Table of Contents

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Chapter 18. Remote Method Invocation

Historically, networking has been concerned with two fundamental applications. The first application is moving files and data between hosts and is handled by FTP, SMTP, HTTP, NFS, IMAP, POP, and many other protocols. The second application is allowing one host to run programs on another host. This is the traditional province of Telnet, rlogin, Remote Procedure Call (RPC), and a lot of database middleware. Most of this book has implicitly concerned itself with file and data transfer. Remote Method Invocation (RMI), however, is an example of the second application for networking: running a program on a remote host from a local machine.

RMI is a core Java API and class library that allows Java programs running in one Java virtual machine to call methods in objects running in a different virtual machine, even when the two virtual machines are running on physically separate hosts. In essence, parts of a single Java program run on a local computer while other parts of the same program run on a remote host. RMI creates the illusion that this distributed program is running on one system with one memory space holding all the code and data used on either side of the actual physical connection.

18.1. What Is Remote Method Invocation?

RMI lets Java objects on different hosts communicate with each other in a way that's similar to how objects running in the same virtual machine communicate with each other: by calling methods in objects. A remote object lives on a server. Each remote object implements a remote interface that specifies which of its methods can be invoked by clients. Clients invoke the methods of the remote object almost exactly as they invoke local methods. For example, an object running on a local client can pass a database query as a `String` argument to a method in a database object running on a remote server to ask it to sum up a series of records. The server can return the result to the client as a `double`. This is more efficient than downloading all the records and summing them up locally. Java-compatible web servers can implement remote methods that allow clients to ask for a complete index of the public files on the site. This could dramatically reduce the time a server spends filling

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

requests from web spiders such as Google. Indeed, Excite already uses a non-Java-based version of this idea.

From the programmer's perspective, remote objects and methods work pretty much like the local objects and methods you're accustomed to. All the implementation details are hidden. You just import one package, look up the remote object in a registry (which takes one line of code), and make sure that you catch `RemoteException` when you call the object's methods. From that point on, you can use the remote object almost as freely and easily as you use an object running on your own system. The abstraction is not perfect. Remote method invocation is much slower and less reliable than regular local method invocation. Things can and do go wrong with remote method invocation that do not affect local method invocations. For instance, a local method invocation is not subject to a Verizon technician disconnecting your DSL line while working on the phone line next door. Network failures of this type are represented as `RemoteExceptions`. However, RMI tries to hide the difference between local and remote method invocation to the maximum extent possible.

More formally, a *remote object* is an object with methods that may be invoked from a different Java virtual machine than the one in which the object itself lives, generally one running on a different computer. Each remote object implements one or more *remote interfaces* that declare which methods of the remote object can be invoked by the foreign system. RMI is the facility by which a Java program running on one machine, say *java.oreilly.com*, can invoke a method in an object on a completely different machine, say *www.ibiblio.org*.

For example, suppose *weather.centralpark.org* is an Internet-connected PC at the Central Park weather station that monitors the temperature, humidity, pressure, wind speed and direction, and similar information through connections to various instruments, and it needs to make this data available to remote users. A Java program running on that PC can offer an interface like Example 18-1 that provides the current values of the weather data.

