CHAPTER 7

Distributed Objects

Until now we have confined our attention to using the message-passing paradigm in distributed computing. Using the message-passing paradigm, processes exchange data and, by observing mutually agreed upon protocols, collaborate to accomplish desirable tasks. Application program interfaces based on this paradigm, such as the Java unicast and multicast socket APIs, provide the abstraction that hides the details of lower-layer network communications and allows us to write code that performs IPC using relatively simple syntax. This chapter introduces a paradigm that offers further abstraction, distributed objects.

7.1 Message Passing versus Distributed Objects

The message-passing paradigm is a natural model for distributed computing in the sense that it mimics interhuman communications. It is an appropriate paradigm for network services where processes interact with each other through the exchange of messages. But the abstraction provided by this paradigm may not meet the needs of some complex network applications for the following reasons:

- Basic message passing requires that the participating processes be **tightly coupled**. Throughout their interaction, the processes must be in direct communication with each other. If communication is lost between the processes (owing to failures in the communication link, in the systems, or in one of the processes), the collaboration fails. As an example, consider a session of the *Echo* protocol: If the communication between the client and the server is disrupted, the session cannot continue.

A distributed object is provided, or **exported**, by a process, here called the **object server**. A facility, here called an **object registry**, or **registry** for short, must be present in the system architecture in order for the distributed object to be registered.

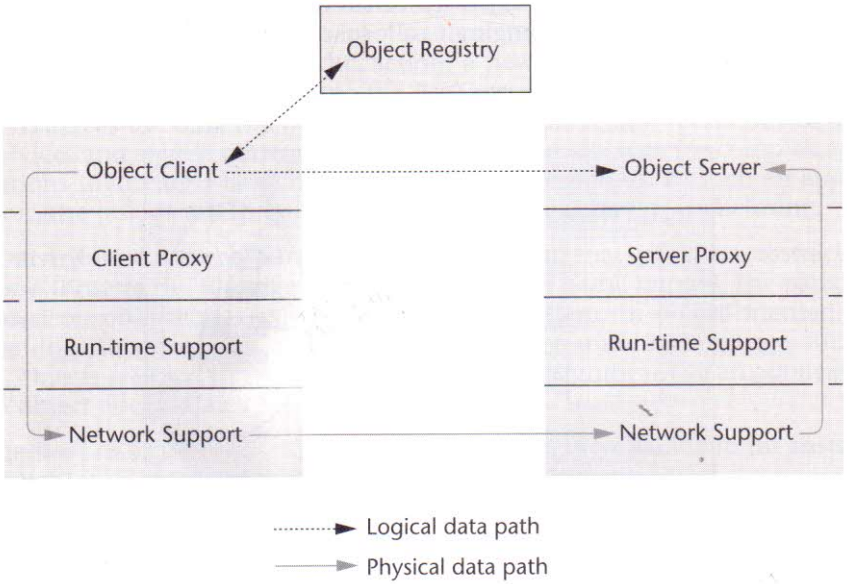


Figure 7.2 An archetypal distributed object system.

In programming language, a reference is a "handle" for an object; it is a representation through which an object can be located in the computer where the object resides.

To access a distributed object, a process, the **object client**, looks up the registry for a **reference** to the object. This reference is used by the object client to make calls to the methods of the remote object, or **remote methods**. Logically, the object client makes a call directly to a remote method. In reality, the call is handled by a software component, called a **client proxy**, which interacts with the software on the client host to provide the **run-time support** for the distributed object system. The run-time support is responsible for the interprocess communication needed to transmit the call to the remote host, including the marshaling of the argument data that needs to be transmitted to the remote object.

The term proxy, in the context of distributed computing, refers to a software component that serves as an intermediary between other software components.

A similar architecture is required on the server side, where the run-time support for the distributed object system handles the receiving of messages and the unmarshaling of data, and forwards the call to a software component called the **server proxy**. The server proxy interfaces with the distributed object to invoke the method call locally, passing in the unmarshaled data for the arguments. The method call triggers the performance of some tasks on the server host. The outcome of the execution of the method, including the marshaled data for the return value, is forwarded by the server proxy to the client proxy, via the run-time support and network support on both sides.

CHAPTER 7

Distributed Objects

Until now we have confined our attention to using the message-passing paradigm in distributed computing. Using the message-passing paradigm, processes exchange data and, by observing mutually agreed upon protocols, collaborate to accomplish desirable tasks. Application program interfaces based on this paradigm, such as the Java unicast and multicast socket APIs, provide the abstraction that hides the details of lower-layer network communications and allows us to write code that performs IPC using relatively simple syntax. This chapter introduces a paradigm that offers further abstraction, distributed objects.

7.1 Message Passing versus Distributed Objects

The message-passing paradigm is a natural model for distributed computing in the sense that it mimics interhuman communications. It is an appropriate paradigm for network services where processes interact with each other through the exchange of messages. But the abstraction provided by this paradigm may not meet the needs of some complex network applications for the following reasons:

- Basic message passing requires that the participating processes be **tightly coupled**. Throughout their interaction, the processes must be in direct communication with each other. If communication is lost between the processes (owing to failures in the communication link, in the systems, or in one of the processes), the collaboration fails. As an example, consider a session of the *Echo* protocol: If the communication between the client and the server is disrupted, the session cannot continue.

- The message-passing paradigm is **data-oriented**. Each message contains data marshaled in a mutually agreed upon format, and each message is interpreted as a request or response according to the protocol. The receiving of each message triggers an action in the receiving process.

  For example, in the *Echo* protocol, the receiving of a message from process *p* elicits in the *Echo* server this action: a message containing the same data is sent to *p*. In the same protocol, the receiving of a message from the *Echo* server by process *p* triggers this action: a new message is solicited from the user, and the message is sent to the *Echo* server.

  Whereas the data orientation of the paradigm is appropriate for network services and simple network applications, it is inadequate for complex applications involving a large mix of requests and responses. In such an application, the task of interpreting the messages can become overwhelming.

The **distributed object paradigm** is a paradigm that provides abstractions beyond those of the message-passing model. As its name implies, the paradigm is based on objects that exist in a distributed system. In object-oriented programming, supported by an object-oriented programming language such as Java, objects are used to represent an entity that is significant to an application. Each object encapsulates

- the state or data of the entity—in Java, such data is contained in the **instance variables** of each object;
- the **operations** of the entity, through which the state of the entity can be accessed or updated—in Java, these are the **methods**.

To illustrate, consider objects of the *DatagramMessage* class presented in Figure 5.12 (in Chapter 5). Each object instantiated from this class contains three state data items: a message, the sender's address, and the sender's port number. In addition, each object contains three operations: (1) a method *putVal*, which allows the values of these data items to be modified, (2) a *getMessage* method, which allows the current value of the message to be retrieved, and (3) a *getAddress* method, which allows the sender's address to be retrieved.

Although we have used objects such as *DatagramMessage* in previous chapters, those are **local** objects instead of **distributed** objects. Local objects are objects whose methods can only be invoked by a **local process**, a process that runs on the same computer on which the object exists. A distributed object is one whose methods can be invoked by a **remote process**, a process running on a computer connected via a network to the computer on which the object exists. In a distributed object paradigm, network resources are represented by distributed objects. To request service from a network resource, a process invokes one of its operations or methods, passing data as parameters to the method. The method is executed on the remote host, and the response is sent back to the requesting process as a return value. Compared to the message-passing paradigm, the distributed objects paradigm is **action-oriented**. The focus is on the invocation of the operations, while the data passed takes on a secondary role (as parameters and return values). Although less intuitive to human beings, the distributed-object paradigm is more natural to object-oriented software development.

