RMI: Observing the Distributed Pattern

Dan-Adrian German
Computer Science Department, Indiana University Bloomington
150 S. Woodlawn Ave., Bloomington, IN 47405 (dgerman@indiana.edu)

Abstract – **This paper presents a software design pattern that capitalizes on the object-oriented patterns already representing the foundation of Java RMI. Providing complete separation between the design and deployment stages, the pattern (not to be confused with an API) allows its user to write code that can run either locally on a single virtual machine, or over the network across several virtual machines, without any modification. The only restrictions placed on the user's code are those specified by Java RMI.**

Index Terms – Distributed processing, Java remote method invocation, Object-oriented patterns, Transparent networking.

INTRODUCTION

Programmers often consider network programming to be cryptic and unapproachable. Not that the APIs are difficult to understand, but more because the programmer is forced to implement protocols on top of these APIs. As a programmer you want to focus on the task at hand, which may require communicating or sending data to remote processes. If transmission of the data is handled invisibly and reliably, it is much easier to spend your energy perfecting your application and not worrying about what's under the hood. It's a lot like driving a car. It's possible to build a special purpose engine for every trip that you want to take, but normally all you want to do is climb in and drive. Of course, you may want to know if the engine is reliable, or powerful, but unless you are an engine designer, you're not going to be rebuilding engines [1].

This is not a tutorial or an introductory paper on Java RMI. This paper presents a software design pattern whereby the *exact same code* can be run either locally on a single virtual machine, or over the network across virtual machines. Thus the network becomes completely transparent and its role is justly reduced to that of a simple but very useful appliance.

The pattern is completely general and is presented in detail through an extensive example.

There are several practical implications of this pattern: first, a program transformation to turn a standalone simulation that involves multiple autonomous agents (competing or cooperating into a shared world) into a multiplayer networked game becomes immediate. Second, using RMI for networking is shown to be as powerful as using `try/catch` blocks for for exception handling: there should be a similar increase in programming productivity, for example. Third, the impact on the ways we think about (and approach the teaching of) the development of distributed applications should be equally significant since the development of a distributed application now becomes equivalent to writing a basic, non-networked, standalone application.

The derivation of the pattern has been described elsewhere [2]. Additional examples can be found in [3].

THE JAVA RMI

Remote Method Invocation (RMI) is a Java-specific version of a CORBA framework. RMI means that an object on one system can call a method in an object somewhere else on a network. It will send over parameters and get a result back automatically. It all happens invisibly and just looks like an invocation of a local method (it may take a little longer time of course). Compared to sockets RMI offers a higher-level interface. Clients can truly make procedure calls directly to their server. It is important to understand that RMI is completely a library feature. There is nothing in the Java language that was added to enable RMI. One can build an RMI library for any programming language.

This paper presents a software design pattern that capitalizes on the object-oriented patterns already representing the foundation of RMI. (In modern terminology such as the one introduced in [5], the `Bridge`, `Facade`, and `Adapter` patterns are those most frequently associated with the implementation of Java RMI.) During execution, applications can run into many kinds of errors of varying degrees of severity. Many programmers do not test for all possible error conditions, and for good reason: code becomes unintelligible if each method invocation checks for all possible errors before the next statement is executed. This trade-off creates a tension between correctness (checking for all errors) and clarity (not cluttering the basic flow of code with many error checks).

Exceptions provide a clean way to check for errors without cluttering code: application logic (code that deals with the intended purpose) is separate from code that deals with the erroneous, or unexpected. Likewise, the pattern presented here keeps the network-specific code (that supports the deployment stage) entirely separate from the application logic (which describes the intended design). This separation of functions in what is otherwise a very tightly coupled development strategy is indeed fortunate. To force an analogy we could say that it's similar to adding wings to an already existing automobile to obtain an airplane: no changes would have to be made to the engine (assuming it is powerful enough). By contrast, programming with sockets would require rebuilding the engine completely in order to turn the existing program (the automobile) into a distributed application (the airplane). To attempt another analogy, the use of this pattern is similar to the ability to participate in an auction either in person, or from

0-7803-8552-7/04/$20.00 © 2004 IEEE October 20 – 23, 2004, Savannah, GA
34th ASEE/IEEE Frontiers in Education Conference

a distance, using a cell phone (as in a teleconference). In this analogy sockets would interfere with the negotiation process, lacking the transparency provided by cell phones. The sentences would have to be crafted specifically for sockets, that is, perhaps by being spoken in a different language. In both cases the predominant aspect is that of component-based (i.e., plug-and-play) networking, one of the declared goals of Jini (see [6]) a technology built on Java RMI.