Example 18-1. The weather interface

```
import java.rmi.*;
import java.util.Date;

public interface Weather extends Remote {

  public double getTemperature( ) throws RemoteException;
  public double getHumidity( ) throws RemoteException;
  public double getPressure( ) throws RemoteException;
  public double getWindSpeed( ) throws RemoteException;
  public double getWindDirection( ) throws RemoteException;
  public double getLatitude( ) throws RemoteException;
  public double getLongitude( ) throws RemoteException;
  public Date   getTime( ) throws RemoteException;

}
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Normally, this interface is limited to other programs running on that same PC—indeed, in the same virtual machine. However, remote method invocations allow other virtual machines running on other computers in other parts of the world to invoke these methods to retrieve the weather data. For instance, a Java program running on my workstation at stallion.elharo.com could look up the current weather object in the RMI registry at weather.centralpark.org. The registry would send it a reference to the object running in weather.centralpark.org's virtual machine. My program could then use this reference to invoke the `getTemperature( )` method. The `getTemperature ( )` method would execute on the server in Central Park, not on my local machine. However, it would return the double value back to my local program running in Brooklyn. This is simpler than designing and implementing a new socket-based protocol for communication between the weather station and its clients. The details of making the connections between the hosts and transferring the data are hidden in the RMI classes.

So far we've imagined a public service that's accessible to all. However, clearly there are some methods you don't want just anyone invoking. More RMI applications than not will have a strictly limited set of permitted users. RMI itself does not provide any means of limiting who's allowed to access RMI servers. These capabilities can be added to RMI programs through the Java Authentication and Authorization Service (JAAS). JAAS is an abstract interface that can be configured with different service providers to support a range of different authentication schemes and different stores for the authentication data.

18.1.1. Object Serialization

When an object is passed to or returned from a Java method, what's really transferred is a reference to the object. In most current implementations of Java, references are handles (doubly indirected pointers) to the location of the object in memory. Passing objects between two machines thus raises some problems. The remote machine can't read what's in the memory of the local machine. A reference that's valid on one machine isn't meaningful on the other.

There are two ways around this problem. The first way is to convert the object to a sequence of bytes and send these bytes to the remote machine. The remote machine receives the bytes and reconstructs them into a copy of the object. However, changes to this copy are not automatically reflected in the original object. This is like pass-by-value.

The second way around this problem is to pass a special remote reference to the object. When the remote machine invokes a method on this reference, that invocation travels back across the Internet to the local machine that originally created the object. Changes made on either machine are reflected on both ends of the connection because they share the same object. This is like pass-by-reference.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Converting an object into a sequence of bytes is more difficult than it appears at first glance because object fields can be references to other objects; the objects these fields point to also need to be copied when the object is copied. And these objects may point to still other objects that also need to be copied. Object serialization is a scheme by which objects can be converted into bytes and then passed around to other machines, which rebuild the original object from the bytes. These bytes can also be written to disk and read back from disk at a later time, allowing you to save the state of an entire program or a single object.

For security reasons, Java places some limitations on which objects can be serialized. All Java primitive types can be serialized, but nonremote Java objects can be serialized only if they implement the `java.io.Serializable` interface. Basic Java types that implement `Serializable` include `String` and `Component`. Container classes such as `Vector` are serializable if all the objects they contain are serializable. Furthermore, subclasses of a serializable class are also serializable. For example, `java.lang.Integer` and `java.lang.Float` are serializable because the class they extend, `java.lang.Number`, is serializable. Exceptions, errors, and other throwable objects are always serializable. Most AWT and Swing components, containers, and events are serializable. However, event adapters, image filters, and peer classes are not. Streams, readers and writers, and most other I/O classes are not serializable. Type wrapper classes are serializable except for `Void`. Classes in `java.math` are serializable. Classes in `java.lang.reflect` are not serializable. The `URL` class is serializable. However, `Socket`, `URLConnection`, and most other classes in `java.net` are not. If in doubt, the class library documentation will tell you whether a given class is serializable.

Object serialization is discussed in much greater detail in Chapter 11 of my previous book, *Java I/O* (O'Reilly).

CORBA

RMI isn't the final word in distributed object systems. Its biggest limitation is that you can call only methods written in Java. What if you already have an application written in some other language, such as C++, and you want to communicate with it? The most general solution for distributed objects is CORBA, the Common Object Request Broker Architecture. CORBA lets objects written in different languages communicate with

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

each other. Java hooks into CORBA through the Java-IDL. This goes beyond the scope of this book; to find out about these topics, see:

- Java-IDL (http://java.sun.com/products/jdk/idl/)
- CORBA for Beginners (http://www.omg.org/gettingstarted/corbafaq.htm)
- The CORBA FAQ list (http://www4.informatik.uni-erlangen.de/~geier/corba-faq/)
- *Client/Server Programming with Java and CORBA* by Dan Harkey and Robert Orfali (Wiley)

18.1.2. Under the Hood

The last two sections skimmed over a lot of details. Fortunately, Java hides most of the details from you. However, it never hurts to understand how things really work.

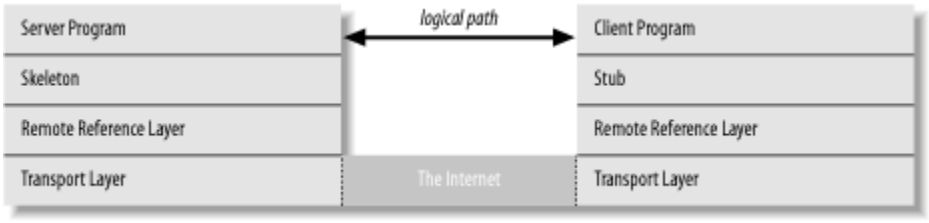
The fundamental difference between remote objects and local objects is that remote objects reside in a different virtual machine. Normally, object arguments are passed to methods and object values are returned from methods by referring to something in a particular virtual machine. This is called *passing a reference*. However, this method doesn't work when the invoking method and the invoked method aren't in the same virtual machine; for example, object 243 in one virtual machine has nothing to do with object 243 in a different virtual machine. In fact, different virtual machines may implement references in completely different and incompatible ways.

Therefore, three different mechanisms are used to pass arguments to and return results from remote methods, depending on the type of the data being passed. Primitive types (`int`, `boolean`, `double`, and so on) are passed by value, just as in local Java method invocation. References to remote objects (that is, objects that implement the `Remote` interface) are passed as *remote references* that allow the recipient to invoke methods on the remote objects. This is similar to the way local object references are passed to local Java methods. Objects that do not implement the `Remote` interface are passed by value; that is, complete copies are passed, using object serialization. Objects that do not allow themselves to be serialized cannot be passed to remote methods. Remote objects run on the server but can be called by objects running on the client. Nonremote, serializable objects run on the client system.

To make the process as transparent to the programmer as possible, communication between a remote object client and a server is implemented in a series of layers, as shown in Figure 18-1.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Figure 18-1. The RMI layer model



To the programmer, the client appears to talk directly to the server. In reality, the client program talks only to a stub object that stands in for the real object on the remote system. The stub passes that conversation along to the remote reference layer, which talks to the transport layer. The transport layer on the client passes the data across the Internet to the transport layer on the server. The server's transport layer then communicates with the server's remote reference layer, which talks to a piece of server software called the *skeleton*. The skeleton communicates with the server itself. (Servers written in Java 1.2 and later can omit the skeleton layer.) In the other direction (server-to-client), the flow is simply reversed. Logically, data flows horizontally (client-to-server and back), but the actual flow of data is vertical.

This approach may seem overly complex, but remember that most of the time you don't need to think about it, any more than you need to think about how a telephone translates your voice into a series of electrical impulses that get translated back to sound at the other end of the phone call. The goal of RMI is to allow your program to pass arguments to and return values from methods without worrying about how those arguments and return values will move across the network. At worst, you'll simply need to handle one additional kind of exception a remote method might throw.

Before you can call a method in a remote object, you need a reference to that object. To get this reference, ask a *registry* for it by name. The registry is like a mini-DNS for remote objects. A client connects to the registry and gives it the URL of the remote object that it wants. The registry replies with a reference to the object that the client can use to invoke methods on the server.

In reality, the client is only invoking local methods in a *stub*. The stub is a local object that implements the remote interfaces of the remote object; this means that the stub has methods matching the signatures of all the methods the remote object exports. In effect, the client thinks it is calling a method in the remote object, but it is really calling an equivalent method in the stub. Stubs are used in the client's virtual machine in place of the real objects and methods that live on the server; you may find it helpful to think of the stub as the remote object's surrogate on the client. When the client invokes a method, the stub passes the invocation to the remote reference layer.

The remote reference layer carries out a specific remote reference protocol, which is independent of the specific client stubs and server skeletons. The remote reference layer is responsible for understanding what a particular remote reference means. Sometimes the remote reference may refer to multiple virtual machines on multiple hosts. In other situations, the reference may refer to a single virtual machine on the local host or a virtual machine on a remote host. In essence, the remote

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

reference layer translates the local reference to the stub into a remote reference to the object on the server, whatever the syntax or semantics of the remote reference may be. Then it passes the invocation to the transport layer.

The transport layer sends the invocation across the Internet. On the server side, the transport layer listens for incoming connections. Upon receiving an invocation, the transport layer forwards it to the remote reference layer on the server. The remote reference layer converts the remote references sent by the client into references for the local virtual machine. Then it passes the request to the skeleton. The skeleton reads the arguments and passes the data to the server program, which makes the actual method call. If the method call returns a value, that value is sent down through the skeleton, remote reference, and transport layers on the server side, across the Internet and then up through the transport, remote reference, and stub layers on the client side. In Java 1.2 and later, the skeleton layer is omitted and the server talks directly to the remote reference layer. Otherwise, the protocol is the same.

18.2. Implementation

Most of the methods you need for working with remote objects are in three packages: `java.rmi`, `java.rmi.server`, and `java.rmi.registry`. The `java.rmi` package defines the classes, interfaces, and exceptions that will be seen on the client side. You need these when you're writing programs that access remote objects but are not themselves remote objects. The `java.rmi.server` package defines the classes, interfaces, and exceptions that will be visible on the server side. Use these classes when you are writing a remote object that will be called by clients. The `java.rmi.registry` package defines the classes, interfaces, and exceptions that are used to locate and name remote objects.

In this chapter and in Sun's documentation, the server side is always considered to be "remote" and the client is always considered "local". This can be confusing, particularly when you're writing a remote object. When writing a remote object, you're probably thinking from the viewpoint of the server, so that the client appears to be remote.

18.2.1. The Server Side

To create a new remote object, first define an interface that extends the `java.rmi.Remote` interface. `Remote` is a marker interface that does not have any methods of its own; its sole purpose is to tag remote objects so that they can be identified as such. One definition of a remote object is an instance of a class that implements the `Remote` interface, or any interface that extends `Remote`.

Your subinterface of `Remote` determines which methods of the remote object clients may call. A remote object may have many public methods, but only those declared in a remote interface can be invoked remotely. The other public methods may be invoked only from within the virtual machine where the object lives.

Each method in the subinterface must declare that it throws `RemoteException`. `RemoteException` is the superclass for most of the exceptions that can be thrown when RMI is used. Many of these are related to the behavior of external systems and networks and are thus beyond your control.

Example 18-2 is a simple interface for a remote object that calculates Fibonacci numbers of arbitrary size. (Fibonacci numbers are the sequence that begins 1, 1, 2, 3, 5, 8, 13 . . . in which each number is the sum of the previous two.) This remote object can run on a high-powered server to calculate results for low-powered clients. The interface declares two overloaded `getFibonacci( )` methods, one of which takes an `int` as an argument and the other of which takes a `BigInteger`. Both methods return `BigInteger` because Fibonacci numbers grow very large very quickly. A more complex remote object could have many more methods.

Example 18-2. The Fibonacci interface

```
import java.rmi.*;
import java.math.BigInteger;

public interface Fibonacci extends Remote {

  public BigInteger getFibonacci(int n) throws RemoteException;
  public BigInteger getFibonacci(BigInteger n) throws RemoteException;