Figure 7.1 illustrates the paradigm. A process running in host *A* makes a method call to a distributed object residing on host *B*, passing with the call the data for the arguments, if any. The method call invokes an action performed by the method on host *A*, and a return value, if any, is passed from host *A* to host *B*. A process that makes use of a distributed object is said to be a **client process** of that object, and the methods of the object are called **remote methods** (as opposed to local methods, or methods belonging to a local object) to the client process.

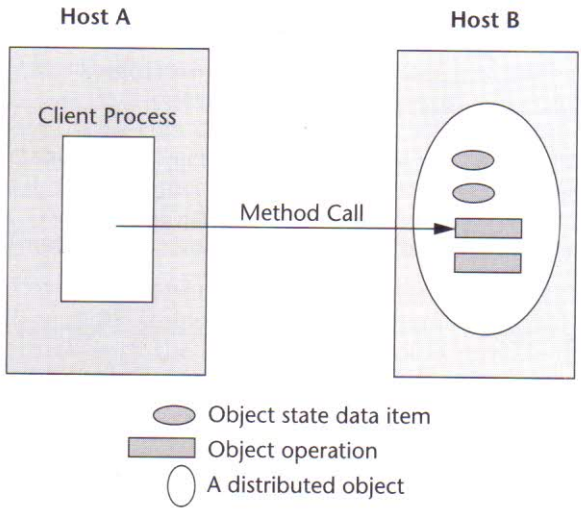


Figure 7.1 The distributed objects paradigm.

In the rest of the chapter, we will present an archetypal facility that supports the distributed object paradigm; then we will explore a sample facility: the **Java Remote Method Invocation (RMI)**.

7.2 An Archetypal Distributed Object Architecture

The premise behind a distributed object system is to minimize the programming differences between remote method invocations and local method calls, thereby allowing remote methods to be invoked in an application using syntax similar to that used for local method invocations. In actuality, there are differences, because remote method invocations involve communication between independent processes, and hence issues such as data marshaling and event synchronization need to be addressed. Such differences are encapsulated in the architecture.

Figure 7.2 presents an archetypal architecture for a facility that supports the distributed objects paradigm.

- The message-passing paradigm is **data-oriented**. Each message contains data marshaled in a mutually agreed upon format, and each message is interpreted as a request or response according to the protocol. The receiving of each message triggers an action in the receiving process.

  For example, in the *Echo* protocol, the receiving of a message from process *p* elicits in the *Echo* server this action: a message containing the same data is sent to *p*. In the same protocol, the receiving of a message from the *Echo* server by process *p* triggers this action: a new message is solicited from the user, and the message is sent to the *Echo* server.

  Whereas the data orientation of the paradigm is appropriate for network services and simple network applications, it is inadequate for complex applications involving a large mix of requests and responses. In such an application, the task of interpreting the messages can become overwhelming.

The **distributed object paradigm** is a paradigm that provides abstractions beyond those of the message-passing model. As its name implies, the paradigm is based on objects that exist in a distributed system. In object-oriented programming, supported by an object-oriented programming language such as Java, objects are used to represent an entity that is significant to an application. Each object encapsulates

- the state or data of the entity—in Java, such data is contained in the **instance variables** of each object;
- the **operations** of the entity, through which the state of the entity can be accessed or updated—in Java, these are the **methods**.

To illustrate, consider objects of the *DatagramMessage* class presented in Figure 5.12 (in Chapter 5). Each object instantiated from this class contains three state data items: a message, the sender's address, and the sender's port number. In addition, each object contains three operations: (1) a method *putVal*, which allows the values of these data items to be modified, (2) a *getMessage* method, which allows the current value of the message to be retrieved, and (3) a *getAddress* method, which allows the sender's address to be retrieved.

Although we have used objects such as *DatagramMessage* in previous chapters, those are **local** objects instead of **distributed** objects. Local objects are objects whose methods can only be invoked by a **local process**, a process that runs on the same computer on which the object exists. A distributed object is one whose methods can be invoked by a **remote process**, a process running on a computer connected via a network to the computer on which the object exists. In a distributed object paradigm, network resources are represented by distributed objects. To request service from a network resource, a process invokes one of its operations or methods, passing data as parameters to the method. The method is executed on the remote host, and the response is sent back to the requesting process as a return value. Compared to the message-passing paradigm, the distributed objects paradigm is **action-oriented**: The focus is on the invocation of the operations, while the data passed takes on a secondary role (as parameters and return values). Although less intuitive to human beings, the distributed-object paradigm is more natural to object-oriented software development.

7.3 Distributed Object Systems

The distributed object paradigm has been widely adopted in distributed applications, for which a large number of toolkits based on the paradigm are available. Among the most well known of such toolkits are

- Java Remote Method Invocation (RMI),
- systems based on the Common Object Request Broker Architecture (CORBA),
- the Distributed Component Object Model (DCOM), and
- toolkits and APIs that support the Simple Object Access Protocol (SOAP).

Of these, the most straightforward is the Java RMI [java.sun.com/products, 7; java.sun.com/doc, 8; developer.java.sun.com, 9; java.sun.com/marketing, 10], which we will discuss in this chapter in detail. CORBA [corba.org, 1] and its implementations are the subjects of Chapter 9. SOAP [w3.org, 2] is a Web-based protocol and will be introduced in Chapter 11 when we discuss Web-based applications. DCOM [microsoft.com, 3; Grimes, 4] is beyond the scope of this textbook; interested readers should consult the references.

It is not possible to cover all of the existing distributed object facilities, and it is safe to say that toolkits supporting the paradigm will continue to emerge. Familiarizing yourself with the Java RMI API should provide the fundamentals and prepare you for learning the details of similar facilities.

7.4 Remote Procedure Calls

Remote Method Invocation (RMI) has its origin in a paradigm called **Remote Procedure Call**.

Procedural programming predates object-oriented programming. In procedural programming a procedure or a function is a control structure that provides the abstraction for an action. The action of a function is invoked by a function call. To allow for variability, a function call may be accompanied by a list of data, known as arguments. The value or the reference of each argument is said to be passed to the function, and it may determine the action actualized by the function. The conventional procedure call is a call to a procedure residing in the same system as the caller, and thus the procedure call can be termed a **local procedure call**.

In the remote procedure call model, a procedure call is made by one process to another, possibly residing in a remote system, with data passed as arguments. When a process receives a call, the actions encoded in the procedure are executed, the caller is notified of the completion of the call, and a return value, if any, is transmitted from the callee to the caller. Figure 7.3 illustrates the RPC paradigm.

Based on the RPC model, a number of application programming interfaces have emerged. These APIs provide remote procedure calls using syntax and semantics resembling those of local procedure calls. To mask the details of interprocess

communications, each remote procedure call is transformed by a tool called **rpcgen** to translate a local procedure call directed to a software module commonly termed a **stub** or, more formally, a **proxy**. Via the proxy, messages representing the procedure call and its arguments are passed to the remote machine.

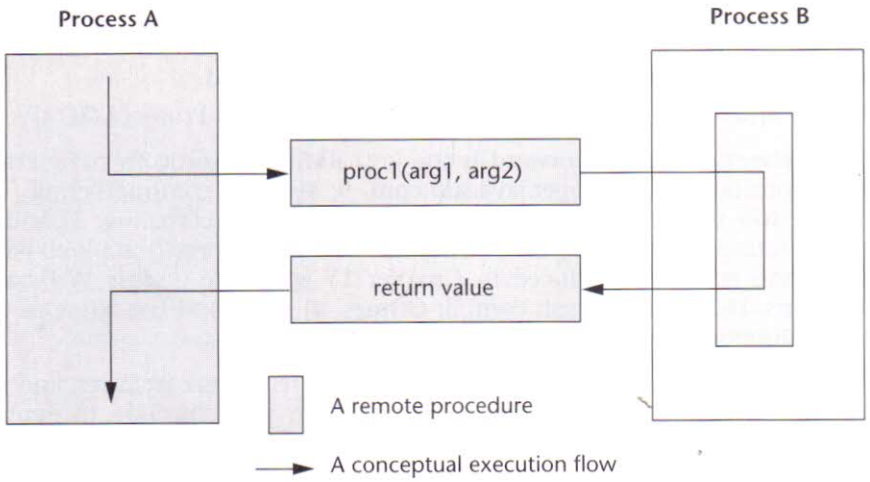


Figure 7.3 The Remote Procedure Call paradigm.