Debugging distributed applications in their run-time environment is notoriously hard and development and testing of application logic must be completed ahead of this step. Using Java RMI allows a developer to separate the two stages (development of the application logic from the deployment of the application in its distributed run-time environment) but the developer must acknowledge a specific pattern from the outset. We present this pattern below: it allows for a stage of fully carried out development of the application in an isolated run-time environment (that is, no network) and makes the switch to a true networked run-time environment completely transparent. No other approach offers this advantage. For related work (both RMI-based) see [7] and [8].

CLIENTS AND SERVERS

Before we present the code we need to clarify the meaning of *client* and *server* (especially in relationship to each other and as used here). As we shall see their meaning is indeed relative: clients can easily act as servers. But there is a difference between the two classes, and a clarification must be made from the outset. Imagine, therefore, that along with some of your friends (Larry, Michael and Toni) you have been invited to another of your friends' house, let's say for a negotiation. Your host's name is Dave, and he is the *server*. The rest of you are *client*s. The purpose of a server (as defined above) is to simplify the process of bringing the clients together, so they can interact. The server has the role of a host in a conference, for example; the participants are the clients. Once in Dave's house there are two ways in which you, Larry, Michael and Toni could take part in the proceedings. One would be for Dave to be the mediator of all communication. A star [9] topology is realized, with the server in its center, and that is the structure we have opted for in [2]: a simple (automated, for simplicity) text-based chat application. The second approach would be for each one of the participants to be able to approach (and discuss directly with) any of the other participants. Thus a *peer-to-peer* relationship is established, and the server is relegated to a background role: it only provides a location for clients to go to, so they could interact directly with all the other clients that are also present. A ring or (should we need it) a mesh (or fully-connected) architecture can be realized as seen in our example below. A third example revisits the star topology and only adds to the complexity of the clients by providing them with a GUI: we develop an applet-based text chat application with a shared graphical whiteboard to show that any of the *extras* on the client side won't have anything to do with the network (which is well packaged, set aside, insulated in the fixed part of the pattern). However many and/or complex, all client-side enhancements remain independent of the basic pattern, and won't interfere with it. (See [3] for details and many more examples).

THE EXAMPLE

In this example we try to mimic the activity on Wall Street: a server is the location/address where all dealers register, so they can buy and sell from each other. A client is any entity that, after initial registration with the server, can contact any similar entity and initiate a transaction. To simplify things a transaction essentially represents a transfer from the seller (the contacted client) to the buyer (the initiator of the transaction). This amount is random, and can be positive or negative. Since any client comes into existence with an initial amount of 0 (zero) this experiment is a zero-sum game: at any given point in time the sum of all balances across all clients should be zero. Also, as indicated, all transactions are direct transactions between the clients, without any interference on the part of the server. The only purpose of the server is to introduce the clients to each other and to check periodically that the total sum of balances is 0 (zero).

Here's the UML time diagram:

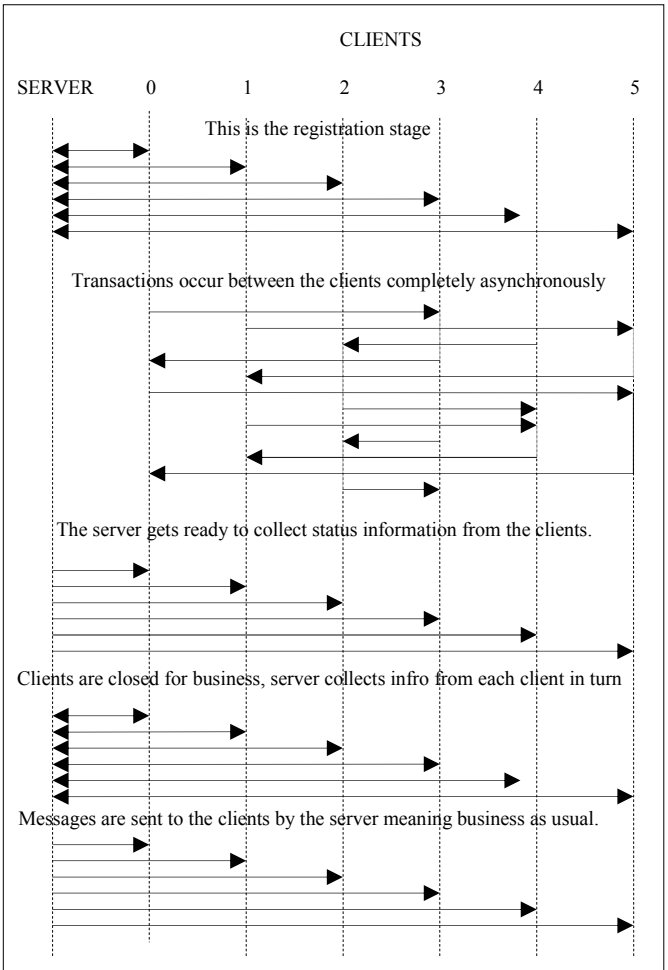


FIGURE 1
TIME DIAGRAM FOR THE EXAMPLE CONSIDERED

There are five stages in this time diagram. The first one is at the top and involves the clients coming into existence, then going through a registration phase. This phase is completely user-dependent and in this case it includes the clients' getting a list of all the other clients currently connected. As discussed a bit earlier the server's role is mostly passive.

The second stage is the most interesting and it can happen (in our example) even before the first stage is completed. In other words the second stage is completely asynchronous. In this stage every client can initiate a transfer with any other client, if both are available. Their availability disappears while the transfer is on-going (between the start of the transfer and the moment when the transfer is completed). It is this second stage that literally helps us put an identity sign between distributed and concurrent processing.

The third stage is just as asynchronous as any of the rest. The server sends messages to the clients shutting them down one by one. This happens by letting current transfers come to completion then remaining unavailable. This stage and the fifth one in the diagram serve only the purpose of protecting the stage in between, which is a verification stage.

In the fourth stage the server checks all balances and their sum and the zero-sum property of the entire set of individual negotiations is verified. Once a report is produced, the server sends messages to all clients opening them for business again.

THE RMI RECIPE

To develop an RMI-based application one needs to follow these steps, reproduced here in short, from [10]:

Create an interface that defines the exported methods that the remote object implements (that is, the methods that the server implements and that clients can invoke remotely).

Define a subclass of `java.rmi.UnicastRemoteObject` that implements the `Remote` interface. This class represents the remote object (or server object). Other than declaring its remote methods to throw `RemoteException`s, the remote object does not need to do anything special to allow its methods to be invoked remotely. The `UnicastRemoteObject` and the rest of the RMI infrastructure handle it automatically.

Write a program (a *server*) that creates an instance of your remote object. Export the object, making it available for use by clients, by registering the object by name with a registry service. This is usually done with the `java.rmi.Naming` class and the *rmiregistry* program. A server program may also act as its own registry server by using the `LocateRegistry` class and registry interface of the `java.rmi.registry` package. After you compile the server program (with `javac`) use `rmic` to generate a *stub* and *skeleton* for the remote object.

Invoke `rmic` with the name of the remote object class (not the interface) on the command line. It creates and compiles two new classes with the suffixes `_Stub` and `_Skel`. With RMI the client and the server do not communicate directly. On the client side the client's reference to a remote object is implemented as an instance of a *stub* class. When the client invokes a remote method, it is a method of this stub object that is actually called. The stub does the necessary networking to pass the invocation to a *skeleton* class on the server. This skeleton translates the networked request into a method invocation on the server object, and passes the returned value back to the stub, which passes it back to the client. This can be a complicated system but fortunately application programmers never have to think about stubs and skeletons; they are generated automatically by the *rmic* tool.

If the server uses the default registry service provided by the `Naming` class you must run the registry server, if it is not already running. You can run the registry server by invoking the *rmiregistry* program. Note however that, as mentioned above, a server program may also act as its own registry server by using the `LocateRegistry` class and registry interface of the `java.rmi.registry` package so you need to run the registry server (or make sure it is running) only if the server uses the default registry service provided by the `Naming` class.

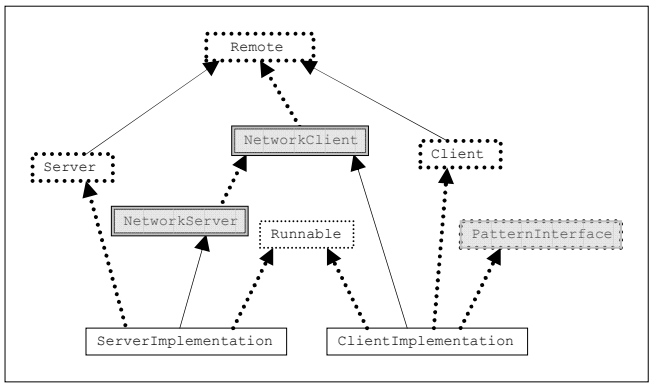
Now you can write a client program to use the remote object exported by the server. The client must first obtain a reference to the remote object exported by the server by using the `Naming` class to look up the object by name. The name is typically an `rmi://` URL. The remote remote reference that is returned is an instance of the `Remote` interface for the object (or more specifically a *stub* object for the remote object). Once this client has this remote object it can invoke methods on it exactly as it would invoke the methods of a local object.

