THE NET WORTH OF AN OBJECT-ORIENTED PATTERN: PRACTICAL IMPLICATIONS OF THE JAVA RMI

D. A. GERMAN
Department of Computer Science, Indiana University,
150 S. Woodlawn Ave., Bloomington, IN 47405-7104, USA
E-mail: dgerman@cs.indiana.edu

This is not a tutorial or an introductory paper on Java RMI. Many excellent introductions exist and the reader is expected to have a basic understanding of Java RMI already. 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, *without any modification.* 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 in three examples. 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 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. Thus one can (and should) design, develop, and test—without any network, then switch to a distributed, networked run-time environment at the end, in a most straightforward fashion—keeping the two stages (development of application logic and deployment of the application into its run time environment) completely separate.

1 Introduction

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 is not a tutorial or an introductory paper on Java RMI. Many excellent introductions exist already and the reader is expected to have a basic understanding of Java RMI at the level of, for example, what's presented in the listed references [1-8]. This is a paper about a pattern and its implications in practice and teaching. Network programming is conspicuously absent from introductory programming courses, and it has long been perceived as an arcane, difficult, and therefore advanced topic. As this paper aims to show, this need not be the case. As soon as students understand methods, and get a basic knowledge of inheritance, interfaces and how an object can act as a semaphore when one of its `synchronized` instance methods is invoked (not a difficult thing to understand, if translated into the informal require-

ment that the object "must do only one thing at a time") a student is ready to write moderately complex distributed applications, such as the one presented at the end of this paper.

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 Gamma et al. [12], 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 errors check). 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)[a]. To attempt another analogy, the use of this pattern is similar to the ability to participate in an auction either in person, or from a distance, using a cell phone (as in a teleconference[b]). In both cases the predominant aspect is that of component-based (i.e., plug-and-play) networking, one of the declared goals of Jini [14]—a technology built on Java RMI.

Granted, sockets—designed by Bill Joy in the late seventies—are an already powerful abstraction and, at the time, had an impact of comparable significance by allowing programmers transparent access to files and network connections. But as he wrote recently [13]: " [...] *instead of extending the capability of the network by defining new protocols and having to test the many implementations of the protocols for compatibility, we create the ability to send Java implementations of protocols around the network to machines that include the Java Virtual Machine (Java VM). In this new architecture, the RMI (remote method invocation) protocol by which the Java VM exchanges agents becomes*[c] *a 'protocol to end all protocols'.*"

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

[a] 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).

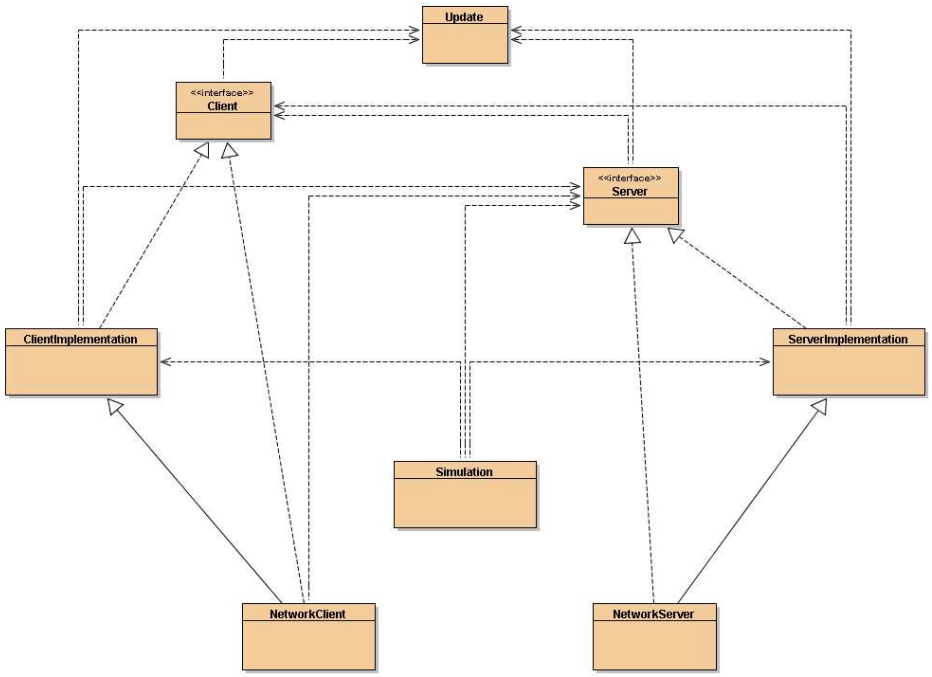
[b] In this analogy sockets would interfere with the negotiation process, lacking the transparency provided by cell phones. The sentences would have to be crafted specifically for sockets, that is, perhaps by being spoken in a different language (e.g—a secret code, to eliminate eavesdropping).