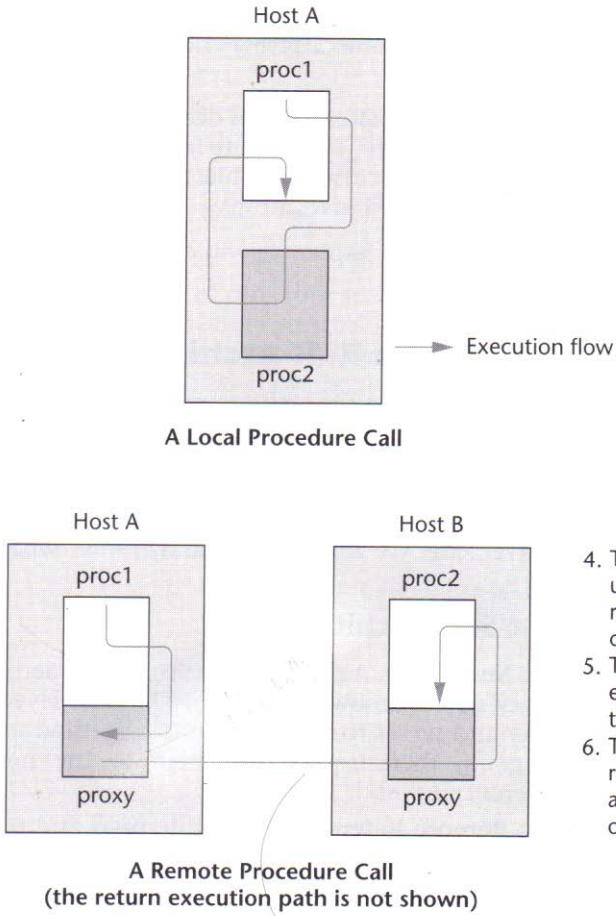
At the other end, a proxy receives the message and transforms it to a local procedure call to the remote procedure. Figure 7.4 illustrates the behind-the-scenes transformation of a remote procedure call into local procedure calls and the intervening message passing.

Note that a proxy is employed on either side to provide the run-time support needed for the interprocess communications, carrying out the necessary data marshaling and socket calls.

Since its introduction in the early 1980s, the Remote Procedure Call model has been widely in use in network applications. There are two prevalent APIs for this paradigm. One, the **Open Network Computing Remote Procedure Call** [ietf.org, 5], evolved from the RPC API that originated from Sun Microsystems, Inc., in the early 1980s. The other well-known API is the **Open Group Distributed Computing Environment** (DCE) RPC [opennc.org, 6]. Both APIs provide a tool, **rpcgen**, for transforming remote procedure calls to local procedure calls to the stub.

In spite of its historical significance, we will not study RPC in detail for the following reasons:

- RPC, as its name implies, is procedure-oriented. RPC APIs employ syntax for procedural or function calls. Hence they are more suitable to programs written in a procedural language such as C. They are, however, not appropriate



1. proc1 on Host A makes a call to proc2 on Host B.
2. The run-time support maps the call to a call to the proxy on Host A.
3. The proxy marshals the data and makes an IPC call to a proxy on Host B.

4. The proxy on Host B unmarshals the data received and issues a call to proc2.
5. The code in proc2 is executed and returns to the proxy on Host B.
6. The proxy marshals the return value and makes an IPC call to the proxy on Host A.

7. The proxy receives the return value, unmarshals the data, and forwards the return value to proc1, which resumes its execution flow.

Figure 7.4 Local Procedure Call versus Remote Procedure Call.

for programs written in Java, the object-oriented language we have adopted for this book.

- In lieu of RPC, Java provides the **Remote Method Invocation** API, which is object-oriented and has a syntax that is more accessible than RPC.

7.5 Remote Method Invocation

Remote Method Invocation (RMI) is an object-oriented implementation of the Remote Procedure Call model. It is an API for Java programs only, but its relative simplicity makes this API a good starting point for students learning to use distributed objects in network applications.

Using RMI, an object server exports a remote object and registers it with a directory service. The object provides remote methods, which can be invoked in client programs.

Syntactically, a remote object is declared with a **remote interface**, an extension of the Java interface. The remote interface is implemented by the object server. An object client accesses the object by invoking its methods, using syntax similar to local method invocations.

In the rest of this chapter we will explore in detail the Java RMI API.

7.6 The Java RMI Architecture

Figure 7.5 illustrates the architecture of the Java RMI API. As with RPC APIs, Java RMI architecture calls for proxy software modules to provide the run-time support needed to transform the remote method invocations to local method calls, and to handle the details for the underlying interprocess communications. In this architecture, three abstraction layers are present on both the client side and the server side. We will look at the two sides separately.

Client-Side Architecture

1. The **Stub layer**. A client process's remote method invocation is directed to a proxy object, known as a **stub**. The stub layer lies beneath the application layer and serves to intercept remote method invocations made by the client program; then it forwards them to the next layer below, the Remote Reference Layer.
2. The **Remote Reference layer** interprets and manages references made from clients to the remote service objects and issues the IPC operations to the next layer, the transport layer, to transmit the method calls to the remote host.
3. The **Transport layer** is TCP based and therefore connection-oriented. This layer and the rest of the network architecture carry out the IPC, transmitting the data representing the method call to the remote host.

Server-Side Architecture

Skeleton, the server-side proxy, is "deprecated" (outdated) since Java 1.2. Its functionalities are replaced by the use of a technique known as "reflection." For our discussion, we will continue to include the skeleton in the architecture as a conceptual presence.

Conceptually, the server-side architecture also involves three abstraction layers, although the implementation varies depending on the Java release.

1. The **Skeleton layer** lies just below the application layer and serves to interact with the stub layer on the client side. Quoting from [java.sun.com/products, 7],

   "The skeleton carries on a conversation with the stub; it reads the parameters for the method call from the link, makes the call to the remote service implementation object, accepts the return value, and then writes the return value back to the stub."

2. The **Remote Reference layer**. This layer manages and transforms the remote reference originating from the client to local references that are understandable to the Skeleton layer.
3. The **Transport layer**. As with client-side architecture, this layer is the connection-oriented transport layer, that is, the TCP in the TCP/IP network architecture.

Object Registry

The RMI API makes it possible for a number of directory services to be used for registering a distributed object. One such directory service is the **Java Naming and Directory Interface** (JNDI), which is more general than the RMI registry that we will use in this chapter in the sense that it can be used by applications that do not use the RMI API. The RMI registry, *rmiregistry*, a simple directory service, is provided with the Java Software Development Kit (SDK). The RMI Registry is a service whose server, when active, runs on the **object server's host machine**, by convention and by default on the TCP port 1099.

The Java SDK is what you download to your machine to obtain the use of the Java class libraries and tools such as the java compiler *javac*.

