OBJECTS AND RMI

James W. Cooper

Java' Remote Method Invocation (RMI) facility is one of the simplest and most elegant ways to write a client-server system. It not only allows you to get numbers or text from the server, you can exchange *objects* and do so without having any deep understanding of how objects are transmitted. Before we talk about how we can use and arrange these objects effectively, let's review how RMI works.

You design and implement an RMI system by carrying out the following steps.

1. Write an *interface* that describes what methods your RMI server object will provide.
2. Then you write the implementation of the server object that *implements* that interface.
3. On the server, you create a main program that registers and launches an instance of the server object.
4. To create a stub and skeleton files to represent the server object on the client, you run the remote object compiler `rmic`.
5. Copy the stub and interface files to your client machine.
6. Write the client application using the interface file to define the classes on the object server.
7. Start the `rmiregistry` program on the server
8. Start the remote object server program on the server.
9. Run the client program, telling it to connect to that particular server.

A Simple Sales Information System RMI Server

Let's consider a server system which contains customer information. You can ask the system to search for any customer by name, and display customer information. Then, you can ask the system to display details about the customers past orders. So let's start by designing the interface:

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

public interface CustomerServer extends Remote{
  public Vector getCustomers(String mask) throws RemoteException;
  public Vector getOrders(int customerKey) throws RemoteException;
}
```

Note that the server must extend the *Remote* interface and each method must throw a RemoteException.

The server itself is pretty simple as well. Listing 1 shows an outline of such a server:

```
public class RMIServer_Impl  extends UnicastRemoteObject
       implements CustomerServer {
    Database db;
    String   dbName;

    public RMIServer_Impl(String name) throws RemoteException {
        super();
        dbName = name;
```

```
try {
    Naming.rebind(name, this);
 } catch (Exception e) {
    System.out.println("rebind exception:"+e.getMessage());
 }
 db = new Database("sun.jdbc.odbc.JdbcOdbcDriver");
}
//-----------------------------------
public Vector getCustomers(String name) throws RemoteException {
   Vector cust = new Vector();
//***fill vector with Customer objects from a database query***
  return cust;
}
//-----------------------------------
public Vector getOrders(int key) throws RemoteException {
   Vector orders =  new Vector();
//***fill vector with orders from database query***
 return orders;
 }
```

Note in our server, we subclass UnicastRemoteObject and implement the methods of our CustomerServer interface. We assume that the *getCustomers* method performs some sort of database lookup and returns a set of Customer objects matching the search mask. We also assume that given a particular customer key, the *getOrders* method returns a list of orders for that customer using a second database query.

In the constructor of this server object, we use the static *rebind* method of the *Naming* class to register the name of the object with the RMI server. To launch this server, we create a master program that creates an instance of the server object.

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

public class RMIServer {

    public RMIServer() {
        // Create and install the security manager
        System.setSecurityManager(new RMISecurityManager());
        try {
            // Create RMIServer_Impl
            RMIServer_Impl server = new RMIServer_Impl("RMIServer");
            } catch (Exception e) {
            System.out.println("Exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
    //---------------------------------------
    public static void main(String args[]) {
        new RMIServer();
    }
}
```

In Java 2, we can't just start this program directly using a Java JVM. We must also set the policy to allow the program to access sockets to trasnfer data via RMI. Policy files can be very finely grained in the permissions they grant. On the other hand, a very simple policy file for RMI can just be the following file: *test.policy*:

```
grant  {
permission java.security.AllPermission;
};
```

To start the RMI server, you first issue the command

```
start rmiregistry
```

from the directory where the server is located. Then you can start the server using the following simple batch file *startserver.bat*:

```
java -Djava.security.policy=test.policy RMIServer
```

The Sales Information RMI Client

The client program must connect to the RMI server and get a local representation of the server object. One common way to create this connection is to use a single class to connect to the server and have it make all the calls to the server object. Here we call it the DataModel class:

```
public class DataModel {
    private CustomerServer server;

    public DataModel(String rmiName) {
      try {
        server = (CustomerServer)Naming.lookup(rmiName);
       } catch (Exception e) {
           System.err.println("System Exception" + e);
      }
    }
//-----------------------------------------------
    //search for customers matching name mask
    public Vector getCustomers(String searchMask) {
        Vector cust =  new Vector();
        try {
         cust = server.getCustomers(searchMask);
       }
       catch (RemoteException e) {
           System.out.println("rmi connection failure");
       }
       return cust;
    }
    //---------------------------------------
    public Vector getOrders(Customer cust) {
        Vector orders = new Vector();
        try {
            orders = server.getOrders (cust.getKey ());
        }
        catch (RemoteException e) {
           System.out.println("rmi connection failure");
       }
      return orders;
    }
```

We can see the client program that carries out this simple search in Figure 1.