[c] We need to add that RMI is more of a language paradigm and enforces a tightly-coupled style of development unlike, for example, XML-RPC and SOAP (with cross-language goals). Especially when compared to its equally powerful but notably different alternatives, the RMI approach stands out as a uniquely elegant and powerful solution.

(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[d].

2 The Pattern

We need a pattern if we are to teach or communicate the methodology to a group of students[e]. The pattern is presented below in a simplified `BlueJ`[9]-produced UML diagram. It shows two interfaces (one for the `Client` and one for the `Server`,) a class for the implementation of each, and two class extensions—in charge of running the server and the client implementation code over the network. The non-networked application is monitored from a class (called `Simulation`) which runs the client and the server implementations locally (as `Threads`). The clients and the server communicate by passing each other `Update` objects—which are completely general, and need only be `Serializable`. The client and the server code (interfaces and implementations, and any `Update` events) are application-specific. The rest of the classes are application-independent and thus comprise the fixed part of the pattern.



[d] For related work (both RMI-based) see `CentiJ`[15] and TransparentRMI[17].
[e] We assume students have signed in to learn the features of only one language (for example, Java).

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 *clients*. 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[10] topology is realized, with the server in its center, and that is the structure of our first example: 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 second example.

The 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 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 complex, client-side enhancements remain independent of the basic pattern, and won't interfere with it.

3 The Star Topology Example

3.1 The Server Interface

When the server acts as a mediator the clients only need to know about it two things: (*a*) that it can accept registrations, and (*b*) that it can be used to broadcast a message of some sort to all the other participants:

```
import java.rmi.*;

public interface Server extends Remote {
  public int register(Client client) throws RemoteException;
  public void broadcast(Update event) throws RemoteException;
}
```

This is the server's business card.

3.2 The Client Interface

The server only needs to know about the clients that they can be updated, so the client interface looks as follows:

```
import java.rmi.*;

public interface Client extends Remote {
  public void update(Update event) throws RemoteException;
}
```

This represents a client's business card.

3.3 The Update Objects

Objects are being passed back and forth: (*a*) from clients to the server, when clients invoke `broadcast` on the server, and (*b*) from the server to the clients, when the server's `broadcast` method invokes each of the clients' `update` methods in turn.

```
import java.io.*;

public class Update implements Serializable {
  public Update(String message) {
    this.message = message;
  }
  String message;
  public String toString() {
    return this.message;
  }
}
```

Our `Update` class here is a little more than just a wrapper for a `String` but the next two examples add some complexity to this initial stage[f].

3.4 The Server Object

The implementation of the server (presented below) reveals an array of up to 100 references to `Clients`, with an `index` that keeps track of the last assigned position in the array. The registration method (which all clients call) is straightforward, while the `broadcast` method simply passes the `event` received, to the `update` methods of each of the the clients registered with this server[g].

[f] Also, through the serialization requirement we do acknowledge the eventual use of the class in a networked environment. But just as in the case of the two interfaces presented above (that extend the `java.rmi.Remote` interface) the use of these features is restricted exclusively to the deployment stage (that is, when, and if the application is run over the network) and does not interfere with anything else whatsoever.

[g] Note though that the registration procedure `returns` the `index` that has been used to store the reference to the client—it will become the `Client`'s `id`.

```
import java.rmi.*;

public class ServerImplementation implements Server {

  Client clients[] = new Client[100];

  int index = -1;

  synchronized public int register(Client client)
                          throws RemoteException {

    clients[++index] = client;
    return index;
  }

  synchronized public void broadcast(Update event)
                           throws RemoteException {

    for (int i = 0; i <= index; i++)
      clients[i].update(event);

    System.out.println("-----------------------------");
  }
}
```

Also, methods are `synchronized` so that access to the array of `Clients` is without fault. `Clients`' methods are `synchronized` too (and also on their host objects).