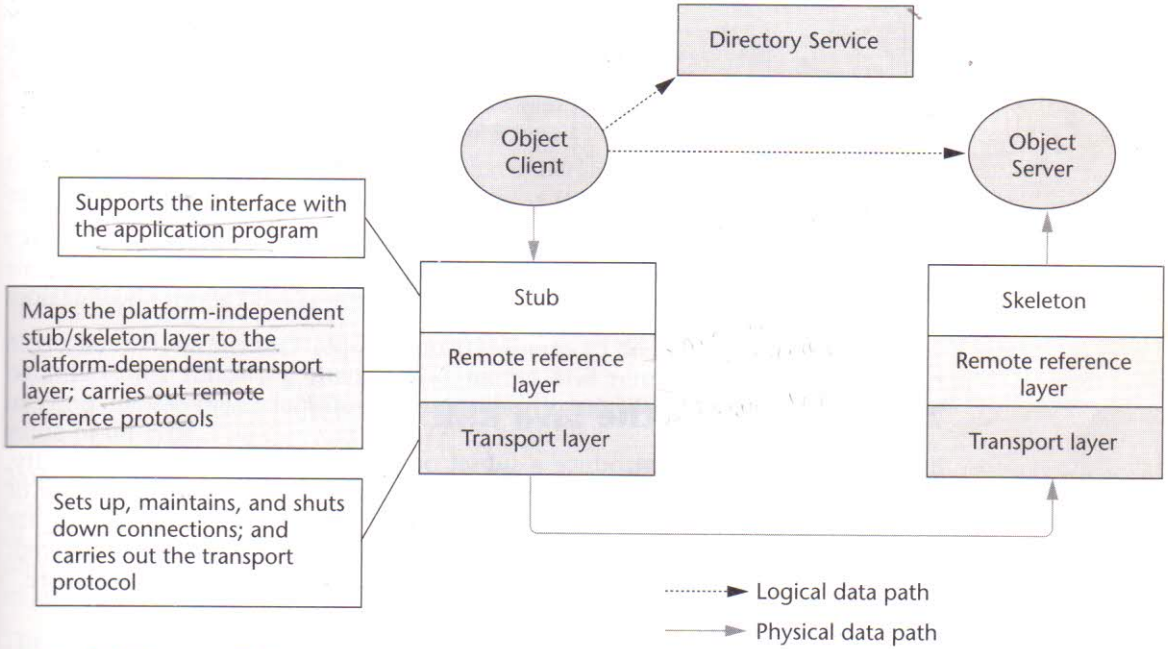


Figure 7.5 The Java RMI architecture.

Logically, from the point of view of the software developer, the remote method invocations issued in a client program interact directly with the remote objects in a server program, in the same manner that a local method call interacts with a local object. Physically, the remote method invocations are transformed to

calls to the stubs and skeletons at run time, resulting in data transmission across the network link. Figure 7.6 is a time-event diagram describing the interaction between the stub and the skeleton.

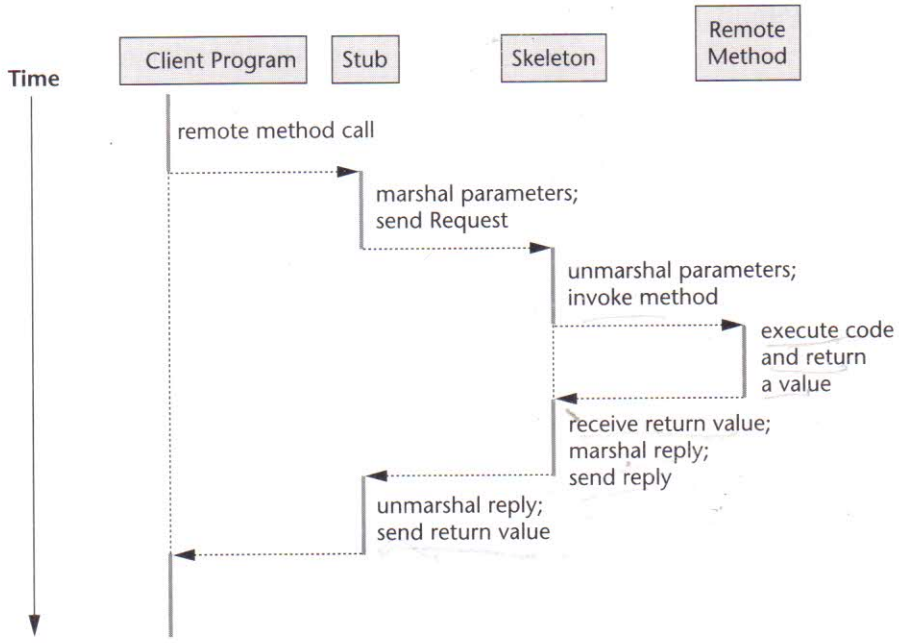


Figure 7.6 Interactions between the RMI stub and the RMI skeleton (based on a diagram in [java.sun.com/docs, 8]).

7.7 The API for the Java RMI

In this section we will introduce a subset of the Java RMI API. (For simplicity, the presentation in this chapter does not cover **security managers**, the use of which is highly recommended for all RMI applications. Security managers are covered in section 8.3 of the next chapter.) There are three areas to be covered: the **remote interface**, the **server-side software**, and the **client-side software**.

The Remote Interface

In the RMI API, the starting point of creating a distributed object is a Java **remote interface**. A Java interface is a class that serves as a template for other classes: It contains declarations, or signatures, of methods whose implementations are to be supplied by classes that realize the interface.

A Java remote interface is an interface that inherits from the Java ***Remote*** class, which allows the interface to be implemented using RMI syntax. Other than the Remote extension and the RemoteException that must be specified with each method signature, a remote interface has the same syntax as a regular or local Java interface. A sample remote interface illustrating the basic syntax is presented in Figure 7.7.

Figure 7.7 A sample Java remote interface.

```
1  // file: SomeInterface.java
2  // to be implemented by a Java RMI server class.
3
4  import java.rmi.*
5
6  public interface SomeInterface extends Remote {
7    // signature of first remote method
8    public String someMethod1( )
9      throws java.rmi.RemoteException;
10   // signature of second remote method
11   public int someMethod2( float someParameter)
12     throws java.rmi.RemoteException;
13   // signature of other remote methods may follow
14 } // end interface
```

In this example, an interface called *SomeInterface* is declared. The interface extends the Java *Remote* class (line 6), making it a remote interface.

Contained within the block enclosed between curly braces (lines 6–14) are the signatures for two **remote methods**, here named *someMethod1* (lines 8–9) and *someMethod2* (lines 11–12), respectively.

As declared, *someMethod1* requires no arguments to be passed (hence the empty parameter list following the method name) and returns a *String* object. The method *someMethod2* requires an argument value of *float* type and returns a value of *int* type.

Note that a serializable object, such as a *String* object or an object of another class, can be an argument, or it can be returned by the remote method. A copy of the item (specifically, a deep copy of the object), be it a primitive type or an object, is passed to the remote method. A return value is handled likewise, but in the opposite direction.

A serializable object is an object of a class that is serializable, so that the object can be data marshaled for transmission over a network.

The *java.rmi.Remote Exception* must be listed in the *throws* clause (lines 9 and 12) of each method signature. An exception of this type is raised when errors occur during the processing of a Remote Method Invocation, and the exception is required to be handled in the method caller's program. Causes of such exceptions include errors that may occur during interprocess communications, such as access failures and connection failures, as well as problems unique to remote method invocations, including errors resulting from the object, the stub, or the skeleton not being found.

The Server-Side Software

An object server is an object that provides the methods of and the interface to a distributed object. Each object server must (1) implement each of the remote methods specified in the interface, and (2) register an object that contains the implementation with a directory service. It is recommended that the two parts be provided as separate classes, as illustrated below.

The Remote Interface Implementation

A class that implements the remote interface should be provided. The syntax is similar to a class that implements a local interface. Figure 7.8 shows a template of the implementation.