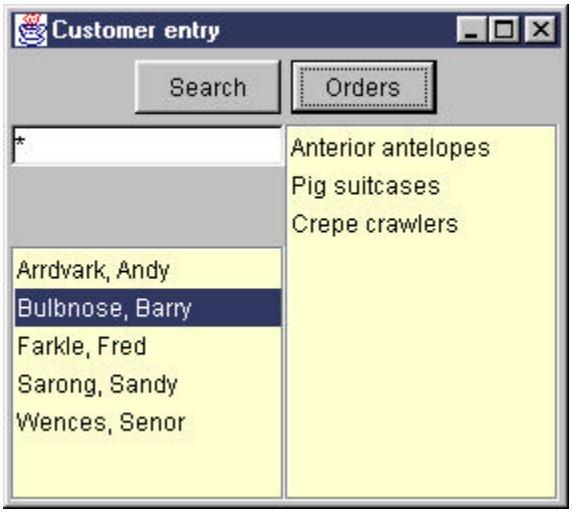
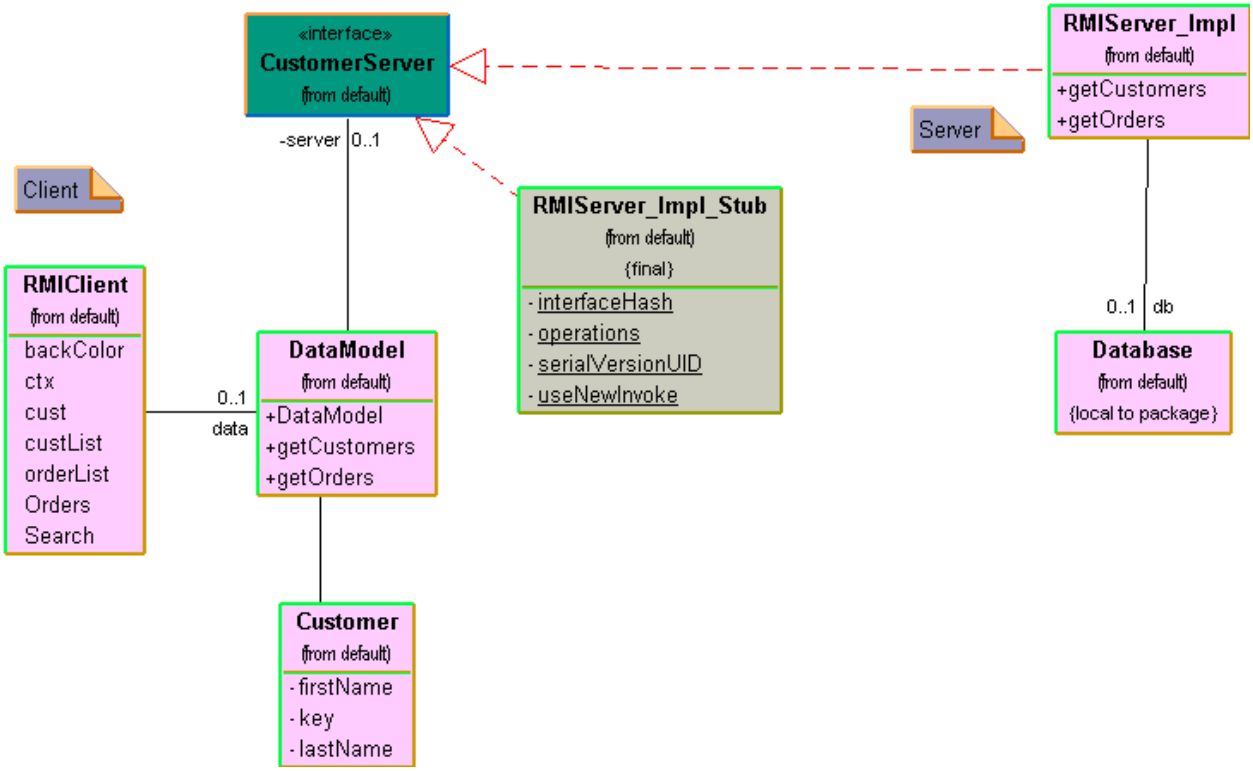


Figure 1: A simple client to our RMIServer

The complete Client-server object relation is shown in Figure 2.



Note that the Server side contains the RMIServer_Impl class which implements the CustomerServer interface and makes use of the Database class for the data lookup. The Clinet side consists of the RMISerevr_Impl_Stub which also implements the CustomerServer and acts as as stand-in for the real RMIServer_Impl class on the server.

Classes Between Clients and Servers

In the above discussion, we created and used a Customer object which contains at the least, the name and database key for each customer:

```
import java.util.*;

public class Customer {
  private String lastName;    //names
  private String firstName;
  private int key;            //database key

  public Customer(String s) implements Serializable{
    //parse customer data from an input string
    StringTokenizer tok = new StringTokenizer(s);
    key = new Integer(tok.nextToken ()).intValue();
    firstName = tok.nextToken();
    lastName = tok.nextToken();
  }
  //----------------------
  public int getKey() {
      return key;
  }
  //----------------------
  public String getName() {
      return lastName +", "+firstName;
  }
}
```

Any object that we want to transmit using RMI must implement the Serializable interface. This interface has no methods but simply sets a flag that allows the RMI system to break down the object for serial transmission between systems.