3.5 The Client Objects

As can be seen in the code below, clients store a reference to the server, in addition to a `String` (for their name) and an `int` for their numeric `id`.

```
import java.rmi.*;

public class ClientImplementation extends Thread
                                  implements Client {
  String name;
  int id;

  Server server;

  public ClientImplementation(String name) {
    this.name = name;
  }
```

The `update` procedure is straightforward, but (because clients are `Threads`) we need to say a few words about `run`: a random period of time the client sleeps, after which it broadcasts an `Update` object—containing the name of the client and an always the same "*says: Howdy!*" message—to the server, for the server to distribute it further to all the clients (including the client with which the message has originated).

```
    public void update(Update event) throws RemoteException {
      System.out.println(this.name +
                         " receives: ***(" + event + ")*** ");
    }

    public void run() {
      while (true) {
        try {
          sleep((int)(Math.random() * 6000 + 10000));
          server.broadcast(
            new Update(this.name + " says: Howdy!"));
        } catch (Exception e) { }
      }
    }
}
```

3.6 Testing and Debugging

We can now run the application locally[h], simulating the deployment over the network. The application-independent part of the pattern is composed of the class presented in this section (named `Simulation`) and the two that follow[i]. In other words, while the server and the client implementations, together with their interfaces (already presented in the previous sections) are determined by the application logic, and can thus vary from case to application, the three classes presented in this and the next two sections are delivered with the pattern, and must always carefully follow the structure presented here.

For the `Simulation` class that means a specific sequence of bringing up the server, and then the clients, and of passing the right type of references around. The task presented below will have to be achieved by two (not just one) `main` methods in the networked version: one `main` for the network-aware version of the client, and one for the server version, respectively.

Also, perhaps this would be a good time to remind ourselves that compiling and running the distributed application will have to include[j] `rmic` being invoked on `NetworkClient` and `NetworkServer`, for the stubs and skeletons to be created.

[h] Just compile and run `Simulation`.

[i] `NetworkServer` and `NetworkClient` that have their own `main` methods.

[j] Another thing that `Simulation` does not need is a `.java.policy` file that gives the server the permission to take incoming calls. The contents of such a file is listed at the end, in the Appendix.

```
import java.rmi.*;

public class Simulation {
  public static void main(String[] args) throws RemoteException {

    // part one: on the server's host
    ServerImplementation server = new ServerImplementation();

    // three part two's: on any of the clients' hosts
    Server far = server; // this line is factored out here

    ClientImplementation larry // first part two (on larry's host)
      = new ClientImplementation("Larry");
    larry.id = far.register(larry);
    larry.server = far;
    larry.start();

    ClientImplementation michael // second part two (on michael's)
      = new ClientImplementation("Michael");
    michael.id = far.register(michael);
    michael.server = far;
    michael.start();

    ClientImplementation toni // third part two (on toni's host)
      = new ClientImplementation("Toni");
    toni.id = far.register(toni);
    toni.server = far;
    toni.start();

  }
}
```

Compiling and running the class above completes the design, test and debug stage. Everything is running locally, outside any network. There are three threads, which sleep a random amount of time, then ask the server to broadcast a greeting. The server calls each client's `update` method with the greeting, and the method prints it to the standard output. This stage does not care that the `Update` objects are `Serializable`, that the interfaces extend `Remote` and/or that the methods never get a chance to throw the `RemoteException`s that they declare. But the key is that the potential is already built in, and that turns out to be useful in the next stage.

3.7 The Network Server

The first thing we need to notice about this class is that it extends the server implementation defined earlier. Thus the code that we designed, tested and debugged

earlier (with `Simulation`) is participating *unchanged*. The class introduces two new features:

1. a constructor (to perform a very specific RMI function. This is necessary since we already chose to extend the server implementation defined earlier and so we can't extend `UnicastRemoteObject` any longer. Fortunately exporting the object in the constructor has the exact same effect.)

2. a `main` method (needed to start the application. We must not forget that in this approach the server resides all by itself, on a specific host, where it has to be started. The `main` method serves that purpose: a security manager is set up, a network server is created, a registry is brought up on a port number specified on the command line, and the server object created a bit earlier is bound under a specific name, in this case `"Dirac"`, to the registry. Clients use this name and the port to get a hold of the server, as we shall see shortly.)

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

public class NetworkServer extends ServerImplementation
                           implements Server {

  public NetworkServer() throws RemoteException {
    UnicastRemoteObject.exportObject(this);
    System.out.println("Server being initialized... ");
  }

  public static void main(String[] args) {

    System.setSecurityManager(new RMISecurityManager());

    try {
      NetworkServer pam = new NetworkServer();

      Registry cat =
        LocateRegistry.createRegistry(Integer.parseInt(args[0]));

      cat.bind("Dirac", pam);