Figure 7.8 Sample syntax for a remote interface implementation.

```
1  import java.rmi.*;
2  import java.rmi.server.*;
3
4  /**
5  * This class implements the remote interface SomeInterface.
6  */
7
8  public class SomeImpl extends UnicastRemoteObject
9    implements SomeInterface {
10
11   public SomeImpl( ) throws RemoteException {
12     super( );
13   }
14
15   public String someMethod1( ) throws RemoteException {
16     // code to be supplied
17   }
18
19   public int someMethod2( ) throws RemoteException {
20     // code to be supplied
21   }
22
23 } //end class
```

The import statements (lines 1–2) are needed for the *UnicastRemoteObject* and the *RemoteException* classes referred to in the code.

The heading of the class (line 8) must specify that (1) it is a subclass of the Java *UnicastRemoteObject* class, and (2) it implements a specific remote interface, named *SomeInterface* in the template. (*Note*: A *UnicastRemoteObject* supports unicast RMI, that is, RMI using unicast IPC. Presumably, a *MulticastRemoteObject* class can also be made available, which supports RMI using multicast IPC.)

A constructor for the class (lines 11–13) should be defined. The first line of the code should be a statement (a *super( )* call) that invokes the base class's constructor. Additional code may appear in the constructor, if needed.

The definition of each remote method should then follow (lines 15–21). The heading of each method should match its signature in the interface file.

Figure 7.9 is a UML diagram for the *SomeImpl* class.

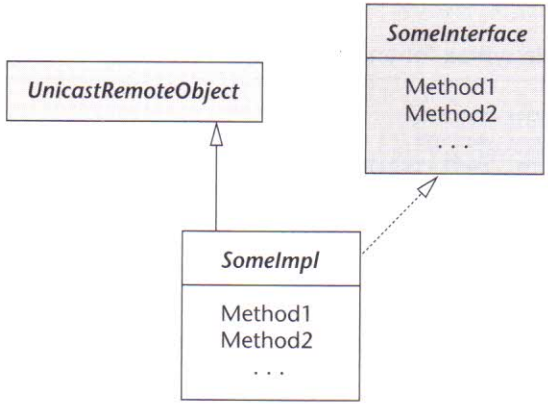


Figure 7.9 The UML class diagram for *SomeImpl*.

Stub and Skeleton Generations

In RMI, a distributed object requires a proxy each for the object server and the object client, known as the object's skeleton and stub, respectively. These proxies are generated from the implementation of a remote interface using a tool provided with the Java SDK: the RMI compiler ***rmic***. To invoke this tool, the command, which can be issued at either a UNIX system prompt or a Windows command prompt, is as follows:

```
rmic <class name of the remote interface implementation>
```

For example:

```
rmic SomeImpl
```

If the compilation is successful, two proxy files will be generated, each prefixed with the implementation class name *SomeImpl_skel.class* and *SomeImpl_stub.class*, for example.

The stub file for the object, as well as the remote interface file, must be shared with each object client: These files are required for the client program to compile. A copy of each file may be provided to the object client side manually (that is, by placing a copy of the file in an appropriate directory on the client side).

In addition, the Java RMI has a feature called **stub downloading**, which allows a stub file to be obtained by a client dynamically. Stub downloading will be covered in Chapter 8, where we will investigate some advanced topics in RMI.

The Object Server

The object server class instantiates and exports an object of the remote interface implementation. Figure 7.10 shows a template for the object server class.

Figure 7.10 Sample syntax for an object server.

```
1  import java.rmi.*;
2  import java.rmi.server.*;
3  import java.rmi.registry.Registry;
4  import java.rmi.registry.LocateRegistry;
5  import java.net.*;
6  import java.io.*;
7
8  /**
9  * This class represents the object server for a distributed
10 * object of class SomeImpl, which implements the remote
11 * interface SomeInterface.
12 */
13
14 public class SomeServer {
15   public static void main(String args[]) {
16     String portNum, registryURL;
17     try{
18       // code for port number value to be supplied
19       SomeImpl exportedObj = new SomeImpl( );
20       startRegistry(RMIPortNum);
21       // register the object under the name "some"
22       registryURL = "rmi://localhost:" + portNum + "/some";
23       Naming.rebind(registryURL, exportedObj);
24       System.out.println("Some Server ready.");
25     }// end try
26     catch (Exception re) {
27       System.out.println(
28       "Exception in SomeServer.main: " + re);
29     } // end catch
30   } // end main
31
32   // This method starts a RMI registry on the local host, if it
33   // does not already exist at the specified port number.
34   private static void startRegistry(int RMIPortNum)
35       throws RemoteException{
36     try {
37       Registry registry = LocateRegistry.getRegistry(RMIPortNum);
38       registry.list( );
39       // The above call will throw an exception
40       // if the registry does not already exist
41     }
```

(continued on next page)

```
42      catch (RemoteException ex) {
43        // No valid registry at that port.
44        System.out.println(
45          "RMI registry cannot be located at port "
46          + RMIPortNum);
47        Registry registry =LocateRegistry.createRegistry(RMIPortNum);
48        System.out.println(
49          "RMI registry created at port " + RMIPortNum);
50      } // end catch
51    } // end startRegistry
52
53  } // end class
```

In the following paragraphs we will look at the various parts of this template.

Creating an Object of the Remote Interface Implementation On line 19, an object of the **implementation** class of the remote interface is created; the reference to this object will subsequently be **exported**.

Exporting the Object Lines 20–23 in the template code export the object. To export the object, its reference must be registered with a directory service. As has already been mentioned, in this chapter we will use the *rmiregistry* service provided with the Java SDK. An *rmiregistry* server overseeing an RMI registry must be running on the object server host to provide this functionality.

Each RMI registry maintains a list of exported objects and supports an interface for looking up these objects. A registry may be shared by all object servers running on the same host. Alternatively, an individual server process may create and use its own registry if desired, in which case multiple *rmiregistry* servers may run at different port numbers on the host, each overseeing a separate list of exported objects.

On a production system, there should be an *rmiregistry* server running at all times, presumably at default port number 1099. For our code samples, we will not assume that the RMI Registry is always available, but we will instead make provisions in our code to start a copy of the server on demand and at a port of your choosing, so that each student may use a separate copy of the registry in his/her experiment in order to avoid name collisions.

A name collision occurs if an attempt is made to export an object under a name that coincides with the name of an object already in the registry.

In our object server template, the static method *startRegistry( )* (lines 34–51) is provided in the program to start up an RMI Registry server, if it is not currently running, at a user specified port number (line 20):

```
startRegistry(RMIPortNum);
```

In a production system where the default RMI registry server is used and where it can be assumed to be constantly running, the *startRegistry* call—and hence the *startRegistry* method itself—can be omitted.

In our object server template, the code for exporting an object (lines 22–23) is as follows:

```
// register the object under the name "some"
registryURL = "rmi://localhost:" + portNum + "/some";
Naming.rebind(registryURL, exportedObj);
```

The *Naming* class provides methods for storing and obtaining references from the registry. In particular, the *rebind* method allows an object reference to be stored in the registry with a URL in the form of

```
rmi://<host name>:<port number>/<reference name>
```

The *rebind* method will overwrite any reference in the registry bound with the given reference name. If the overwriting is not desirable, there is also a *bind* method.

The host name should be the name of the server, or simply "localhost.". The reference name is a name of your choice and should be unique in the registry.

The sample code first checks to see if an RMI registry is currently running at the default port. If not, an RMI registry is activated.

Alternatively, an RMI registry can be activated by hand using the *rmiregistry* utility, which comes with the Java Software Development Kit (SDK), by entering the following command at the system prompt:

```
rmiregistry <port number>
```

where the port number is a TCP port number. If no port number is specified, port number 1099 is assumed.