}
```

Nothing in this interface says anything about how the calculation is implemented. For instance, it could be calculated directly, using the methods of the `java.math.BigInteger` class. It could be done equally easily with the more efficient methods of the `com.ibm.BigInteger` class from IBM's alphaWorks (http://www.alphaworks.ibm.com/tech/bigdecimal). It could be calculated with `int`s for small values of `n` and `BigInteger` for large values of `n`. Every calculation could be

performed immediately, or a fixed number of threads could be used to limit the load that this remote object places on the server. Calculated values could be cached for faster retrieval on future requests, either internally or in a file or database. Any or all of these are possible. The client neither knows nor cares how the server gets the result as long as it produces the correct one.

The next step is to define a class that implements this remote interface. This class should extend `java.rmi.server.UnicastRemoteObject`, either directly or indirectly (i.e., by extending another class that extends `UnicastRemoteObject`):

```
public class UnicastRemoteObject extends RemoteServer
```

Without going into too much detail, the `UnicastRemoteObject` provides a number of methods that make remote method invocation work. In particular, it marshals and unmarshals remote references to the object. (*Marshalling* is the process by which arguments and return values are converted into a stream of bytes that can be sent over the network. *Unmarshalling* is the reverse: the conversion of a stream of bytes into a group of arguments or a return value.)

If extending `UnicastRemoteObject` isn't convenient—for instance, because you'd like to extend some other class—you can instead export your object as a remote object by passing it to one of the static `UnicastRemoteObject.exportObject()` methods:

```
public static RemoteStub exportObject(Remote obj)
 throws RemoteException
public static Remote exportObject(Remote obj, int port)  // Java 1.2
 throws RemoteException
public static Remote exportObject(Remote obj, int port,  // Java 1.2
 RMIClientSocketFactory csf, RMIServerSocketFactory ssf)
 throws RemoteException
```

These create a remote object that uses your object to do the work. It's similar to how a `Runnable` object can be used to give a thread something to do when it's inconvenient to subclass `Thread`. However, this approach has the downside of preventing the use of dynamic proxies in Java 1.5, so you need to manually deploy stubs. (In Java 1.4 and earlier, you always have to use stubs.)

There's one other kind of `RemoteServer` in the standard Java class library, the `java.rmi.activation.Activatable` class:

```
public abstract class Activatable extends RemoteServer // Java 1.2
```

A `UnicastRemoteObject` exists only as long as the server that created it still runs. When the server dies, the object is gone forever. `Activatable` objects allow clients to reconnect to servers at different times across server shutdowns and restarts and still access the same remote objects. It also has static `Activatable.exportObject( )` methods to invoke if you don't want to subclass `Activatable`.

Example 18-3, the `FibonacciImpl` class, implements the remote interface `Fibonacci`. This class has a constructor and two `getFibonacci( )` methods. Only the `getFibonacci( )` methods will be available to the client, because they're the only ones defined by the `Fibonacci` interface. The constructor is used on the server side but is not available to the client.

Example 18-3. The FibonacciImpl class

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.math.BigInteger;

public class FibonacciImpl extends UnicastRemoteObject implements Fibonacci {

  public FibonacciImpl( ) throws RemoteException {
    super( );
  }

  public BigInteger getFibonacci(int n) throws RemoteException {
    return this.getFibonacci(new BigInteger(Long.toString(n)));
  }

  public BigInteger getFibonacci(BigInteger n) throws RemoteException {

    System.out.println("Calculating the " + n + "th Fibonacci number");
    BigInteger zero = new BigInteger("0");
    BigInteger one  = new BigInteger("1");

    if (n.equals(zero)) return one;
    if (n.equals(one)) return one;

    BigInteger i  = one;
    BigInteger low  = one;
    BigInteger high  = one;

    while (i.compareTo(n) == -1) {
      BigInteger temp = high;
      high = high.add(low);
      low = temp;
      i = i.add(one);
    }

    return high;

  }
}
```

The `FibonacciImpl( )` constructor just calls the superclass constructor that exports the object; that is, it creates a `UnicastRemoteObject` on some port and starts it listening for connections. The constructor is declared to throw `RemoteException` because the `UnicastRemoteObject` constructor can throw that exception.

The `getFibonacci(int n)` method is trivial. It simply returns the result of converting its argument to a `BigInteger` and calling the second `getFibonacci( )` method. The second method actually performs the calculation. It uses `BigInteger` throughout the calculation to allow

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

for arbitrarily large Fibonacci numbers of an arbitrarily large index to be calculated. This can use a lot of CPU power and huge amounts of memory. That's why you might want to move it to a special-purpose calculation server rather than performing the calculation locally.

Although `getFibonacci( )` is a remote method, there's nothing different about the method itself. This is a simple case, but even vastly more complex remote methods are not algorithmically different than their local counterparts. The only difference—that a remote method is declared in a remote interface and a local method is not—is completely external to the method itself.

Next, we need to write a server that makes the `Fibonacci` remote object available to the world. Example 18-4 is such a server. All it has is a `main( )` method. It begins by entering a `try` block that catches `RemoteException`. Then it constructs a new `FibonacciImpl` object and binds that object to the name "fibonacci" using the `Naming` class to talk to the local registry. A registry keeps track of the available objects on an RMI server and the names by which they can be requested. When a new remote object is created, the object adds itself and its name to the registry with the `Naming.bind( )` or `Naming.rebind( )` method. Clients can then ask for that object by name or get a list of all the remote objects that are available. Note that there's no rule that says the name the object has in the registry has to have any necessary relation to the class name. For instance, we could have called this object "Fred". Indeed, there might be multiple instances of the same class all bound in a registry, each with a different name. After registering itself, the server prints a message on `System.out` signaling that it is ready to begin accepting remote invocations. If something goes wrong, the `catch` block prints a simple error message.

Example 18-4. The FibonacciServer class

```
import java.net.*;
import java.rmi.*;

public class FibonacciServer {

  public static void main(String[] args) {

    try {
      FibonacciImpl f = new FibonacciImpl( );
      Naming.rebind("fibonacci", f);
      System.out.println("Fibonacci Server ready.");
    }
     catch (RemoteException rex) {
      System.out.println("Exception in FibonacciImpl.main: " + rex);
    }
    catch (MalformedURLException ex) {
      System.out.println("MalformedURLException " + ex);
    }

  }

}
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Although the `main( )` method finishes fairly quickly here, the server will continue to run because a nondaemon thread is spawned when the `FibonacciImpl` object is bound to the registry. This completes the server code you need to write.

18.2.2. Compiling the Stubs

RMI uses stub classes to mediate between local objects and the remote objects running on the server. Each remote object on the server is represented by a stub class on the client. The stub contains the information in the `Remote` interface (in this example, that a `Fibonacci` object has two `getFibonacci( )` methods). Java 1.5 can sometimes generate these stubs automatically as they're needed, but in Java 1.4 and earlier, you must manually compile the stubs for each remote class. Even in Java 1.5, you still have to manually compile stubs for remote objects that are not subclasses of `UnicastRemoteObject` and are instead exported by calling `UnicastRemoteObject.exportObject( )`.

Fortunately, you don't have to write stub classes yourself: they can be generated automatically from the remote class's byte code using the *rmic* utility included with the JDK. To generate the stubs for the `FibonacciImpl` remote object, run *rmic* on the remote classes you want to generate stubs for. For example:

```
% rmic FibonacciImpl
% ls Fibonacci*
Fibonacci.class      FibonacciImpl_Stub.class  FibonacciServer.java
FibonacciImpl.class  Fibonacci.java
FibonacciImpl.java   FibonacciServer.class
```

rmic reads the *.class* file of a class that implements `Remote` and produces *.class* files for the stubs needed for the remote object. The command-line argument to *rmic* is the fully package-qualified class name (e.g., `com.macfaq.rmi.examples.Chat`, not just `Chat`) of the remote object class.

rmic supports the same command-line options as the *javac* compiler: for example, `-classpath` and `-d`. For instance, if the class doesn't fall in the class path, you can specify the location with the `-classpath` command-line argument. The following command searches for *FibonacciImpl.class* in the directory *test/classes*:

```
% rmic -classpath test/classes FibonacciImpl
```

18.2.3. Starting the Server

Now you're ready to start the server. There are actually two servers you need to run, the remote object itself (`FibonacciServer` in this example) and the registry that allows local clients to download a reference to the remote object. Since the server expects to talk to the registry, you must start the registry first. Make sure all the stub and server classes are in the server's class path and type:

```
% rmiregistry &
```

On Windows, you start it from a DOS prompt like this:

```
C:> start rmiregistry
```

In both examples, the registry runs in the background. The registry tries to listen to port 1,099 by default. If it fails, especially with a message like "java.net. SocketException: Address already in use", then some other program is using port 1099, possibly (though not necessarily) another registry service. You can run the registry on a different port by appending a port number like this:

```
% rmiregistry 2048 &
```

If you use a different port, you'll need to include that port in URLs that refer to this registry service.

Finally, you're ready to start the server. Run the server program just as you'd run any Java class with a `main( )` method:

```
% java FibonacciServer
Fibonacci Server ready.
```

Now the server and registry are ready to accept remote method calls. Next we'll write a client that connects to these servers to make such remote method calls.

18.2.4. The Client Side

Before a regular Java object can call a method, it needs a reference to the object whose method it's going to call. Before a client object can call a remote method, it needs a remote reference to the object whose method it's going to call. A program retrieves this remote reference from a registry on the server where the remote object runs. It queries the registry by calling the registry's `lookup ( )` method. The exact naming scheme depends on the registry; the `java.rmi.Naming` class provides a URL-based scheme for locating objects. As you can see in the following code, these URLs have been designed so that they are similar to *http* URLs. The protocol is *rmi*. The URL's

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

file field specifies the remote object's name. The fields for the hostname and the port number are unchanged:

```
Object o1 = Naming.lookup("rmi://login.ibiblio.org/fibonacci");
Object o2 = Naming.lookup("rmi://login.ibiblio.org:2048/fibonacci");
```

Like objects stored in `Hashtables`, `Vectors`, and other data structures that store objects of different classes, the object that is retrieved from a registry loses its type information. Therefore, before using the object, you must cast it to the remote interface that the remote object implements (not to the actual class, which is hidden from clients):

```
Fibonacci calculator = (Fibonacci) Naming.lookup("fibonacci");
```

Once a reference to the object has been retrieved and its type restored, the client can use that reference to invoke the object's remote methods pretty much as it would use a normal reference variable to invoke methods in a local object. The only difference is that you'll need to catch `RemoteException` for each remote invocation. For example:

```
try {
  BigInteger f56 = calculator.getFibonacci(56);
  System.out.println("The 56th Fibonacci number is " + f56);
  BigInteger f156 = calculator.getFibonacci(new BigInteger(156));
  System.out.println("The 156th Fibonacci number is " + f156);
}
catch (RemoteException ex) {
  System.err.println(ex)
}
```

Example 18-5 is a simple client for the `Fibonacci` interface of the last section.

Example 18-5. The FibonacciClient

```
import java.rmi.*;
import java.net.*;
import java.math.BigInteger;

public class FibonacciClient {

  public static void main(String args[]) {

    if (args.length == 0 || !args[0].startsWith("rmi:")) {
      System.err.println(
        "Usage: java FibonacciClient rmi://host.domain:port/fibonacci number");
      return;
    }

    try {
      Object o = Naming.lookup(args[0]);
      Fibonacci calculator = (Fibonacci) o;
      for (int i = 1; i < args.length; i++) {
        try {
          BigInteger index = new BigInteger(args[i]);
          BigInteger f = calculator.getFibonacci(index);
          System.out.println("The " + args[i] + "th Fibonacci number is "
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
              + f);
            }
            catch (NumberFormatException e) {
              System.err.println(args[i] + "is not an integer.");
            }
          }
        }
        catch (MalformedURLException ex) {
          System.err.println(args[0] + " is not a valid RMI URL");
        }
        catch (RemoteException ex) {
          System.err.println("Remote object threw exception " + ex);
        }
        catch (NotBoundException ex) {
          System.err.println(
           "Could not find the requested remote object on the server");
        }
      }
    }
```

Compile the class as usual. Notice that because the object that `Naming.lookup( )` returns is cast to a `Fibonacci`, either the *Fibonacci.java* or *Fibonacci.class* file needs to be available on the local host. A general requirement for compiling a client is to have either the byte or source code for the remote interface you're connecting to. To some extent, you can relax this a little bit by using the reflection API, but you'll still need to know at least something about the remote interface's API. Most of the time, this isn't an issue, since the server and client are written by the same programmer or team. The point of RMI is to allow a VM to invoke methods on remote objects, not to compile against remote objects.

18.2.5. Running the Client

Go back to the client system. Make sure that the client system has *FibonacciClient.class*, *Fibonacci.class*, and *FibonacciImpl_Stub.class* in its class path. (If both the client and the server are running Java 1.5, you don't need the stub class.) On the client system, type:

```
C:\>java FibonacciClient rmi://host.com/fibonacci 0 1 2 3 4 5 55 155
```

You should see:

```
The 0th Fibonacci number is 1
The 1th Fibonacci number is 1
The 2th Fibonacci number is 2
The 3th Fibonacci number is 3
The 4th Fibonacci number is 5
The 5th Fibonacci number is 8
The 55th Fibonacci number is 225851433717
The 155th Fibonacci number is 1788903347851831682574552878917 92
```

The client converts the command-line arguments to `BigInteger` objects. It sends those objects over the wire to the remote server. The server receives each of those objects, calculates the Fibonacci number for that index, and sends a `BigInteger` object back over the Internet to the client. Here,

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

I'm using a PC for the client and a remote Unix box for the server. You can actually run both server and client on the same machine, although that's not as interesting.

18.3. Loading Classes at Runtime

All the client really has to know about the remote object is its remote interface. Everything else it needs—for instance, the stub classes—can be loaded from a web server (though not an RMI server) at runtime using a class loader. Indeed, this ability to load classes from the network is one of the unique features of Java. This is especially useful in applets. The web server can send the browser an applet that communicates back with the server; for instance, to allow the client to read and write files on the server. However, as with any time that classes are loaded from a potentially untrusted host, they must be checked by a `SecurityManager`.

Unfortunately, while remote objects are actually quite easy to work with when you can install the necessary classes in the local client class path, doing so when you have to dynamically load the stubs and other classes is fiendishly difficult. The class path, the security architecture, and the reliance on poorly documented environment variables are all bugbears that torment Java programmers. Getting a local client object to download remote objects from a server requires manipulating all of these in precise detail. Making even a small mistake prevents programs from running, and only the most generic of exceptions is thrown to tell the poor programmers what they did wrong. Exactly how difficult it is to make the programs work depends on the context in which the remote objects are running. In general, applet clients that use RMI are somewhat easier to manage than standalone application clients. Standalone applications are feasible if the client can be relied on to have access to the same *.class* files as the server has. Standalone applications that need to load classes from the server border on impossible.

Example 18-6 is an applet client for the `Fibonacci` remote object. It has the same basic structure as the `FibonacciClient` in Example 18-5. However, it uses a `TextArea` to display the message from the server instead of using `System.out`.

Example 18-6. An applet client for the Fibonacci object

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.rmi.*;
import java.math.BigInteger;

public class FibonacciApplet extends Applet {
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
    private TextArea  resultArea
     = new TextArea("", 20, 72, TextArea.SCROLLBARS_BOTH);
    private TextField inputArea  = new TextField(24);
    private Button calculate = new Button("Calculate");
    private String server;

    public void init( ) {

      this.setLayout(new BorderLayout( ));

      Panel north = new Panel( );
      north.add(new Label("Type a non-negative integer"));
      north.add(inputArea);
      north.add(calculate);
      this.add(resultArea, BorderLayout.CENTER);
      this.add(north, BorderLayout.NORTH);
      Calculator c = new Calculator( );
      inputArea.addActionListener(c);
      calculate.addActionListener(c);
      resultArea.setEditable(false);

      server = "rmi://" + this.getCodeBase( ).getHost( ) + "/fibonacci";

    }

    class Calculator implements ActionListener {

      public void actionPerformed(ActionEvent evt) {

        try {
          String input = inputArea.getText( );
          if (input != null) {
            BigInteger index = new BigInteger(input);
            Fibonacci f = (Fibonacci) Naming.lookup(server);
            BigInteger result =  f.getFibonacci(index);
            resultArea.setText(result.toString( ));
          }
        }
        catch (Exception ex) {
          resultArea.setText(ex.getMessage( ));
        }
      }
    }
  }
```

You'll notice that the *rmi* URL is built from the applet's own codebase. This helps avoid nasty security problems that arise when an applet tries to open a network connection to a host other than the one it came from. RMI-based applets are certainly not exempt from the usual restrictions on network connections.