      System.out.println("Server is ready... ");
    } catch (Exception e) {
      System.out.println("Server error: " + e + "... ");
    }
  }
}
```

3.8 The Network Client

The network client is just as simple[k]. It relies on the client implementation that has already been defined, and only adds the two ingredients that the network server also had to provide, that is:

1. a constructor to export the object
2. a `main` method to create the client, look for the server, and register the client with the server.

The starting of a client from the command line assumes that the server's host, the port number of the server on the server's host, and the client's name are provided on the command line, and in that order.

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

public class NetworkClient extends ClientImplementation
                           implements Client {

  public NetworkClient(String name) throws RemoteException {
    super(name);
    UnicastRemoteObject.exportObject(this);
  }

  public static void main(String[] args) {
    try {
      Server far =
        (Server)Naming.lookup( // note the URL format
          "rmi://" + args[0] + ":" + args[1] + "/Dirac");

      NetworkClient here = new NetworkClient(args[2]);

      here.id = far.register(here);
      here.server = far;
      here.start();

    } catch (Exception e) {
      System.out.println("Error in client... " + e);
      e.printStackTrace();
    }

  }

}
```

[k] Not many classes couldn't be more complicated, as the saying goes. (At this point, that is.)

As indicated earlier, the purpose of the two `main` methods (one in the network client and one in the network server class) is to assume the responsibilities that `Simulation` had in the non-networked version. The reader will notice that the `main` in `Simulation` is essentially identical to the two `main` methods in the network server and the network client classes, concatenated. The sequence of steps in `Simulation`'s `main` also indicates the sequence of steps in which we need to proceed when we deploy the application on the network:

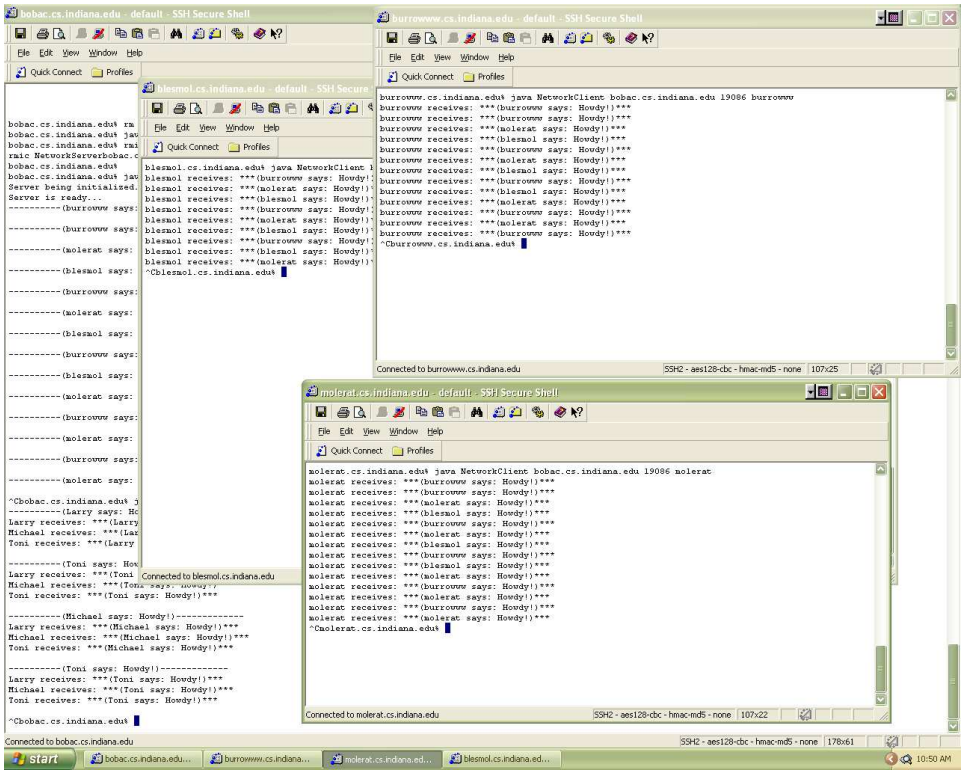
1. first we start the server (on a certain host, using a certain port)