When an object server is executed, the exporting of the distributed object causes the server process to begin to listen and wait for clients to connect and request the service of the object. An RMI object server is a concurrent server: Each request from an object client is serviced using a separate thread of the server. Since invocations of remote methods may be executed concurrently, it is important that the implementation of a remote object is thread-safe. Readers may wish to review the discussion on "Concurrent Programming" in section 1.5 in Chapter 1.

The Client-Side Software

The program for the client class is like any other Java class. The syntax needed for RMI involves locating the RMI registry in the server host and looking up the remote reference for the server object; the reference then can be cast to the remote interface class and the remote methods invoked. A template for an object server is presented in Figure 7.11.

The Import Statements The import statements (lines 1–4) are needed in the code in order for the program to compile.

Figure 7.11 Template for an object client.

```
 1  import java.io.*;
 2  import java.rmi.*;
 3  import java.rmi.registry.Registry;
 4  import java.rmi.registry.LocateRegistry;
 5
 6  /**
 7  * This class represents the object client for a distributed
 8  * object of class SomeImpl, which implements the remote
 9  * interface SomeInterface.
10  */
11
12  public class SomeClient {
13    public static void main(String args[ ]) {
14      try {
15        int RMIPort;
16        String hostName;
17        String portNum;
18        // Code for obtaining hostName and RMI Registry port
19        // to be supplied.
20
21        // Look up the remote object and cast its reference
22        // to the remote interface class--replace "localHost" with
23        // the appropriate host name of the remote object.
24        String registryURL =
25          "rmi://localhost:" + portNum + "/some";
26        SomeInterface h =
27          (SomeInterface)Naming.lookup(registryURL);
28        // invoke the remote method(s)
29        String message = h.method1( );
30        System.out.println(message);
31        // method2 can be invoked similarly
32      } // end try
33      catch (Exception ex) {
34        ex.printStackTrace( );
35      } // end catch
36    } // end main
37    // Definition for other methods of the class, if any.
38  } // end class
```

Looking Up the Remote Object The code on lines 24–27 are for looking up the remote object in the registry. The *lookup* method of the *Naming* class is used to retrieve the object reference, if any, previously stored in the registry by the object server. Note that the retrieved reference should be cast to the remote interface (*not* its implementation) class.

```
String registryURL = "rmi://localhost:" + portNum + "/some";
SomeInterface h = (SomeInterface)Naming.lookup(registryURL);
```

Invoking the Remote Method The remote interface reference can be used to invoke any of the methods in the remote interface, as on lines 29–30 in the example:

```
String message = h.method1( );
System.out.println(message);
```

Note that the syntax for the invocation of the remote methods is the same as for local methods.

7.8 A Sample RMI Application

Figures 7.12 through 7.15 comprise the complete listing of a sample RMI application, Hello. The server exports an object that contains a single remote method, *sayHello*. As you study the code, try to identify the different parts we discussed in the previous section.

Figure 7.12 *HelloInterface.java.*

```
 1  // A simple RMI interface file - M. Liu
 2  import java.rmi.*;
 3
 4  /**
 5  * This is a remote interface.
 6  * @author M. L. Liu
 7  */
 8
 9  public interface HelloInterface extends Remote {
10  /**
11  * This remote method returns a message.
12  * @para name - a string containing a name.
13  * @return a String message.
14  */
15    public String sayHello(String name)
16      throws java.rmi.RemoteException;
17
18    }
```

Figure 7.13 *HelloImpl.java.*

```
1  import java.rmi.*;
2  import java.rmi.server.*;
3
4  /**
5  * This class implements the remote interface
6  * HelloInterface.
7  * @author M. L. Liu
8  */
9
10 public class HelloImpl extends UnicastRemoteObject
11 implements HelloInterface {
12
13   public HelloImpl() throws RemoteException {
14     super( );
15   }
16
17   public String sayHello(String name)
18       throws RemoteException {
19     return "Hello, World!" + name;
20   }
21 } // end class
```

Figure 7.14 *HelloServer.java.*

```
1  import java.rmi.*;
2  import java.rmi.server.*;
3  import java.rmi.registry.Registry;
4  import java.rmi.registry.LocateRegistry;
5  import java.net.*;
6  import java.io.*;
7
8  /**
9  *  This class represents the object server for a distributed
10 *  object of class Hello, which implements the remote interface
11 *  HelloInterface.
12 *  @author M. L. Liu
13 */
14
15 public class HelloServer {
16   public static void main(String args[ ]) {
17     InputStreamReader is = new InputStreamReader(System.in);
18     BufferedReader br = new BufferedReader(is);
19     String portNum, registryURL;
```

(continued on next page)

```
20      try{
21        System.out.println("Enter the RMIregistry port number:");
22        portNum = (br.readLine()).trim();
23        int RMIPortNum = Integer.parseInt(portNum);
24        startRegistry(RMIPortNum);
25        HelloImpl exportedObj = new HelloImpl();
26        registryURL = "rmi://localhost:" + portNum + "/hello";
27        Naming.rebind(registryURL, exportedObj);
28  /**/  System.out.println
29  /**/  ("Server registered. Registry currently contains:");
30  /**/  // list names currently in the registry
31  /**/  listRegistry(registryURL);
32        System.out.println("Hello Server ready.");
33      }// end try
34      catch (Exception re) {
35        System.out.println("Exception in HelloServer.main: " + re);
36      } // end catch
37    } // end main
38
39    // This method starts an RMI registry on the local host, if it
40    // does not already exist at the specified port number.
41    private static void startRegistry(int RMIPortNum)
42        throws RemoteException{
43      try {
44        Registry registry = LocateRegistry.getRegistry(RMIPortNum);
45        registry.list( ); // This call will throw an exception
46        // if the registry does not already exist
47      }
48      catch (RemoteException e) {
49        // No valid registry at that port.
50  /**/  System.out.println
51  /**/    ("RMI registry cannot be located at port "
52  /**/    + RMIPortNum);
53        Registry registry =
54          LocateRegistry.createRegistry(RMIPortNum);
55  /**/  System.out.println(
56  /**/    "RMI registry created at port " + RMIPortNum);
57      } // end catch
58    } // end startRegistry
59
60    // This method lists the names registered with a Registry object
61    private static void listRegistry(String registryURL)
62        throws RemoteException, MalformedURLException {
63      System.out.println("Registry " + registryURL + " contains: ")
64      String [ ] names = Naming.list(registryURL);
65      for (int i=0; i < names.length; i++)
66        System.out.println(names[i]);
67    } // end listRegistry
68
69  } // end class
```

Figure 7.15 *HelloClient.java.*

```
1  import java.io.*;
2  import java.rmi.*;
3
4  /**
5  * This class represents the object client for a distributed
6  * object of class Hello, which implements the remote interface
7  * HelloInterface.
8  * @author M. L. Liu
9  */
10
11 public class HelloClient {
12
13   public static void main(String args[ ]) {
14     try {
15       int RMIPort;
16       String hostName;
17       InputStreamReader is = new InputStreamReader(System.in);
18       BufferedReader br = new BufferedReader(is);
19       System.out.println("Enter the RMIRegistry host namer:");
20       hostName = br.readLine();
21       System.out.println("Enter the RMIregistry port number:");
22       String portNum = br.readLine();
23       RMIPort = Integer.parseInt(portNum);
24       String registryURL =
25         "rmi://" + hostName+ ":" + portNum + "/hello";
26       // find the remote object and cast it to an interface object
27       HelloInterface h =
28         (HelloInterface)Naming.lookup(registryURL);
29       System.out.println("Lookup completed " );
30       // invoke the remote method
31       String message = h.sayHello("Donald Duck");
32       System.out.println("HelloClient: " + message);
33     } // end try
34     catch (Exception e) {
35       System.out.println("Exception in HelloClient: " + e);
36     } // end catch
37   } // end main
38 } // end class
```

Once you understand the basic structure of the sample RMI application we have just presented, you should be able to use the syntax in the template to build any RMI application by replacing the presentation and application logic; the service logic (using RMI) is invariant.