Example 18-7 is a simple HTML file that can be used to load the applet from the web browser.

Example 18-7. FibonacciApplet.html

```
<html>
<head>
<title>RMI Applet</title>
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
</head>
<body>
<h1>RMI Applet</h1>

<p>
<applet align="center" code="FibonacciApplet" width="300" height="100">
</applet>
<hr />
</p>
</body>
</html>
```

Place *FibonacciImpl_Stub.class*, *Fibonacci.class*, *FibonacciApplet.html*, and *FibonacciServer.class* in the same directory on your web server. Add this directory to the server's class path and start *rmiregistry* on the server. Then start `FibonacciServer` on the server. For example:

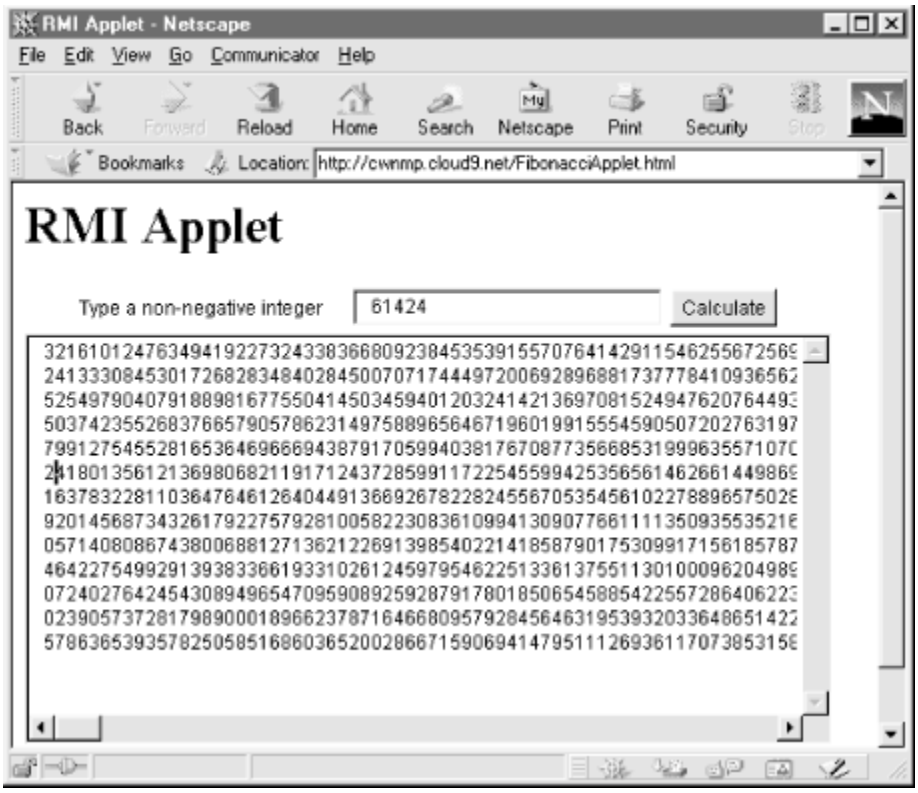
```
% rmiregistry &
% java FibonacciServer &
```

Make sure that both of these are running on the actual web server machine. Many web server farms use different machines for site maintenance and web serving, even though both mount the same filesystems. To get past the applet security restriction, both *rmiregistry* and `FibonacciServer` have to be running on the machine that serves the *FibonacciApplet.class* file to web clients.

Now load *FibonacciApplet.html* into a web browser from the client. Figure 18-2 shows the result.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Figure 18-2. The Fibonacci applet



For applications, it's much easier if you can load all the classes you need before running the program. You can load classes from a web server running on the same server the remote object is running on, if necessary. To do this, set the `java.rmi.server.codebase` Java system property on the server (where the remote object runs) to the URL where the *.class* files are stored on the network. For example, to specify that the classes can be found at http://www.cafeaulait.org/rmi2/, you would type:

```
% java -Djava.rmi.server.codebase=http://www.cafeaulait.org/rmi2/
FibonacciServer & Fibonacci Server ready.
```

If the classes are in packages, the `java.rmi.server.codebase` property points to the directory containing the top-level *com* or *org* directory rather than the directory containing the *.class* files themselves. Both servers and clients will load the *.class* files from this location if the files are not found in the local class path first.

Loading classes from the remote server makes the coupling between the server and the client a little less tight. However, any client program you write will normally have to know quite a bit about the system it's talking to in order to do something useful. This usually involves having at least the remote interface available on the client at compile time and runtime. Even if you use reflection to avoid that, you'll still need to know the signatures and something about the behavior of the methods you plan to invoke. RMI just doesn't lend itself to truly loose coupling like you might see in a SOAP

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

or, better yet, RESTful server. The RMI design metaphor is more running one program on several machines than it is having several programs on different machines that communicate with each other. Therefore, it's easiest if both sides of the connection have all the code available to them when the program starts up.

18.4. The java.rmi Package

The `java.rmi` package contains the classes that are seen by clients (objects that invoke remote methods). Both clients and servers should import `java.rmi`. While servers need a lot more infrastructure than is present in this package, `java.rmi` is all clients need. This package contains one interface, three classes, and a handful of exceptions.

18.4.1. The Remote Interface

The `Remote` interface tags objects as remote objects. It doesn't declare any methods; remote objects usually implement a subclass of `Remote` that does declare some methods. The methods that are declared in the interface are the methods that can be invoked remotely.

Example 18-8 is a database interface that declares a single method, `SQLQuery( )`, which accepts a `String` and returns a `String` array. A class that implements this interface would include the code to send an SQL query to a database and return the result as a `String` array.

Example 18-8. A database interface

```
import java.rmi.*;

public interface SQL extends Remote {

  public String[] SQLQuery(String query) throws RemoteException;

}
```

An `SQLImpl` class that implemented the `SQL` interface would probably have more methods, some of which might be public. However, only the `SQLQuery()` method can be invoked by a client. Because the `Remote` interface is not a class, a single object can implement multiple `Remote` subinterfaces. In this case, any method declared in any `Remote` interface can be invoked by a client.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

18.4.2. The Naming Class

The `java.rmi.Naming` class talks to a registry running on the server in order to map URLs like *rmi://login.ibiblio.org/myRemoteObject* to particular remote objects on particular hosts. You can think of a registry as a DNS for remote objects. Each entry in the registry has a name and an object reference. Clients give the name (via a URL) and get back a reference to the remote object.

As you've seen, an *rmi* URL looks exactly like an *http* URL except that the scheme is `rmi` instead of `http`. Furthermore, the path part of the URL is an arbitrary name that the server has bound to a particular remote object, not a filename.

The biggest deficiency of `Naming` is that for security reasons (avoiding man-in-the-middle attacks), it has to run on the same server as the remote objects. It cannot register multiple objects on several different servers. If this is too restrictive, a Java Naming and Directory Interface (JNDI) context can add an additional layer of indirection so that multiple RMI registries can be presented through a single directory. Clients need only know the address of the main JNDI directory. They do not need to know the addresses of all the individual RMI registries the JNDI context is proxying for.

