType", "Ordinary");
  privileged = false;
  validated = true;
}
if (!validated) {
  return null;
}
```

With a successful log-in, the method checks whether the terminal used is within the library premises or outside. The attribute `location` reflects this assessment. The `currentUserId` is set to the user's id for ordinary users and to the empty string (`""`) for privileged users. (We have already seen how this attribute is used to handle commands such as Issue Book.)

```
if (servlet.libraryInvocation(request)) {
  servlet.setAttribute(request, "location", "Library");
  location = LIBRARY;
} else {
  servlet.setAttribute(request, "location", "Outside");
}
servlet.setAttribute(request, "userId", userId);
if (!privileged) {
  servlet.setAttribute(request, "currentUserId", userId);
} else {
  servlet.setAttribute(request, "currentUserId", "");
}
return getMenu(servlet, privileged, location);
```

The final step is to return the appropriate menu. This is done by the method `getMenu` that has three parameters. The code assembles the HTML page by reading from four different files in addition to the files for beginning and ending the page. These meet the requirements we set forth under 'Developing User Requirements'. If the user has logged in from the library, the Issue Book command is inserted into the menu. For privileged users, commands such as Add and Remove Book are inserted. Ordinary members always get to issue commands such as placing a hold and removing a hold. These commands are also available to superusers who log in from a library terminal. Finally, the exit command is available to all users from anywhere.

```
  private String getMenu(LibraryServlet servlet, boolean privileged,
                                                 boolean location) {
    boolean OUTSIDE = false;
    boolean LIBRARY = true;
    String html = servlet.getFile(LibraryServlet.HEADER);
    if (location == LIBRARY) {
      html += servlet.getFile(LibraryServlet.LIBRARY_COMMANDS);
    }
    if (privileged && location == LIBRARY) {
      html += servlet.getFile(LibraryServlet.PRIVILEGED_COMMANDS);
    }
    if (!privileged || location == LIBRARY) {
      html += servlet.getFile(LibraryServlet.GLOBAL_COMMANDS);
    }
    html += servlet.getFile(LibraryServlet.EXIT_COMMAND);
    html += servlet.getFile(LibraryServlet.END_PAGE);
    return html;
  }
```

There is a third version of the `getMenu` method, which gets invoked when a user goes back to main menu from the middle or at the end of a command. In this case, the user id and password are already known; so the attributes are read from the session object to determine what the menu should be. The menu itself is created using the 3-parameter `getMenu` method (the second version) we just discussed. This third version also sets the attribute `currentUserid` to the empty string (`""`) for privileged users. The critical part of the code is given below.

```
  if (!servlet.getAttribute(request, "userType").equals("Privileged")) {
      privileged = false;
  } else {
      servlet.setAttribute(request, "currentUserId", "");
  }
```

```
if (servlet.getAttribute(request, "location").equals("Library")) {
    location = LIBRARY;
}
return getMenu(servlet, privileged, location);
```

12.4 Discussion and Further Reading

RMI provides a level of abstraction much higher than the traditional communication mechanism in networks, viz. sockets. A socket is an endpoint of a communication channel to or from which data is transmitted in the network. Sockets are analogous to phones and a socket allocated on a machine is uniquely associated with a process running on it. The type of socket associated with a process depends on the transport layer in use (TCP or UDP, for example). A socket can have an associated port number using which processes may send messages to it. Socket programming is possible in many modern programming languages including C and Java. The book by Stevens [1] is an excellent reference for programming in C in the Unix environment.

The difficulty with the socket model of programming is that it is very different from the imperative paradigm of programming, which involves procedure calls. Remote Procedure Call (RPC) [2] provides an improved model that allows programs to issue a call to a procedure in another address space, which could actually be in a remote computer on a network. For the most part, this can be done without worrying about the underlying network. The code that runs on a centralised machine would run without too much modification on a network as well. RPC is a popular way of implementing distributed systems using the C programming language. Businesses that have made use of RPC include Xerox, Sun, and Microsoft. RMI is essentially RPC extended to the world of object-oriented systems. For a description of RPC, see RFC 707.

The Common Object Request Broker Architecture (CORBA), standardised by the Object Management Group (OMG), is another approach to distributed object-based computing. It allows a distributed, heterogeneous collection of objects to interoperate, and automates many common network programming tasks such as object registration, location, and activation, error-handling, parameter marshalling and demarshalling, security control and concurrency control.

Like RMI, the services that a CORBA object provides are defined by its interface. Again, as in RMI, object references are really of interface types. The Object Request Broker (ORB) is responsible for delivering requests from a client to a remote object and to return the results.

The ORB does a little more than just send requests and receive replies. Unlike RMI, CORBA is language independent (it is also platform independent), and the ORB plays a crucial role in this. The client may issue the request in a programming language different from that of the CORBA object to which it issues the request. The ORB does the necessary translation between programming languages. Language bindings are defined for all popular programming languages. For a quick overview of CORBA, refer to [3].

As mentioned in the chapter, the Java Servlet technology is just one of the tools available for creating web-based systems. PHP is a scripting language that usually runs on the server side. It can have HTML code embedded into it and outputs web pages. ASP.NET is another competing scripting technology from Microsoft for building web-based applications. JSP is similar to PHP and ASP, the difference being that we intersperse Java code with HTML code to create dynamic web pages. Other technologies such as Ruby on Rails (RoR) are also available.

The World Wide Web Consortium develops technologies for the utilisation of the web. This includes specifications for HTML and HTTP. The reader is encouraged to take a look at their site at http://www.w3.org/ to get an overview of the work provided by that group.

12.5 Exercises

1. Consider the implementation of the library system using Java RMI with a single server that runs classes such as `Library`, `Catalog`, `Book`, etc., multiple clients, each running an instance of `UserInterface`. Which classes do you need to modify? What other modifications do you need to make? Examine the parameters and return values for the remote method calls and verify that they all conform to RMI requirements.
2. Modify the distributed library system so that a command to list the catalog is available.
3. Consider the solution to Question 2. Incorporate a mechanism by which a user can place holds on books by selecting one or more books from the catalog listing.
4. Learn another technology for implementing web-based systems. A relatively easy exercise would be to learn Java Server Pages (JSP). Re-implement the library system using JSP. What are the advantages of JSP compared to Java servlets?
5. Suppose that instead of allowing no commands (other than exit) to be issued by a superuser from terminals outside the library, we want them to be able to do some telecommuting from outside. Make changes to the web-based system so that the commands to save and retrieve data and the command to process holds can be done by a superuser from anywhere.

References

1. R. Stevens, *UNIX Network Programming* (Prentice Hall, Englewood Cliffs, 1998)
2. A.D. Birrell, B.J. Nelson, Implementing remote procedure calls. ACM Trans. Comput. Syst. **2**(1), 39–59 (1984)
3. O.M. Group. Corba basics. http://www.omg.org/gettingstarted/corbafaq.htm