RMI: Observing the Distributed Pattern

Dan-Adrian German
Indiana University, Bloomington

November 14, 2003

Abstract

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 (no network) and makes the switch to a true networked run-time environment completely transparent. No other approach offers this advantage[1].

The Basic Pattern

Our basic pattern of development contains the following steps:

1. design
2. implementation
3. debugging and testing, and
4. network deployment

The true power of this approach is that the network only shows up in (and need not even be mentioned before) the very last step. The prerequisite, however, is the careful use of the pattern presented in this paper.

[1] Granted, another abstraction (designed by Bill Joy in the late seventies) had an impact whose significance at the time was comparable—by allowing programmers transparent access to files and network connections. But as he wrote recently: "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 a 'protocol to end all protocols'." We need to add that RMI is a language paradigm and enforces a tightly-coupled style of development unlike, for example, XML-RPC and SOAP. Especially when compared to its equally powerful (but notably different) alternatives, the RMI approach emerges distinctly as a uniquely elegant and powerful solution.

The Practice Problem

We choose to implement a simple problem, consisting of a basic (text-based) chat application. This concentrates on the passing of information between the participants. Once this is understood, a slightly more sophisticated application (such as a multiplayer game) could be easily built, by endowing the server and the client(s) with the ability to keep a copy of the world and update it according to the rules of the game. What's very important is that this can be done later, even after deployment:

1. we disable the network,
2. add application logic,
3. test and debug, and then
4. switch back to production (distributed, run-time) mode.

This ability to separate the application logic from the specifics of the run-time environment is reminiscent of the functionality of `try/catch` blocks which can be added at the very end of an implementation stage. Let's start now by looking at the server.

The Server

Object-oriented programming is really about distributed processing and using a server from the outset is simply an indication of good design. As we see, the server implements an interface. The interface collects all the methods that the server would like to make available to its clients. In this respect the interface is more or less acting like the server's business card, and plays the role of a more pedantic type of `public` visibility modifier. Since the clients will be using callbacks they will have to provide their own interfaces in due time. Here's the server code:

```
public class Server implements ServerExports
{
    int index = -1;

    public int register( ClientExports client)
    {   clients[++index] = client;
        return index;
    }

    ClientExports clients[] = new ClientExports[100];

    public void broadcast( Update event) { }
}

public interface ServerExports
{   public int register(ClientExports client);
    public void broadcast(Update event);
}
```

A generic `Update` type of object is provided to support the exchange of information between the clients and the servers.

```
public class Update {

}
```

The Client

The client uses callbacks, and implements a required interface:

```
public class Client extends Thread implements ClientExports
{
    public void update( Update event) {

    }

    public void run() {

    }
}

public interface ClientExports {
    public void update(Update event);
}
```

Although the code is not yet complete the basic pattern is starting to emerge: the subsequent layers of development will be concentric and will involve changes in both the client and the server. More important, however, at this stage is to realize that in a distributed environment both the client and the server will be brought into being by separate `main` methods on their respective hosts, so we need to develop our `main` method in a specific way.

```
public class Simulation
{   public static void main(String[] args)
    {   // on the server's host (server starts first)
        Server server = new Server();
        // on any of the clients' hosts
        Client adrian = new Client("Adrian");
        Client raja = new Client("Raja");
        Client dijkstra = new Client("Edsger");

        adrian.start();
        raja.start();
        dijkstra.start();
    }
}
```

We can add some functionality to the clients code now:

```
public class Client extends Thread implements ClientExports
{   String name;

    public Client(String name) { this.name = name; }

    public void update(Update event) { }

    public void run()
    {
        while (true)
        {   try { sleep((int)(Math.random() * 6000 + 1000)); }
            catch (Exception e) { }
            System.out.println(this.name + " here...");
        }
    }
}
```

Now we need for the clients to start sharing the server; it will be their communication medium. This is where portions of the server need to be `synchronized`. Here's the updated code:

```
public class Simulation
{
    public static void main(String[] args)
    {   // on the server's host
        Server server = new Server();

        // on any of the clients' hosts
        ServerExports far = server;

        Client adrian = new Client("Adrian");
        adrian.id = far.register(adrian);
        adrian.server = far;
        adrian.start();

        Client raja = new Client("Raja");
        raja.id = far.register(raja);
        raja.server = far;
        raja.start();

        Client dijkstra = new Client("Edsger");
        dijkstra.id = far.register(dijkstra);
        dijkstra.server = far;
        dijkstra.start();

    }
}
```