   ```
   server.host% java NetworkServer 19086
   ```

2. then we start the clients, one by one

   ```
   larrys.host% java NetworkClient server.host 19086 Larry
   ```

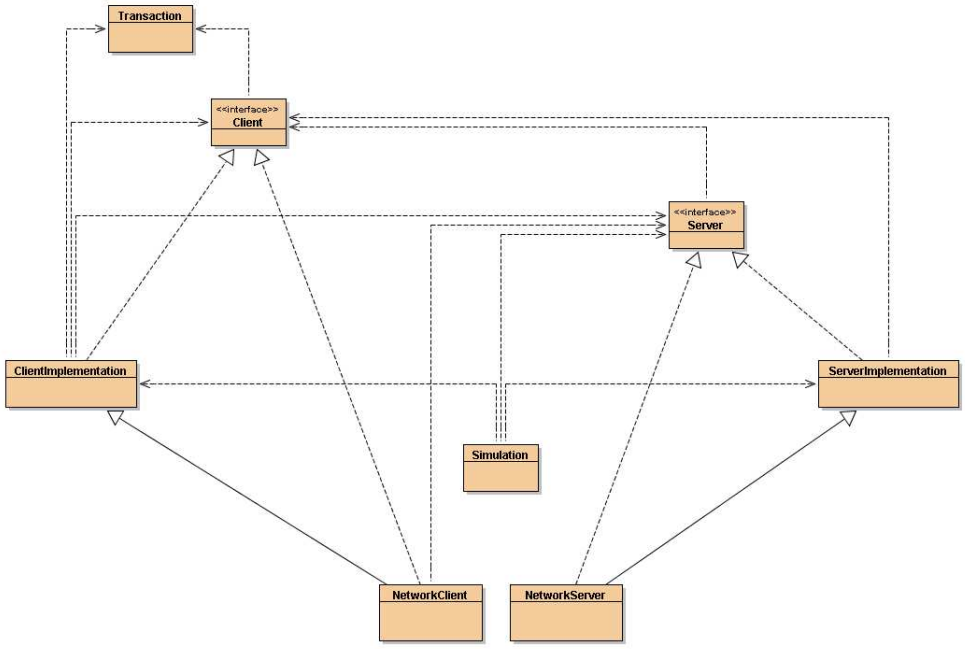
   indicating the server's host, the port number and the name of the client, in that order (for each client) on the command line.



The picture above shows the server being started on `bobac.cs.indiana.edu` on port 19086 and being accessed from `molerat`, `blesmol` and `burrowww`. One can also see `Simulation` being run by itself, in the server window (on `bobac`) right after the server and the clients have been stopped.

4 Peer to Peer Topologies

In this second 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[1]. Here's the UML class diagram of this second example.



Before we look at the code let's note the exact same components as in the previous case study: (*a*) there are two interfaces, and (*b*) two implementations (one for the server and one for the client); (*c*) there's a `Transaction` type of object that is being passed around, and (*d*) there's a `Simulation` class that runs the code locally, while (*e*) the distributed version of the code is run using a network client and a network server, just as in the previous example. We can now look at the code.

[1] 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).

4.1 The Interfaces

The server's role has diminished, as announced, but clients now have a lot more responsibility and autonomy: while the server only helps clients register, the clients now process transactions, keep track of their own peers (using their own `register` methods), must be able to set their numeric `id`, retrieve their current balance, and provide a `report`, while also responding to directions from the server (which invokes `setAvailable` on each client with a value of `false` before verifying that the entire experiment is consistent, and the global balance amounts to 0 (zero)). The `Transaction` objects are also a bit more complex now, although some of the complexity in them remains unused. Here's the server interface:

```
import java.rmi.*;

public interface Server extends Remote {
    public void register (Client client) throws RemoteException;
}
```

Here's the client interface (with all publicly available functionality listed):

```
import java.rmi.*;

public interface Client extends Remote {
    public int process (Transaction t) throws RemoteException;
    public void register (Integer i, Client c) throws RemoteException;
    public void setAvailable (Boolean b) throws RemoteException;
    public String report () throws RemoteException;
    public void setID (Integer i) throws RemoteException;
    public int getBalance () throws RemoteException;
}
```

And here's the `Transaction` object:

```
import java.io.*;

public class Transaction implements Serializable {
    String initiatorName;
    int initiatorID;
    int amount; // always positive
    String direction; // "requested", "sent"
    public Transaction(String name, int id, int amt, String dir) {
        this.initiatorName = name;
        this.initiatorID = id;
        this.amount = amt;
        this.direction = dir;
    }
}
```

4.2 The Server Implementation

```
import java.rmi.*;

public class ServerImplementation extends Thread implements Server {

    Client[] clients = new Client[100];
    int index = -1;

    synchronized public void register (Client client) throws
                                                        RemoteException {
        clients[++this.index] = client;
        client.setID (new Integer(this.index));
        for (int i = 0; i <= this.index; i++) {
            try {
                clients[i].register(new Integer(this.index), client);
            } catch (Exception e) { }
        }
    }

    synchronized public void broadcast() throws RemoteException {
        for (int i = 0; i <= this.index; i++)
            clients[i].setAvailable(new Boolean(false));
        String report = "";
        String calculation = "";
        int check = 0;
        for (int i = 0; i <= this.index; i++) {
            report += clients[i].report() + "\n";
            calculation += "(" + clients[i].getBalance() + ")..";
            check += clients[i].getBalance();
        }
        System.out.print("Server report indicates: \n" + report);
        System.out.println(calculation + " ---> " + check);
        for (int i = 0; i <= this.index; i++) {
            clients[i].setAvailable(new Boolean(true));
        }
    }

    public void run() {
        while (true) {
            try {
                sleep((int) (Math.random() * 5000 + 5000));
                this.broadcast();
            } catch (Exception e) { }
        }
    }
}
```

There are (essentially) three parts to the server:

1. First, there's the 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 (thus establishing a fully connected network).