In our first program, we fetched the Customer objects, asked each one for its key and made a separate call back to the server to get the list of orders. When each Customer object is returned from the server, it does not itself contain any of the order data, although that data should logically be exclusively part of each Customer's private information. Rather than asking our client to retrieve data, we should ask the Customer object to retrieve it.

Let's assume that the server obtains the customer list from a database:

```
public Vector getCustomers(String name) throws RemoteException {
       return db.getCustomers (name);
    }
```

The database object will make the query, and create each Customer object. But if each Customer object is to keep its own orders, it will have to ask for them when it is created. Thus, each Customer must keep a reference to the Database object:

```
public class Customer implements Serializable {
  Database db;
  Vector orders;
// other variables and methods omitted
  public Customer(String s, Database datab) {
    db = datab;
    orders = db.getOrders (key);
  }
//----etc. -----
}
```

This is inconvenient for two reasons: the Database object may not be Serializable and thus can't be transmitted to the client, and second, the client system needs to have a copy of the Database object code so it can compile without error. This certainly breaks the kind of client-server encapsulation we hoped to enforce when we started out. Further, if there are a substantial number

of customers to return, each having a fair number of orders, having each customer cache the orders before returning from the server will have dire performance implications.

Another approach – Making the Customer Smarter

Well, suppose that the Customer object only looks up the orders when asked to, making the server query only when needed. We could make the getOrders method in the Customer object make a call to the DataModel object on the client to get the order Vector from the server

```
public Vector getOrders(DataModel dmodel) {
      return dmodel.orders(key);
  }
```

This suffers from the same problems: now the Customer object has to know about a client-specific class, and this information would have to be part of the server system as well so that the Customer object would compile. Again, this breaks the idea of encapsulation.

Here then is the problem we need to solve:

1. Customer objects should be the only ones to know about orders for that customer.
2. The client should not need to know about databases.
3. The server should not need to know about the client's DataModel connection to the server.

How do we solve these problems? Well, the solution is the same no matter which approach you take. You either create a dbCustomer object which does the queries and exudes a normal Customer object, or you create a modelCustomer object which uses the Customer object as an argument, and encapsulates that object inside one that does know about the DataModel object.

In the case where we want the server to look up all the orders ahead of time and return them in each Customer object, we create a dbCustomer class to look up the orders, but to return a Customer object:

```
public dbCustomer(String s, Database datab) {
    StringTokenizer tok = new StringTokenizer(s);
    key = new Integer(tok.nextToken ()).intValue();
    firstName = tok.nextToken();
    lastName = tok.nextToken();
    db = datab;
    orders = db.getOrders (key);
  }
  public Customer getCustomer() {
      return new Customer(firstName, lastName, key, orders);
  }
```

In the case where the client creates the new class, we have a class called modelCustomer which encapsulates Customer

```
public class modelCustomer {
    private DataModel dmodel;   //data model
    //encapsulate customer object
    private Customer cust;
    public modelCustomer(Customer c, DataModel data) {
        dmodel = data;      //copy in data
        cust = c;           //and customer
    }
    public String getName() {
        return cust.getName ();
    }
    //use data model reference to get the orders
```

```
    public Vector getOrders() {
        return dmodel.getOrders (cust);
    }
}
```

When we fetch the customer Vector from the server in the DataModel class, we create a new vector of modelCustomer object which then knows how to fetch their order information when needed:

```
    public Vector getCustomers(String searchMask) {
        Vector cust =  new Vector();
        try {
         cust = server.getCustomers(searchMask);
       }
       catch (RemoteException e) {
           System.out.println("rmi connection failure");
       }
       Vector modCust = new Vector();
       for(int i=0; i< cust.size(); i++) {
           Customer c = (Customer)cust.elementAt (i);
           modCust.addElement (new modelCustomer(c, this));
       }
       return modCust;
    }
```

Summary of Object Relations

We started with a DataModel that fetched data external to out Customer object instead of having the Customer be the only object that knows about its data. We considered having the order information obtained at customer lookup time or having it obtained when the client requests it. In both cases, we found that client objects needed to know about server objects or vice-versa. TO solve this, we created a dbCustomer object which is a Factory for an actual Customer object or a modelCustomer object which contains or encapsulates the basic Customer object. In both cases, the barrier between client and server operations is preserved.

Note on the Example Programs

Since many users will not find it convenient to try these examples on separate client and server machines, we have modified the Database and RMIServer_Impl classes to operate locally without using RMI if the database name is “local.” In that case, you include the RMIServer_Impl class in the same directory as the client classes and it is used directly.