Note that all the classes change a little bit[2]. First the events change structure, as they now carry a message:

```
public class Update
{
    String message;

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

Clients start using the server for broadcasting[3]:

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

    ServerExports server;

    public Client(String name)
    {
        this.name = name;
    }

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

    public void run()
    {
        while (true)
        {
            try { sleep((int)(Math.random() * 6000 + 10000)); }
            catch (Exception e) { }

            server.broadcast(
              new Update(this.name + " says: Howdy!"));

        }
    }
}
```

And continuing the circular pattern of development mentioned before, we need to add the server's `broadcast` method, which is extremely simple:

[2]Code updates are distributed and synchronized.
[3]And expect to be updated (on what they need to show) by the server.

```
public class Server implements ServerExports {

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

    synchronized public int register(ClientExports client) {
        clients[++index] = client;
        return index;
    }

    synchronized public void broadcast(Update event) {
        for (int i = 0; i <= index; i++)
            clients[i].update(event);
    }
}
```

Here's how the program runs:

```
frilled.cs.indiana.edu%javac *.java
frilled.cs.indiana.edu%java Simulation

Adrian receives: ***(Adrian says: Howdy!)***
Raja receives: ***(Adrian says: Howdy!)***
Edsger receives: ***(Adrian says: Howdy!)***

Adrian receives: ***(Edsger says: Howdy!)***
Raja receives: ***(Edsger says: Howdy!)***
Edsger receives: ***(Edsger says: Howdy!)***

Adrian receives: ***(Raja says: Howdy!)***
Raja receives: ***(Raja says: Howdy!)***
Edsger receives: ***(Raja says: Howdy!)***

Adrian receives: ***(Edsger says: Howdy!)***
Raja receives: ***(Edsger says: Howdy!)***
Edsger receives: ***(Edsger says: Howdy!)***

Adrian receives: ***(Adrian says: Howdy!)***
Raja receives: ***(Adrian says: Howdy!)***
Edsger receives: ***(Adrian says: Howdy!)***

Adrian receives: ***(Raja says: Howdy!)***
Raja receives: ***(Raja says: Howdy!)***
Edsger receives: ***(Raja says: Howdy!)***

Adrian receives: ***(Raja says: Howdy!)***
Raja receives: ***(Raja says: Howdy!)***
Edsger receives: ***(Raja says: Howdy!)***

Adrian receives: ***(Edsger says: Howdy!)***
```

```
Raja receives: ***(Edsger says: Howdy!)***
Edsger receives: ***(Edsger says: Howdy!)***

Adrian receives: ***(Adrian says: Howdy!)***
Raja receives: ***(Adrian says: Howdy!)***
Edsger receives: ***(Adrian says: Howdy!)***
^Cfrilled.cs.indiana.edu%
```

We're done. Now we can switch to a truly distributed run-time environment and we can do that without much thinking. The only significant change is in the `main` method which has to split into two `main`'s (one for each host) that also need to perform some RMI specific initializations. We also need to use `rmic` to create the stubs and the skeletons. But everything that we have developed thus far remains virtually unchanged. There are minor changes in the interfaces extending `Remote` now and such but these are truly minor changes as can be seen in the next section.

Deployment

Here's the first round of changes[4]:

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

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

    ServerExports server;

    public Client(String name) { this.name = name; }

    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) { }
        }
    }
```

[4]The reader is invited to *check* the differences between this and the previous stage.

```
    public static void main(String[] args) {

    }
}
```

The client interface becomes:

```
import java.rmi.*;

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

Here are the changes in the server file:

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

public class Server extends UnicastRemoteObject
                   implements ServerExports
{
    ClientExports clients[] = new ClientExports[100];
    int index = -1;

    synchronized public int register(ClientExports 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);
    }

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

    public static void main(String[] args) {

    }
}
```

And here is the current server interface:

```
import java.rmi.*;