2. Second, it runs as a `Thread`. Its `run` method forces a `broadcast` every once in a while. What happens then is explained below.

3. So, finally, every `broadcast` is composed of three parts: all clients are turned off, a global report is put together out of individual client reports and printed by the server, then clients are turned on again so they can resume activity.

4.3 The Client Implementation

One should be able to understand it by just reading the comments listed below:

```
import java.rmi.*;

public class ClientImplementation extends Thread
                                  implements Client {
    String name; // this client's name

    public ClientImplementation(String name) {
        this.name = name;
    } // this is a simple constructor

    public int getBalance() throws RemoteException {
        return this.balance;
    } // a basic accessor method

    int id; // numeric id of this client (each client is unique)
    Server server; // reference to the server
    Client[] peer = new Client[100]; // the peer network

    int balance; // the balance of this client

    Boolean available = new Boolean(true); // client on or off
    int index = -1; // last allocated entry in the array of peers

    synchronized public void setAvailable(Boolean availability)
                             throws RemoteException {
        this.available = availability;
    } // server uses setAvailable to turn the clients on and off

    public void run() {
        while (true) {
```

```
        try {
            sleep((int) (Math.random() * 1000 + 1000));
            if (this.available.booleanValue()) // active?
                this.initiateTransfer(); // defined below
        } catch (Exception e) { }
    }
} // periodically choose one of the peers to deal with

synchronized private void initiateTransfer()
                          throws RemoteException {
    if (this.index > 0) {
        int chosen = (int) (Math.random() * (this.index + 1));
        if (chosen == id // always deal with someone else
            || peer[chosen] == null) return;
        this.balance +=
          peer[chosen].process( // also described below
                        new Transaction(
                          this.name,
                          this.id,
                          (int) (Math.random() * 10 + 1),
                          Math.random() > 0.5 ? "sent" :
                                                "requested"));
    }
} // even the direction of the transfer is random

synchronized public int process (Transaction transaction)
                        throws RemoteException {
    if (this.available.booleanValue())
        if (transaction.direction.equals("sent")) {
            this.balance += transaction.amount;
            return - (transaction.amount);
        } else {
            this.balance -= transaction.amount;
            return   (transaction.amount);
        }
    else return 0; // object unavailable: method idempotent
}

public String report() throws RemoteException {
    return this.name + ": " + this.balance;
}

synchronized public void register (Integer index,
                                   Client client)
                        throws RemoteException {
    this.index = index.intValue();
```

```
        this.peer[this.index] = client;
    } // peer registration of client

    synchronized public void setID (Integer index)
                            throws RemoteException {
        this.id = index.intValue();
    }
}
```

This is the end of the application-specific part of the pattern.

4.4 Simulation

The remaining part is almost identical to what we've seen in the first example[m].

```
import java.rmi.*;

public class Simulation {
    public static void main(String[] args) throws RemoteException {

        // on the server host
        ServerImplementation server = new ServerImplementation();
        server.start(); // this part is new

        // on the clients hosts
        Server far = server;

        for (int i = 0; i < 6; i++) {
            ClientImplementation dealer
              = new ClientImplementation("Dealer_" + i);

            far.register(dealer);
            dealer.server = far;
            dealer.start();

        }
    }
}
```

4.5 The Network Server

This class (and the following) will be almost indistinguishable from the previous example. We emphasize once again the fixed aspect of this part of the pattern.

[m] One minor but noticeable difference being that the server must now be `started`, since it's a `Thread` just like the `Clients`. This however need not be the case in general (see the first example).