The `Naming` class has five public methods: `list( )`, to list all the names bound in the registry; `lookup( )`, to find a specific remote object given its URL; `bind( )`, to bind a name to a specific remote object; `rebind( )`, to bind a name to a different remote object; and `unbind( )`, to remove a name from the registry. Let's look at these methods in turn.

18.4.2.1. public static String[] list(String url) throws RemoteException, MalformedURLException

The `list( )` method returns an array of strings, one for each URL that is currently bound. The `url` argument is the URL of the `Naming` registry to query. Only the protocol, host, and port are used. The path part of the URL is ignored. `list( )` throws a `MalformedURLException` if `url` is not a valid *rmi* URL. A `RemoteException` is thrown if anything else goes wrong, such as the registry's not being reachable or refusing to supply the requested information.

Example 18-9 is a simple program that lists all the names currently bound in a particular registry. It's sometimes useful when debugging RMI problems. It allows you to determine whether the names you're using are the names the server expects.

Example 18-9. RegistryLister

```
import java.rmi.*;

public class RegistryLister {

  public static void main(String[] args) {

    int port = 1099;

    if (args.length == 0) {
      System.err.println("Usage: java RegistryLister host port");
      return;
    }

    String host = args[0];

    if (args.length > 1) {
      try {
        port = Integer.parseInt(args[1]);
        if (port <1 || port > 65535) port = 1099;
      }
      catch (NumberFormatException ex) {}

    }

    String url = "rmi://" + host + ":" + port + "/";
    try {
      String[] remoteObjects = Naming.list(url);
      for (int i = 0; i < remoteObjects.length; i++) {
        System.out.println(remoteObjects[i]);
      }
    }
    catch (RemoteException ex) {
      System.err.println(ex);
    }
    catch (java.net.MalformedURLException ex) {
      System.err.println(ex);
    }
  }
}
```

Here's a result from a run against the RMI server I was using to test the examples in this chapter:

```
% java RegistryLister login.ibiblio.org
rmi://login.ibiblio.org:1099/fibonacci
rmi://login.ibiblio.org:1099/hello
```

You can see that the format for the strings is full *rmi* URLs rather than just names. It turns out this is a bug; in Java 1.4.1 and later, the bug has been fixed. In these versions, the scheme part of the URI is no longer included. In other words, the output looks like this:

```
//login.ibiblio.org:1099/fibonacci
//login.ibiblio.org:1099/hello
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

18.4.2.2. public static Remote lookup(String url) throws RemoteException, NotBoundException, AccessException, MalformedURLException

A client uses the `lookup()` method to retrieve the remote object associated with the file portion of the name; so, given the URL *rmi://login.ibiblio.org:2001/myRemoteObject*, it would return the object bound to `myRemoteObject` from *login.ibiblio.org* on port 2,001.

This method throws a `NotBoundException` if the remote server does not recognize the name. It throws a `RemoteException` if the remote registry can't be reached; for instance, because the network is down or because no registry service is running on the specified port. An `AccessException` is thrown if the server refuses to look up the name for the particular host. Finally, if the URL is not a proper *rmi* URL, it throws a `MalformedURLException`.

18.4.2.3. public static void bind(String url, Remote object) throws RemoteException, AlreadyBoundException, MalformedURLException, AccessException

A server uses the `bind( )` method to link a name like `myRemoteObject` to a remote object. If the binding is successful, clients will be able to retrieve the remote object stub from the registry using a URL like *rmi://login.ibiblio.org:2001/myRemoteObject*.

Many things can go wrong with the binding process. `bind()` throws a `MalformedURLException` if `url` is not a valid *rmi* URL. It throws a `RemoteException` if the registry cannot be reached. It throws an `AccessException`, a subclass of `RemoteException`, if the client is not allowed to bind objects in this registry. If the URL is already bound to a local object, it throws an `AlreadyBoundException`.

18.4.2.4. public static void unbind(String url) throws RemoteException, NotBoundException, AlreadyBoundException, MalformedURLException, AccessException // Java 1.2

The `unbind( )` method removes the object with the given URL from the registry. It's the opposite of the `bind( )` method. What `bind( )` has bound, `unbind( )` releases. `unbind( )` throws a `NotBoundException` if `url` was not bound to an object in the first place. Otherwise, this method can throw the same exceptions for the same reasons as `bind()`.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

18.4.2.5. public static void rebind(String url, Remote object) throws RemoteException, AccessException, MalformedURLException

The `rebind( )` method is just like the `bind( )` method, except that it binds the URL to the object, even if the URL is already bound. If the URL is already bound to an object, the old binding is lost. Thus, this method does not throw an `AlreadyBoundException`. It can still throw `RemoteException`, `AccessException`, or `MalformedURLException`, which have the same meanings as they do when thrown by `bind()`.

18.4.3. The RMISecurityManager Class

A client loads stubs from a potentially untrustworthy server; in this sense, the relationship between a client and a stub is somewhat like the relationship between a browser and an applet. Although a stub is only supposed to marshal arguments and unmarshal return values and send them across the network, from the standpoint of the virtual machine, a stub is just another class with methods that can do just about anything. Stubs produced by *rmic* shouldn't misbehave; but there's no reason someone couldn't handcraft a stub that would do all sorts of nasty things, such as reading files or erasing data. The Java virtual machine does not allow stub classes to be loaded across the network unless there's some `SecurityManager` object in place. (Like other classes, stub classes can always be loaded from the local class path.) For applets, the standard `AppletSecurityManager` fills this need. Applications can use the `RMISecurityManager` class to protect themselves from miscreant stubs:

```
public class RMISecurityManager extends SecurityManager
```

In Java 1.1, this class implements a policy that allows classes to be loaded from the server's codebase (which is not necessarily the same as the server itself) and allows the necessary network communications between the client, the server, and the codebase. In Java 1.2 and later, the `RMISecurityManager` doesn't allow even that, and this class is so restrictive, it's essentially useless. In the Java 1.5 documentation, Sun finally admitted the problem: "`RMISecurityManager` implements a policy that is no different than the policy implemented by `SecurityManager`. Therefore an RMI application should use the `SecurityManager` class or another application-specific `SecurityManager` implementation instead of this class."

18.4.4. Remote Exceptions

The `java.rmi` package defines 16 exceptions, listed in Table 18-1. Most extend `java.rmi.RemoteException`. `java.rmi.RemoteException` extends

`java.io.IOException`. `AlreadyBoundException` and `NotBoundException` extend `java.lang.Exception`. Thus, all are checked exceptions that must be enclosed in a `try` block or declared in a `throws` clause. There's also one runtime exception, `RMISecurityException`, a subclass of `SecurityException`.

Remote methods depend on many things that are not under your control: for example, the state of the network and other necessary services such as DNS. Therefore, any remote method can fail: there's no guarantee that the network won't be down when the method is called. Consequently, all remote methods must be declared to throw the generic `RemoteException` and all calls to remote methods should be wrapped in a `try` block. When you just want to get a program working, it's simplest to catch `RemoteException`:

```
try {
  // call remote methods...
}
catch (RemoteException ex) {
  System.err.println(ex);
}
```

More robust programs should try to catch more specific exceptions and respond accordingly.

Table 18-1. Remote exceptions

Exception	Meaning
`AccessException`	A client tried to do something that only local objects are allowed to do.
`AlreadyBoundException`	The URL is already bound to another object.
`ConnectException`	The server refused the connection.
`ConnectIOException`	An I/O error occurred while trying to make the connection between the local and the remote host.
`MarshalException`	An I/O error occurred while attempting to marshal (serialize) arguments to a remote method. A corrupted I/O stream could cause this exception; making the remote method call again might be successful.
`UnmarshalException`	An I/O error occurred while attempting to unmarshal (deserialize) the value returned by a remote method. A corrupted I/O stream could cause this exception; making the remote method call again might be successful.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Exception	Meaning
`NoSuchObjectException`	The object reference is invalid or obsolete. This might occur if the remote host becomes unreachable while the program is running, perhaps because of network congestion, system crash, or other malfunction.
`NotBoundException`	The URL is not bound to an object. This might be thrown when you try to reference an object whose URL was rebound out from under it.
`RemoteException`	The generic superclass for all exceptions having to do with remote methods.
`ServerError`	Despite the name, this is indeed an exception, not an error. It indicates that the server threw an error while executing the remote method.
`ServerException`	A `RemoteException` was thrown while the remote method was executing.
`StubNotFoundException`	The stub for a class could not be found. The stub file may be in the wrong directory on the server, there could be a namespace collision between the class that the stub substitutes for and some other class, or the client could have requested the wrong URL.
`UnexpectedException`	Something unforeseen happened. This is a catchall that occurs only in bizarre situations.
`UnknownHostException`	The host cannot be found. This is very similar to `java.net.UnknownHostException`.

The `RemoteException` class contains a single public field called `detail`:

```
public Throwable detail
```

This field may contain the actual exception thrown on the server side, so it gives you further information about what went wrong. For example:

```
try {
  // call remote methods...
}
catch (RemoteException ex) {
  System.err.println(ex.detail);
  ex.detail.printStackTrace( );
}
```

In Java 1.4 and later, use the standard `getCause()` method to return the nested exception instead:

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
try {
  // call remote methods...
}
catch (RemoteException ex) {
  System.err.println(ex.getCause( ));
  ex.getCause( ).printStackTrace( );
}
```

18.5. The java.rmi.registry Package

How does a client that needs a remote object locate that object on a distant server? More precisely, how does it get a remote reference to the object? Clients find out what remote objects are available by querying the server's *registry*. A registry advertises the availability of the server's remote objects. Clients query the registry to find out what remote objects are available and to get remote references to those objects. You've already seen one: the `java.rmi.Naming` class for interfacing with registries.

The `Registry` interface and the `LocateRegistry` class allow clients to retrieve remote objects on a server by name. A `RegistryImpl` is a subclass of `RemoteObject`, which links names to particular `RemoteObject` objects. Clients use the methods of the `LocateRegistry` class to retrieve the `RegistryImpl` for a specific host and port.

18.5.1. The Registry Interface

The `java.rmi.registry.Registry` interface has five public methods: `bind( )`, to bind a name to a specific remote object; `list( )`, to list all the names bound in the registry; `lookup( )`, to find a specific remote object given its URL; `rebind( )`, to bind a name to a different remote object; and `unbind( )`, to remove a name from the registry. All of these behave exactly as previously described in the `java.rmi.Naming` class, which implements this interface. Other classes that implement this interface may use a different scheme for mapping names to particular objects, but the methods still have the same meaning and signatures.

Besides these five methods, the `Registry` interface also has one field, `Registry.REGISTRY_PORT`, the default port on which the registry listens. Its value is 1099.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

18.5.2. The LocateRegistry Class

The `java.rmi.registry.LocateRegistry` class lets the client find the registry in the first place. This is achieved with five overloaded versions of the `static LocateRegistry.getRegistry()` method:

```
public static Registry getRegistry( ) throws RemoteException
public static Registry getRegistry(int port) throws RemoteException
public static Registry getRegistry(String host) throws RemoteException
public static Registry getRegistry(String host, int port)
 throws RemoteException
public static Registry getRegistry(String host, int port,  // Java 1.2
 RMIClientSocketFactory factory) throws RemoteException
```

Each of these methods returns a `Registry` object that can be used to get remote objects by name. `LocateRegistry.getRegistry( )` returns a stub for the `Registry` running on the local host on the default port, 1,099. `LocateRegistry.getRegistry(int port)` returns a stub for the `Registry` running on the local host on the specified port. `LocateRegistry.getRegistry(String host)` returns a stub for the `Registry` for the specified host on the default port, 1,099. `LocateRegistry.getRegistry(String host, int port)` returns a stub for the `Registry` on the specified host on the specified port. Finally, `LocateRegistry.getRegistry(String host, int port, RMIClientSocketFactory factory)` returns a stub to the registry running on the specified host and port, which will be contacted using sockets created by the provided `java.rmi.server.RMIClientSocketFactory` object. If the `host String` is `null`, `getRegistry( )` uses the local host; if the `port` argument is negative, it uses the default port. Each of these methods can throw an arbitrary `RemoteException`.

For example, a remote object that wanted to make itself available to clients might do this:

```
Registry r = LocateRegistry.getRegistry( );
r.bind("My Name", this);
```

A remote client that wished to invoke this remote object might then say:

```
Registry r = LocateRegistry.getRegistry("thehost.site.com");
RemoteObjectInterface tro = (RemoteObjectInterface) r.lookup("MyName");
tro.invokeRemoteMethod( );
```

The final two methods in the `LocateRegistry` class are the overloaded `LocateRegistry.createRegistry( )` methods. These create a registry and start it listening on the specified port. As usual, each can throw a `RemoteException`. Their signatures are:

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
public static Registry createRegistry(int port) throws RemoteException
public static Registry createRegistry(int port,
 RMIClientSocketFactory csf, RMIServerSocketFactory ssf) // Java 1.2
 throws RemoteException
```

18.6. The java.rmi.server Package

The `java.rmi.server` package is the most complex of all the RMI packages; it contains the scaffolding for building remote objects and thus is used by objects whose methods will be invoked by clients. The package defines 6 exceptions, 9 interfaces, and 10-12 classes (depending on the Java version). Fortunately, you only need to be familiar with a few of these in order to write remote objects. The important classes are the `RemoteObject` class, which is the basis for all remote objects; the `RemoteServer` class, which extends `RemoteObject`; and the `UnicastRemoteObject` class, which extends `RemoteServer`. Any remote objects you write will likely either use or extend `UnicastRemoteObject`. Clients that call remote methods but are not themselves remote objects don't use these classes and therefore don't need to import `java.rmi.server`.

18.6.1. The RemoteObject Class

Technically, a remote object is not an instance of the `RemoteObject` class but an instance of any class that implements a `Remote` interface. In practice, most remote objects will be instances of a subclass of `java.rmi.server.RemoteObject`:

```
public abstract class RemoteObject extends Object
 implements Remote, Serializable
```

You can think of this class as a special version of `java.lang.Object` for remote objects. It provides `toString( )`, `hashCode( )`, `clone( )`, and `equals( )` methods that make sense for remote objects. If you create a remote object that does not extend `RemoteObject`, you need to override these methods yourself.

The `equals( )` method compares the remote object references of two `RemoteObjects` and returns true if they point to the same `RemoteObject`. As with the `equals( )` method in the `Object` class, you may want to override this method to provide a more meaningful definition of equality.

The `toString( )` method returns a `String` that describes the `RemoteObject`. Most of the time, `toString()` returns the hostname and port from which the remote object came as well as

a reference number for the object. You can override this method in your own subclasses to provide more meaningful string representations.

The `hashCode( )` method maps a presumably unique `int` to each unique object; this integer may be used as a key in a `Hashtable`. It returns the same value for all remote references that refer to the same remote object. Thus, if a client has several remote references to the same object on the server, or multiple clients have references to that object, they should all have the same hash code.

The final instance method in this class is `getRef()`:

```
public RemoteRef getRef( ) // Java 1.2
```

This returns a remote reference to the class:

```
public abstract interface RemoteRef extends Externalizable
```

There's also one static method, `RemoteObject.toStub( )`:

```
public static Remote toStub(Remote ro) // Java 1.2
 throws NoSuchObjectException
```

`RemoteObject.toStub( )` converts a given remote object into the equivalent stub object for use in the client virtual machine, which can help you dynamically generate stubs from within your server without using *rmic*.

18.6.2. The RemoteServer Class

The `RemoteServer` class extends `RemoteObject`; it is an abstract superclass for server implementations such as `UnicastRemoteObject`. It provides a few simple utility methods needed by most server objects:

```
public abstract class RemoteServer extends RemoteObject
```

`UnicastRemoteObject` is the most commonly used subclass of `RemoteServer` included in the core library. Two others, `Activatable` and `ActivationGroup`, are found in the `java.rmi.activation` package. You can add others (for example, a UDP or multicast remote server) by writing your own subclass of `RemoteServer`.

18.6.2.1. Constructors

`RemoteServer` has two constructors:

```
protected RemoteServer( )
protected RemoteServer(RemoteRef r)
```

However, you won't instantiate this class yourself. Instead, you will instantiate a subclass like `UnicastRemoteObject`. That class's constructor calls one of these protected constructors from the first line of its constructor.

18.6.2.2. Getting information about the client

The `RemoteServer` class has one method to locate the client with which you're communicating:

```
public static String getClientHost( ) throws ServerNotActiveException
```

`RemoteServer.getClientHost()` returns a `String` that contains the hostname of the client that invoked the currently running method. This method throws a `ServerNotActiveException` if the current thread is not running a remote method.

18.6.2.3. Logging

For debugging purposes, it is sometimes useful to see the calls that are being made to a remote object and the object's responses. You get a log for a `RemoteServer` by passing an `OutputStream` object to the `setLog( )` method:

```
public static void setLog(OutputStream out)
```

Passing `null` turns off logging. For example, to see all the calls on `System.err` (which sends the log to the Java console), you would write:

```
myRemoteServer.setLog(System.err);
```

Here's some log output I collected while debugging the Fibonacci programs in this chapter:

```
Sat Apr 29 12:20:36 EDT 2000:RMI:TCP Accept-1:[titan.oit.unc.edu:
sun.rmi.transport.DGCImpl[0:0:0, 2]: java.rmi.dgc.Lease
dirty(java.rmi.server.ObjID[], long, java.rmi.dgc.Lease)]
Fibonacci Server ready.
Sat Apr 29 12:21:27 EDT 2000:RMI:TCP Accept-2:[macfaq.dialup.cloud9.net:
sun.rmi.transport.DGCImpl[0:0:0, 2]: java.rmi.dgc.Lease
dirty(java.rmi.server.ObjID[], long, java.rmi.dgc.Lease)]
Sat Apr 29 12:22:36 EDT 2000:RMI:TCP Accept-3:[macfaq.dialup.cloud9.net:
```

```
sun.rmi. transport.DGCImpl[0:0:0, 2]: java.rmi.dgc.Lease
dirty(java.rmi.server.ObjID[], long, java.rmi.dgc.Lease)]
Sat Apr 29 12:22:39 EDT 2000:RMI:TCP Accept-3:[macfaq.dialup.cloud9.net:
FibonacciImpl[0]: java.math.BigInteger getFibonacci(java.math.BigInteger)]
Sat Apr 29 12:22:39 EDT 2000:RMI:TCP Accept-3:[macfaq.dialup.cloud9.net:
FibonacciImpl[0]: java.math.BigInteger getFibonacci(java.math.BigInteger)]
```

If you want to add extra information to the log along with what's provided by the `RemoteServer` class, you can retrieve the log's `PrintStream` with the `getLog( )` method:

```
public static PrintStream getLog( )
```

Once you have the print stream, you can write on it to add your own comments to the log. For example:

```
PrintStream p = RemoteServer.getLog( );
p.println("There were " + n + " total calls to the remote object.");
```

18.6.3. The UnicastRemoteObject Class

The `UnicastRemoteObject` class is a concrete subclass of `RemoteServer`. To create a remote object, you can extend `UnicastRemoteObject` and declare that your subclass implements some subinterface of `java.rmi.Remote`. The methods of the interface provide functionality specific to the class, while the methods of `UnicastRemoteObject` handle general remote object tasks like marshalling and unmarshalling arguments and return values. All of this happens behind the scenes. As an application programmer, you don't need to worry about it.

A `UnicastRemoteObject` runs on a single host, uses TCP sockets to communicate, and has remote references that do not remain valid across server restarts. While this is a good general-purpose framework for remote objects, it is worth noting that you can implement other kinds of remote objects. For example, you may want a remote object that uses UDP, or one that remains valid if the server is restarted, or even one that distributes the load across multiple servers. To create remote objects with these properties, extend `RemoteServer` directly and implement the abstract methods of that class. However, if you don't need anything so esoteric, it's much easier to subclass `UnicastRemoteObject`.

The `UnicastRemoteObject` class has three protected constructors:

```
protected UnicastRemoteObject( ) throws RemoteException
protected UnicastRemoteObject(int port)              // Java 1.2
 throws RemoteException
protected UnicastRemoteObject(int port, RMIClientSocketFactory csf,
 RMIServerSocketFactory ssf) throws RemoteException  // Java 1.2
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

When you write a subclass of `UnicastRemoteObject`, you call one of these constructors, either explicitly or implicitly, in the first line of each constructor of your subclass. All three constructors can throw a `RemoteException` if the remote object cannot be created.

The noargs constructor creates a `UnicastRemoteObject` that listens on an anonymous port chosen at runtime. By the way, this is an example of an obscure situation I mentioned in Chapter 9 and Chapter 10. The server is listening on an anonymous port. Normally, this situation is next to useless because it is impossible for clients to locate the server. In this case, clients locate servers by using a registry that keeps track of the available servers and the ports they are listening to.

The downside to listening on an anonymous port is that it's not uncommon for a firewall to block connections to that port. The next two constructors listen on specified ports so you can ask the network administrators to allow traffic for those ports through the firewall.

If the network administrators are uncooperative, you'll need to use HTTP tunneling or a proxy server or both. The third constructor also allows you to specify the socket factories used by this `UnicastRemoteObject`. In particular, you can supply a socket factory that returns sockets that know how to get through the firewall.

The `UnicastRemoteObject` class has several public methods:

```
public Object clone( ) throws CloneNotSupportedException
public static RemoteStub exportObject(Remote r) throws RemoteException
public static Remote exportObject(Remote r, int port)
 throws RemoteException       // Java 1.2
public static Remote exportObject(Remote r, int port,
 RMIClientSocketFactory csf, RMIServerSocketFactory ssf)
 throws RemoteException       // Java 1.2
public static boolean unexportObject(Remote r, boolean force)
 throws NoSuchObjectException // Java 1.2
```

The `clone( )` method simply creates a clone of the remote object. You call the `UnicastRemoteObject.exportObject( )` to use the infrastructure that `UnicastRemoteObject` provides for an object that can't subclass `UnicastRemoteObject`. Similarly, you pass an object `UnicastRemoteObject.unexportObject( )` to stop a particular remote object from listening for invocations.

18.6.4. Exceptions

The `java.rmi.server` package defines a few more exceptions. The exceptions and their meanings are listed in Table 18-2. All but `java.rmi.server.ServerNotActiveException` extend, directly or indirectly,

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

`java.rmi.RemoteException`. All are checked exceptions that must be caught or declared in a `throws` clause.

Table 18-2. java.rmi.server exceptions

Exception	Meaning
`ExportException`	You're trying to export a remote object on a port that's already in use.
`ServerNotActiveException`	An attempt was made to invoke a method in a remote object that wasn't running.
`ServerCloneException`	An attempt to clone a remote object on the server failed.
`SocketSecurityException`	This subclass of `ExportException` is thrown when the `SecurityManager` prevents a remote object from being exported on the requested port.

This chapter has been a fairly quick look at Remote Method Invocation. For a more detailed treatment, see *Java RMI*, by William Grosso (O'Reilly).

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004