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

Here's the change in compilation steps.

```
frilled.cs.indiana.edu%ls -ld *.java
-rw-------   1 dgerman  faculty       775 Nov 14 21:55 Client.java
-rw-------   1 dgerman  faculty       132 Nov 14 21:50 ClientExports.java
-rw-------   1 dgerman  faculty       635 Nov 14 21:53 Server.java
-rw-------   1 dgerman  faculty       205 Nov 14 21:51 ServerExports.java
-rw-------   1 dgerman  faculty       172 Nov 14 21:47 Update.java
frilled.cs.indiana.edu%javac *.java
frilled.cs.indiana.edu% rmic Server
frilled.cs.indiana.edu% rmic Client
frilled.cs.indiana.edu%ls -ld *.class
-rw-------   1 dgerman  faculty      1242 Nov 14 21:58 Client.class
-rw-------   1 dgerman  faculty       214 Nov 14 21:58 ClientExports.class
-rw-------   1 dgerman  faculty      1593 Nov 14 21:58 Client_Skel.class
-rw-------   1 dgerman  faculty      2877 Nov 14 21:58 Client_Stub.class
-rw-------   1 dgerman  faculty       679 Nov 14 21:58 Server.class
-rw-------   1 dgerman  faculty       267 Nov 14 21:58 ServerExports.class
-rw-------   1 dgerman  faculty      1937 Nov 14 21:58 Server_Skel.class
-rw-------   1 dgerman  faculty      3613 Nov 14 21:58 Server_Stub.class
-rw-------   1 dgerman  faculty       349 Nov 14 21:58 Update.class
frilled.cs.indiana.edu%
```

Now the finishing touches. The server's `main` becomes:

```
public static void main(String[] args) {

    System.setSecurityManager(new RMISecurityManager());

    try
    {
        Server pam = new Server();
        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 + "... ");
    }

}
```

And here's what the client `main` method becomes:

```
public static void main(String[] args) {

    try {
        ServerExports far =
            (ServerExports)Naming.lookup(
                "rmi://" + args[0] + ":" + args[1] + "/Dirac");

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

        UnicastRemoteObject.exportObject(here);

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

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

}
```

Here's how the program runs. First the server:

```
frilled.cs.indiana.edu%
frilled.cs.indiana.edu%java Server 19801
Server being initialized...
Server is ready...
```

Next, the clients. This one's on `burrowww`:

```
burrowww.cs.indiana.edu% java Client frilled.cs.indiana.edu 19801 Adrian
Adrian receives: ***(Adrian says: Howdy!)***
Adrian receives: ***(Adrian says: Howdy!)***
Adrian receives: ***(Adrian says: Howdy!)***
Adrian receives: ***(Adrian says: Howdy!)***
Adrian receives: ***(Adrian says: Howdy!)***

Adrian receives: ***(Adrian says: Howdy!)***
Adrian receives: ***(Dijkstra says: Howdy!)***
Adrian receives: ***(Adrian says: Howdy!)***
Adrian receives: ***(Dijkstra says: Howdy!)***
```

This client was started a bit later (on `tucotuco`):

```
tucotuco.cs.indiana.edu% java Client frilled.cs.indiana.edu 19801 Dijkstra
Dijkstra receives: ***(Adrian says: Howdy!)***
Dijkstra receives: ***(Dijkstra says: Howdy!)***
Dijkstra receives: ***(Adrian says: Howdy!)***
Dijkstra receives: ***(Dijkstra says: Howdy!)***
```

Truly important change:

```
import java.io.*;

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

Truly important file (to be used for testing purposes only):

```
frilled.cs.indiana.edu% ls -ld .java*
-rw-------   1 dgerman  faculty        56 Nov 14 22:49 .java.policy
frilled.cs.indiana.edu% cat .java.policy
grant {
    permission java.security.AllPermission;
};
frilled.cs.indiana.edu%
```

Conclusion

We have shown how using RMI the development complexity of a distributed application can be delegated entirely to a non-networked environment, in which the developer can (and should be) concentrating exclusively on the application logic. The power of an abstraction pattern resides in its ability to guide the development from start to finish. In some sense the meaning of the development pattern presented here is that if one wants to write a distributed application, one has to be able first to think about it properly and develop it in its entirety in a non-networked environment. The transparency provided by this methodology is a powerful reminder that (as the saying goes) "the network *is* the computer".

Acknowledgements

We thank Raja Sooriamurthi for a careful reading of an earlier version of this paper and for providing thoughtful comments on that manuscript.