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

public class NetworkServer extends ServerImplementation
                           implements Server {

  public NetworkServer() throws RemoteException {
    UnicastRemoteObject.exportObject(this);
    System.out.println("Server being initialized...");
  }

  public static void main(String[] args) {
    System.setSecurityManager(new RMISecurityManager());
    try {
      NetworkServer pam = new NetworkServer();

      Registry cat =
        LocateRegistry.createRegistry(
          Integer.parseInt(args[0]));
      cat.bind("Dirac", pam);

      pam.start(); // that's the only new thing
      System.out.println("Server is ready... ");
    } catch (Exception e) {
      System.out.println("Server error: " + e + "... ");
    }
  }
}
```

4.6 The Network Client

Likewise, the network client is refreshingly unchanged:

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

public class NetworkClient extends ClientImplementation
                           implements Client {

  public NetworkClient(String name) throws RemoteException {
    super(name);
    UnicastRemoteObject.exportObject(this);
  }
```

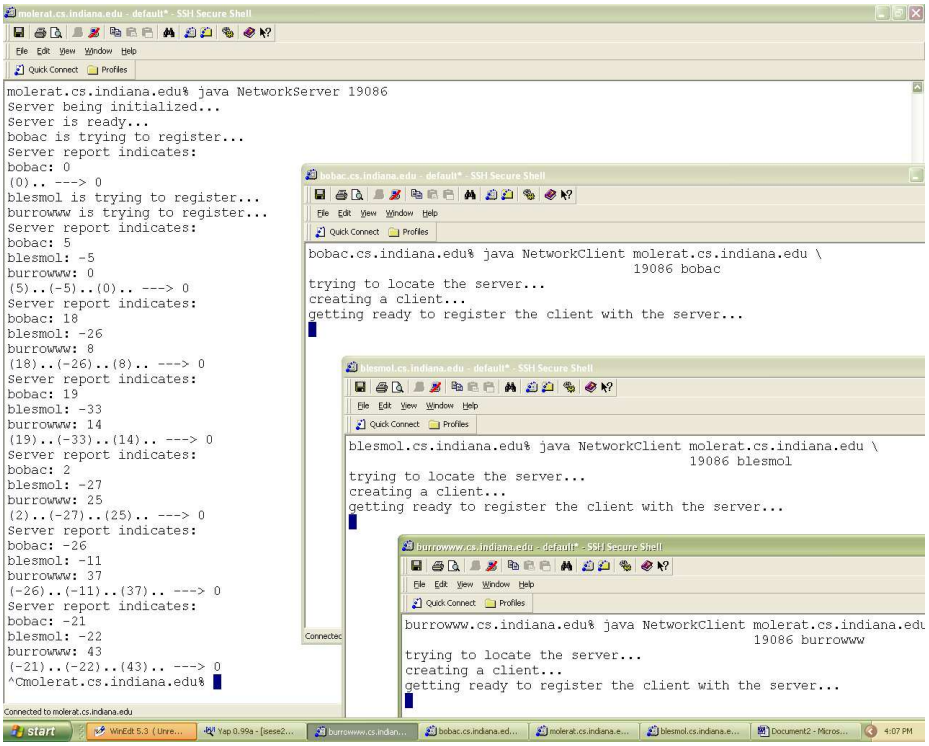
```
    public static void main(String[] args) {
      try {
        Server far =
          (Server) Naming.lookup(
            "rmi://" + args[0] + ":" + args[1] + "/Dirac");

        NetworkClient here = new NetworkClient(args[2]);

        far.register(here);
        here.server = far;
        here.start();

      } catch (Exception e) {
        System.out.println("Error in client..." + e);
      }
    }
  }
```

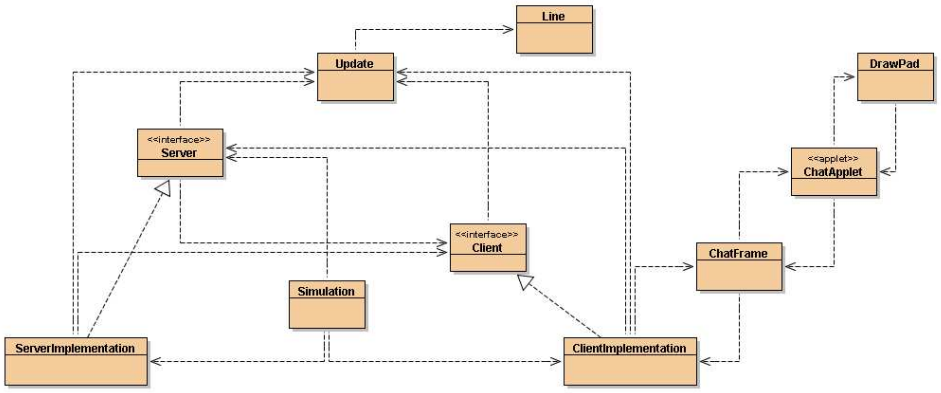
Here's a snapshot of the distributed version in action