RMI uses the serializaton mechanism to transfer the stub object from the server to the client. Because the client may load an untrusted stub object, it should have a security manager installed to prevent a malicious (or just buggy) stub from deleting files or otherwise causing harm. The `RMISecurityManager` class is a suitable security manager that all RMI clients should install.

The last stage is to start up the server, and run the client.

OBSERVING THE DISTRIBUTED PATTERN

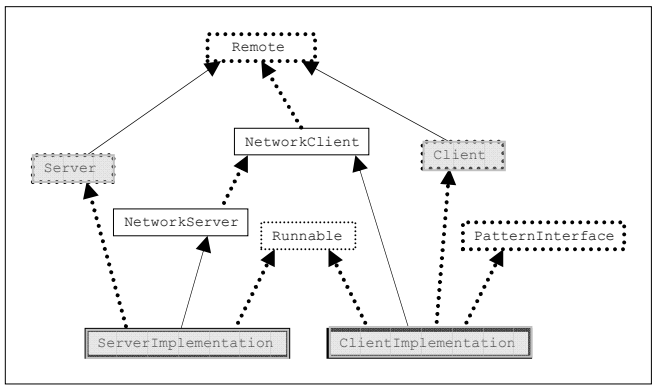
As an alternative to the guidelines mentioned above we offer an object-oriented pattern composed of two classes (that package most of the RMI recipe mentioned above) and an interface (whose purpose is to further guide the design of the distributed application). Here's what the class diagram of the relevant part of the example looks like. (The two classes and the interface provided are highlighted in light grey.)



The first benefit of this pattern is to streamline the guidelines for the developing of applications based on RMI. In

that respect the base functionality is that of a network client. A network server is a special case of a network client. The pattern interface is very much like an API with the difference that the user provides the method definitions. The pattern interface is available to any implementation that needs its functionality. In our example the server can safely ignore it.

Let's take a look now at the exact same picture in which the classes and interfaces to be provided by the user are identified, and in almost the same way. Two interfaces in the diagram (`Remote` and `Runnable`) are already provided by Java so we just assume them.



Many types of clients and servers could be defined. Each one of them must come with an interface (that acts as a sort of business card if you will) and an implementation. Everything is up to the user of the pattern, who has complete freedom to implement its application-specific code.

The only thing the user needs to keep in mind is that the parameters top the methods in the business card interfaces be `Serializable`. Now we need to clarify the two classes and one interface, as well as the structure and meaning of the pattern shown in the diagram above.

The source code is available online [3].

I. The NetworkClient Class

Any `NetworkClient` should be able to do two things: export its methods, and locate a (possibly) remote peer. Both are easy to achieveand it would be worthwhile exploring how (below):

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

public class NetworkClient implements Remote {

  public void exportMethods() throws RemoteException {
    UnicastRemoteObject.exportObject(this);
  }

  public Remote locatePeer(String peerHost,
                           int peerPort,
                           String peerName)
              throws Exception {
    return Naming.lookup("rmi://" + peerHost + ":" +
                         peerPort + "/" + peerName);
  }
}
```

II. The NetworkServer Class

A NetworkServer is a NetworkClient that knows how to start as a server. Exporting methods properly is a useful client feature.

```
import java.rmi.*;
import java.rmi.registry.*;

public class NetworkServer extends NetworkClient {
  public void startAsNetworkServer(String name, int port) {
    System.setSecurityManager(new RMISecurityManager());
    try {
      this.exportMethods();
      Registry registry = LocateRegistry.createRegistry(port);
      registry.bind(name, this);
      System.out.println("Server is ready ... ");
    } catch (Exception e) {
      System.out.println("Server error: " + e + " ... ");
    }
  }
}
```

III. The PatternInterface Interface.

The reason for this interface is to allow the user to provide application specific definitions for what it means to start as a client. Since such definitions are fairly standard at least one of the two could be defined here, which would turn the interface into an abstract class. This approach is more flexible though.