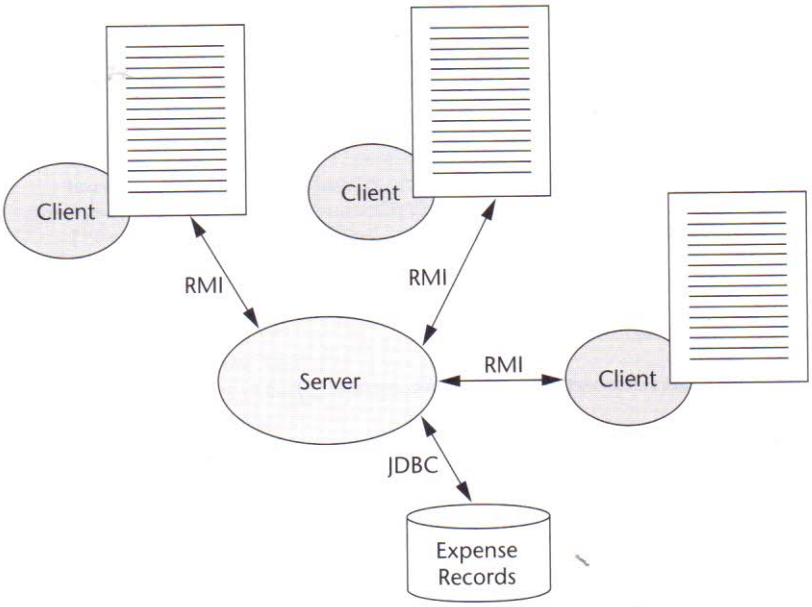


Figure 7.16 A sample RMI application.

The RMI technology is a good candidate for a software component at the service layer. A sample industrial application is the expense report system for an enterprise, shown in Figure 7.16 and described in [java.sun.com/marketing, 8]. In the illustrated application, the object server provides remote methods that allow the object clients to look up or update the data in an expense records database. Programs that are clients of the object provide the application or business logic for processing the data and the presentation logic for the user interface.

The Java RMI facility is rich in features. This chapter has presented a very basic subset of those features, as an illustration of a distributed object system. Some of the more interesting advanced features of RMI will be introduced in the next chapter.

7.9 Steps for Building an RMI Application

Having looked at the various aspects of the RMI API, we now conclude with a description of the step-by-step procedure for building an RMI application so that you can experiment with the paradigm. We describe the algorithm for doing so on both sides of the object server and the object client. Keep in mind that in a production environment it is likely that the development of software on the two sides may proceed independently.

The algorithm is phrased in terms of an application named *Some*. The steps will apply to any application by replacing the name *Some* with the name of the application.

Algorithm for Developing the Server-Side Software

1. Open a directory for all the files to be generated for this application.
2. Specify the remote server interface in *SomeInterface.java*. Compile and revise it until there is no more syntax error.
3. Implement the interface in *SomeImpl.java*. Compile and revise it until there is no more syntax error.
4. Use the RMI compiler ***rmic*** to process the implementation class and generate the stub file and skelton file for the remote object:

   ```
   rmic SomeImpl
   ```

   The files generated can be found in the directory as *SomeImpl_Skel.class* and *SomeImpl_Stub.class*. Steps 3 and 4 should be repeated each time a change is made to the interface implementation.
5. Create the object server program *SomeServer.java*. Compile and revise it until there is no more syntax error.
6. Activate the object server

   ```
   java SomeServer
   ```

Algorithm for Developing the Client-Side Software

1. Open a directory for all the files to be generated for this application.
2. Obtain a copy of the remote interface class file. Alternatively, obtain a copy of the source file for the remote interface, and compile it using ***javac*** to generate the interface class file.
3. Obtain a copy of the stub file for the implementation of the interface *SomeImpl_Stub.class*.
4. Develop the client program *SomeClient.java* and compile it to generate the client class.
5. Activate the client.

   ```
   java SomeClient
   ```

Figure 7.17 illustrates the placement of the various files for an application on the client side and the server side. The remote interface class and the stub class files for each remote object must be present on the object client host, along with the object client class. On the server side are the interface class, the object server class, the interface implementation class, and the stub class for the remote object.

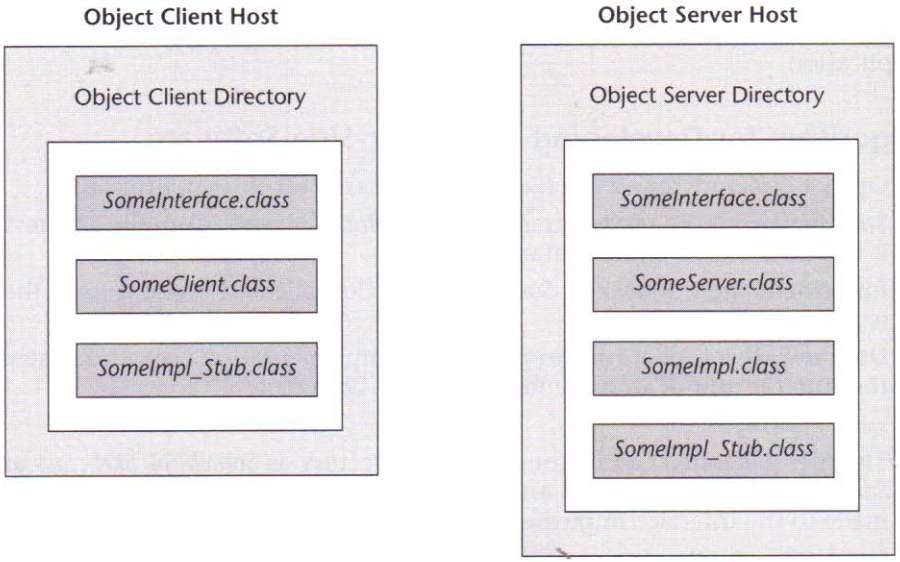


Figure 7.17 Placement of files for an RMI application.

7.10 Testing and Debugging

As with any form of network programming, the tasks of testing and debugging concurrent processes are nontrivial. It is recommended that you adhere to the following incremental steps in developing an RMI application:

1. Build a template for a minimal RMI program. Start with a remote interface containing a single method signature, its implementation using a stub, a server program that exports the object, and a client program with just enough code that invokes the remote method. Test the template programs on one host until the remote method can be made successfully.
2. Add one signature at a time to the interface. With each addition, modify the client program to invoke the added method.
3. Fill in the definition of each remote method, one at a time. Test and thoroughly debug each newly added method before proceeding with the next one.
4. After all remote methods have been thoroughly tested, develop the client application using an incremental approach. With each increment, test and debug the programs.
5. Distribute the programs on separate machines. Test and debug.

7.11 Comparison of RMI and Socket APIs

The Remote Method Invocation API, as representative of the distributed object paradigm, is an efficient tool for building network applications. It can be used in lieu of the socket API (representing the message-passing paradigm) to build a network application rapidly. However, you should be aware that the gain in convenience is not without its drawbacks.

Some of the trade-offs between the RMI API and the socket API are enumerated below:

- The socket API works closely with the operating system and hence has less execution overhead. RMI requires additional software support, including the proxies and the directory service, which inevitably incur run-time overhead. For applications that require high performance, the socket API may remain the only viable solution.
- On the other hand, the RMI API provides the abstraction that eases the task of software development. Programs developed with a higher level of abstraction are more comprehensible and hence easier to debug.
- Because it operates at a lower layer, a socket API is typically platform and language independent. The same may not be true with RMI. The Java RMI, for example, requires Java-specific run-time supports. As a result, an application implemented using Java RMI must be written in Java and can only run on Java platforms.