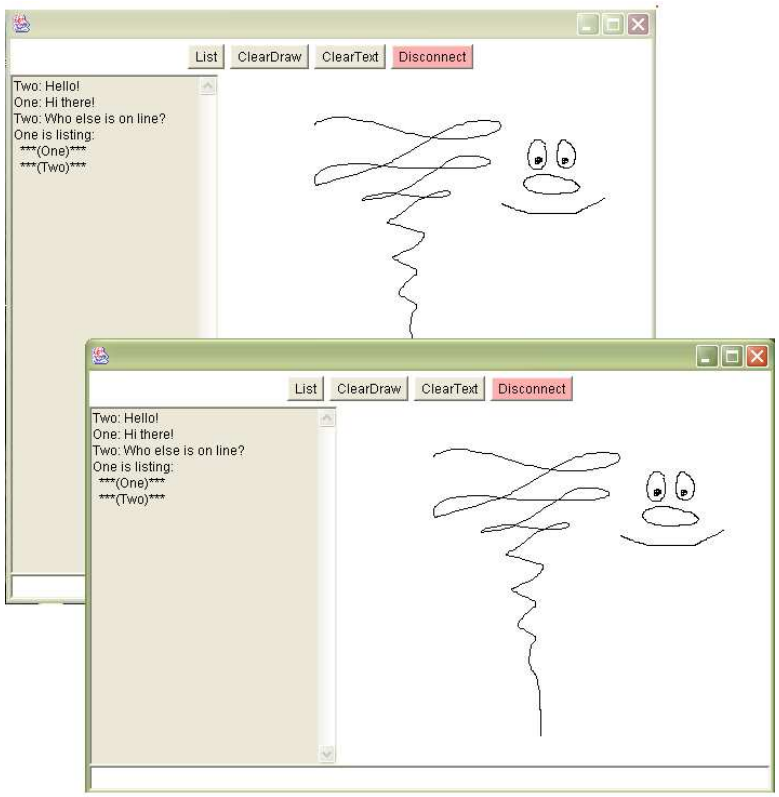
in which the server has been started on `molerat.cs.indiana.edu` on port 19086, and clients have been started on hosts `bobac`, `blesmol`, and `burrowww` respectively.

5 A Shared Virtual Whiteboard

A more involved example (adapted from Downing [4]) can be found on-line [11]. Its UML diagram is presented below[n] along with a snapshot of the program in action.



Here's a snapshot of the chat/whiteboard in action, with two clients connected:



[n] One should be thoroughly familiar with most of the diagram by now.

Conclusion

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 [16].

Appendix

Sample `.java.policy` file that can be used to test the programs presented here. It is not recommended that this file be used in a production environment since the spectrum of permissions granted by it is, by any standard, too wide.

```
burrowww.cs.indiana.edu% cat .java.policy
grant {
  permission java.security.AllPermission;
};
burrowww.cs.indiana.edu%
```

References

1. Ken Arnold, James Gosling, David Holmes, *The Java Programming Language—Third Edition.* Addison Wesley (2000)
2. Peter van der Linden, *Just Java 2 Fifth Edition*, Prentice Hall PTR (2002)
3. Horstmann, Cornell *Core Java 2: Vol. 2*, Prentice Hall PTR (2000)
4. Troy Bryan Downing, *Java RMI*, IDG Books Worldwide (1998)
5. Mark Wutka, *Special Edition Using Java 2 Enterprise Edition*, Que (2001)
6. David Flanagan, *Java in a Nutshell*, O'Reilly (2001)
7. David Flanagan, *Java Examples in a Nutshell*, O'Reilly (2001)
8. Mary Campione, Kathy Walrath, Alison Huml and the Tutorial Team, *The Java Tutorial Continued: The Rest of the JDK*, Addison-Wesley (1999).
9. `http://www.bluej.org`
10. W. Stallings, *Data and Computer Communications*, Macmillan (1991)
11. `http://www.cs.indiana.edu/~dgerman/whiteboard.html`
12. Gamma, Helm, Johnson, Vlissides. *Design Patterns.* Addison-Wesley, 1994.
13. Bill Joy, *Shift from Protocols to Agents* in *Internet Computing Online* (2001) (available at `http://www.computer.org/internet/v4n1/joy.htm`)
14. `http://wwws.sun.com/software/jini/faqs/`
15. Douglas Lyon, *CentiJ: An RMI Code Generator* in *Journal of Object Technologies* (*Nov. 2002*, at `http://www.jot.fm/issues/issue_2002_11/article2`)
16. Friedman and Felleisen, *A Little Java—A Few Patterns*, MIT Press (1998)
17. `http://trmi.sourceforge.net/`