```
import java.rmi.*;

public interface PatternInterface {

  public void startAsNetworkClientOf(String peerHost,
                                     int peerPort,
                                     String peerName)
              throws Exception;
  public void startAsClientOf(Remote peer)
              throws RemoteException;

}
```

IV. The Implementation of the PatternInterface Interface.

Starting as a network client (in our example) is very basic:

```
this.exportMethods();
Remote peer = locatePeer(peerHost, peerPort, peerName);
this.startAsClientOf(peer);
```

One needs to export one's methods, then locate the peer whose client one needs to start as, and finally to delegate the rest to the second method of the interface. No distinction is made between a remote reference and a local one. The only special step we take (when starting as a network client) is that of getting a remote reference first and of exporting our methods too at the same time.

It should be emphasized that the code above (as general as one may perceive it) is still application-specific.

It is also very important to note that most of the benefits of RMI stem from using generic types (interfaces) for references to the objects. It truly becomes irrelevant what the second method looks like. For conformity though, we are providing it below:

```
public void startAsClientOf(Remote peer) throws
                                        RemoteException
{   Server far = (Server)peer;
    far.register(this);
    this.server = far;
    new Thread(this).start();
}
```

Under the circumstances the only thing that remains to be explained is the extent to which other guidelines (if any) need to influence the writing of the application-specific code.

USING THE PATTERN

The user can define any number of types (servers, clients, or simply peers). For each such type an interface should collect all methods to be made available publicly. These interfaces must extend `Remote` as indicated earlier. Any interaction between these types of objects must be described in terms of the interfaces that they implement. In our example there are essentially three parts to the server: (a) first there is a registration procedure: when a client first calls, the server assigns it a number, records both the number and a reference to the client, then registers the new client with all existing clients, establishing a fully connected network; (b) second, it runs as a thread, its `run` method forces a `broadcast` every once in while and (c) when that happens, the `broadcast`() method has three parts, so all clients are being turned off, a global report is put together out of the individual client reports by the server, then the clients are turned on to resume activity. The clients simply define a procedure for exchanging portions of their balances, randomly. All of this is described in terms of the interfaces that the user-defined types implement. No mention is made of networking or distributed environments. Finally we put together a local setup class

```
class LocalSetup {
  public static void main(String[] args)
                    throws /*Remote*/Exception
{
    ServerImplementation server = new ServerImplementation();
    server.startAsLocalServer();

    for (int i = 0; i < 6; i++) {
      ClientImplementation dealer =
        new ClientImplementation("Dealer_" + i);
      dealer.startAsClientOf(server);
    }
  }
}
```

in which clients and servers need to be started in the same way we'd be doing it in a distributed environment. Starting as a local server really means invoking `start()` as a thread but the method is provided in case additional steps need to be taken in bringing the server up (initialization and such).

This method creates the objects and introduces them to each other, then lets the application-specific code run. We can now test and debug. When we are finished the deployment programs should be placed on their hosts, stubs and skeletons should be created for the clients and the servers and the programs should be started from the line.

```
// equivalent of the local setup (individually)
public static void main(String[] args)
                    throws Exception
{
    String ownName = args[0],
           serverHostName = args[1],
           serverPortNumber = args[2],
           serverName = args[3];

    ClientImplementation
        client = new ClientImplementation(ownName);

    client.startAsNetworkClientOf(
                serverHostName,
                Integer.parseInt(serverPortNumber),
                serverName);
}
```

Here's how the server gets started as a standalone application.

```
public static void main(String[] args)
                            throws Exception {
    String portNumber = args[0], ownName = args[1];
    ServerImplementation here =
                new ServerImplementation();

    here.startAsNetworkServer(ownName,
                Integer.parseInt(portNumber));

    new Thread(here).start();
}
```

CONCLUSIONS

We have shown how using Java RMI the development complexity of a distributed application can be delegated *entirely* to a non-networked environment, in which the developer, or the student, can and should be concentrating exclusively on the application logic. When the correctness of that stage has been established the network can be added (almost) as an afterthought. Equally important is the possibility that once exposed to the power of such a pattern in action, students will apply themselves with increased interest to an in-depth study of the implementation and use of this pattern, and of others [10].

REFERENCES

[1] Downing, T, B, *Developing Distributed Java Applications. Java RMI: Remote Method Invocation,* IDG Books Worldwide, 1998.

[2] German, D, A, "The Net Worth of an Object-Oriented Pattern: Practical Implications of the Java RMI", *Proceedings of the Tenth International Conference on Parallel and Distributed Systems,* July 7-9, 2004.

[3] `http://www.cs.indiana.edu/~dgerman/fie2004.html`

[4] Felleisen and Friedman, *A Little Java, A Few Patterns*, MIT Press 1998.

[5] Gamma et. al. *Design Patterns* Addison-Wesley 1999.

[6] `http://wwws.sun.com/software/jini/faqs`

[7] `http://www.jot.fm/issues/issue_2002_11/article2`

[8] `http://trmi.sourceforge.net`

[9] Stallings, W., *Data and Computer Communication*, Macmillan (1991)

[10] Flanagan, D., *Java Enterprise in a Nutshell*, O'Reilly 2001.