The choice of an appropriate paradigm and an appropriate API is a key decision in the design of an application. Depending on the circumstances, it is possible for one paradigm or API to be used in some parts of an application, while another paradigm or API is used elsewhere in the application.

Because of the relative ease with which network applications can be developed using RMI, RMI is a good candidate for the rapid development of a prototype for an application.

In software engineering, a prototype is an initial version put together quickly to demonstrate the user interface and functionalities of a proposed application.

7.12 Food for Thought

The model of distributed object-oriented computing presented in this chapter is based on the vision that, from the programmer's point of view, there is no essential distinction between local objects and remote objects. Although this vision is widely accepted as the basis of distributed object systems, it is not without its detractors. The authors of [research.sun.com, 11], for one, argue that this vision, though convenient, is inappropriate because it ignores the inherent differences between local and remote objects. Exercise 11 at the end of the chapter asks you to look into this argument.

Summary

This chapter introduced you to the distributed object paradigm. Following is a summary of the key points:

- The distributed object paradigm is at a higher level of abstraction than the message-passing paradigm.
- Using the paradigm, a process invokes methods of a remote object, passing in data as arguments and receiving a return value with each call, using syntax similar to local method calls.
- In a distributed object system, an object server provides a distributed object whose methods can be invoked by an object client. Each side requires a proxy that interacts with the system's run-time support to perform the necessary IPC. Additionally, an object registry must be available to enable distributed objects to be registered and looked up.
- Among the best-known distributed object system protocols are the Java Remote Method Invocation (RMI), the Distributed Component Object Model (DCOM), the Common Object Request Broker Architecture (CORBA), and the Simple Object Access Protocol (SOAP).
- Java RMI is representative of distributed object systems. Some of the topics related to RMI covered are:
  - The architecture of the Java remote method invocation API includes three architecture layers on both the client side and the server side. On the client side, the stub layer accepts a remote method invocation and transforms it into messages to the server side. On the server side, the skeleton layer receives the messages and transforms them to a local method call to the remote method. A directory service such as JNDI or the RMI Registry can be used for the object registry.
  - The software for an RMI application includes a remote interface, server-side software, and client-side software. The syntax and recommended algorithms for developing the software were presented.
- Some trade-offs between the socket API and the Java RMI API were discussed.

Exercises

1. Compare and contrast the message-passing paradigm with the distributed object paradigm.
2. Compare and contrast a local procedure call with a remote procedure call.
3. Describe the Java RMI architecture. What is the role of the RMI Registry?
4. Consider a simple application where a client sends two integer values to a server, which sums the values and returns the sum to the client.
   a. Describe how you would implement the application using the socket API. Describe the messages exchanged and the actions triggered by each message.
   b. Describe how you would implement the application using the RMI API. Describe the interface, the remote methods, and the remote method invocations in the client program.
5. This exercise makes use of the *Hello* example.
   a. Open a directory for this exercise. Place the source files for the *Hello* example in the directory.
   b. Compile *HelloInterface.java* and *HelloImpl.java*.
   c. Use *rmic* to compile *HelloImpl*. Check the folder to see that the proxy classes are generated. What are their names?
   d. Compile *HelloServer.java*. Check the contents of your folder.
   e. Run the server, specifying a random port number for the RMI Registry. Check the messages displayed, including the list of names currently in the registry. Do you see the name under which the server registered the remote object (as specified in the program)?
   f. Compile and run *HelloClient.java*. When prompted, specify "localhost" for the host name and the RMI registry port number you previously specified. What happened? Explain.
   g. Run the client program on a separate machine. Were you successful?
6. Open a new folder. Copy all the source files of the *Hello* example to the folder. Add code in the *sayHello* method of *HelloImpl.java* so that there is a 5-second delay before the method returns. This has the effect of artificially lengthening the latency for each invocation of the method. Compile and start the server.

   In separate screens, start two or more clients. Observe the sequence of events displayed on the screens. Can you tell if the method calls are executed by the object server concurrently or iteratively? Explain.

7. Open a new folder. Copy all the source files of the *Hello* example to the folder. Modify the *sayHello* method so that an argument, a name string, is passed in as an argument, and the return string is the string "Hello!" concatenated with the name string.
   a. Show the code modifications.
   b. Recompile and run the server and then the client. Describe and explain what happened.
   c. Run *rmic* again to generate the new proxies for the modified interface, and then run the server and the client.

   Hand in the source listings and a script of the run outcomes.

8. Use RMI to implement a *Daytime* server and client suite.

9. Using RMI, write an application for a prototype opinion poll system. Assume that only one issue is being polled. Respondents may choose *yes, no, or don't care*. Write a server application to accept the votes, keep the tally (in transient memory), and provide the current counts to those who are interested.
   a. Write the interface file first. It should provide remote methods for accepting a response to the poll, providing the current counts (e.g., "10 yes, 2 no, 5 don't care") only when the client requests it.
   b. Design and implement a server to (i) export the remote methods, and (ii) maintain the state information (the counts). Note that the updates of the counts should be protected with mutual exclusion.
   c. Design and implement a client application to provide a user interface for accepting a response and/or a request, and to interact with the server appropriately via remote method invocations.
   d. Test your application by running two or more clients on different machines (preferably on different platforms).
   e. Hand in listings of your files, which should include your source files (the interface file, the server file(s), the client file(s)), and a README file that explains the contents and interrelationship of your source files and the procedure for running your work.

10. Create an RMI application for conducting an election. The server exports two methods:
   - *castVote*, which accepts as a parameter a string containing a candidate's name (Gore or Bush), and returns nothing, and
   - *getResult*, which returns, in an *int* array, the current count for each candidate.

   Test your application by running all processes on one machine. Then test your application by running the client and server on *separate* machines.

   Turn in the source code for the remote interface, the server, and the client.

11. Read reference [research.sun.com, 11]. Summarize the reasons why the authors find fault with the distributed object model, which minimizes the programming differences between local and remote object invocations. Do you agree with the authors? What might be an alternative model for distributed objects that addresses the authors' complaints? (*Hint*: Look into the Jini Network Technology [sun.com, 12].)

References

1. Object Management Group. Welcome to the OMG's CORBA Website, *http://www.corba.org/*
2. World Wide Web Consortium. SOAP Version 1.2 Part 0: Primer, *http://www.w3.org/TR/soap12-part0/*
3. Distributed Component Object Model (DCOM)—Downloads, Specifications, Samples, Papers, and Resources for Microsoft DCOM, *http://www.microsoft.com/com/tech/DCOM.asp*, Microsoft.
4. Richard T. Grimes. *Professional DCOM Programming*. Chicago, IL: Wrox Press, Inc., 1997.
5. RFC 1831: Remote Procedure Call Protocol Specification Version 2, August 1995, *http://www.ietf.org/rfc/rfc1831.txt*
6. The Open Group, DCE 1.1: Remote Procedure Call, *http://www.opennc.org/public/pubs/catalog/c706.htm*
7. Java™ Remote Method Invocation, *http://java.sun.com/products/jdk/rmi*
8. RMI—The Java Tutorial, *http://java.sun.com/docs/books/tutorial/rmi/*
9. Introduction to Distributed Computing with RMI, *http://developer.java.sun.com/developer/onlineTraining/rmi/RMI.html*
10. Java Remote Method Invocation—Distributed Computing for Java, *http://java.sun.com/marketing/collateral/javarmi.html*
11. Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A Note on Distributed Computing, Report TR-94-29, Sun Microsystems Laboratories, 1994, *http://research.sun.com/research/techrep/1994/smli_tr-94-29.pdf*
12. Jini Network Technology, An Executive Overview, white paper. Sun Microsystems, Inc., 2001, *http://www.sun.com/software/jini/whitepapers/jini-execoverview.pdf*