Introduction to RMI

Michael B. Spring
Department of Information Science and Telecommunications
University of Pittsburgh
spring@imap.pitt.edu
http://www.sis.pitt.edu/~spring

RMI Overview

- Origin and background on RMI
- RMI paradigm
- RMI Classes
- Defining remote interfaces
- Building an RMI server
- Building an RMI client
- Running the rmiregistry
- Running the server
- Connecting with the client
- An Example
- An Exercise

Background(1)

- In the beginning client server programmers had to manage both the specific packets being exchanged and the data contents of those packets.
  - TCP allowed programmers to ignore the individual packets and focus on the dialog and messages.
  - XDR – the eXternal Data Representation (XDR) API developed by SUN allowed for the transparent interchange of integers, floats, and other complex data types with automatic conversions

Background (2)

- Even with data management, client server coding required rendezvous management – what port, and provided an awkward pipe based programming style. Both of these problems were addressed via a library API called Remote Procedure Call (RPC)
  - Environments supporting RPC run a portmapper service. The portmapper service allows servers to register their existence and gives clients a single access point.
  - XDR is automatically used for “data marshalling”
  - A utility called RPCgen allows an interface to be written and interpreted such that the programmer can code in a way that allows “remote procedures” to be invoked on the client and implemented on the server.

RMI Basic Features

- Remote Method Invocation (RMI) is Java equivalent of RPC, it allows development of clients and servers in an object oriented rather than communications oriented fashion. The server is simply a remote object.
- RMI also addresses the vagaries of different native data formats by providing an standard definition across machines – it manages the marshalling.
- Finally, RMI assumes a remote method registry that allows all clients to connect to a single port – the rmiregistry port and then be redirected as appropriate to the specific server which is running on a port that has previously been registered.

RMI paradigm

- Develop any interface classes needed to define the data interface between the client and server – extending Remote
- Define a server using the interface. The server extends UnicastRemoteObjectshould and implements the interface.
- Use rmic to compile a stub class for the server
- Define a client that uses the interface by remote interface reference. (The server stub class for the interface needs to be available to the client)
- Use rmiregistry to manage communications
- Run the server – which will register the rmiregistry
- Run the client which will check with the registry, use the stub to marshall data to and from the remote methods

RMI Classes (1)

- There are several classes and interfaces of interest to programming with RMI
  - The java.rmi package provides access to the basic interface used by objects, i.e. Remote. It also provides access to the Naming Class used to register with the rmiregistry (bind, unbind, and rebind) and to check for registered services (lookup and list).
  - The java.rmi package also provides details on most of the exceptions thrown by remote objects.
  - The java.rmi package provides the RMISecurityManager class which is used to protect the client in automated downloads of interface and implementation classes.
  - The java.rmi.server package provides access to the UnicastRemoteObject which is the base class from which almost all servers are developed

RMI Classes (2)

- The Remote interface from java.rmi is the base class from which all RMI interfaces are defined.

```
public interface myinterface extends Remote{
        Some method definition}
```

- The UnicastRemoteObject from java.rmi.server is the basis for standard TCP RMI server implementations. All servers extend this class implementing the interface which defines the data interchange

```
public class myinterfaceImpl extends UnicastRemoteObject
        Implements myinterface{
            Some class definition including a main()
            }
```

RMI Classes (3)

- The Naming class from java.rmi provides the basic method used in main by which the server registers
  - static void bind(String name, Remote obj) binds the specified name to the specified object.
    - The name is a URL and a user selected service name
    - The object is the remote server
  - The function can throw four exceptions:
    - The AlreadyBoundException says the registry already had a binding for the given name
    - The MalformedURLException says the string is not a valid URL
    - The RemoteException says the registry could not be contacted
    - The AccessException says the caller (the server) is not allowed to access the remote object
  - The already bound exception can be avoided by using
    - static void rebind(String name, Remote obj)

RMI Classes (4)

- The client uses the naming class method lookup() from java.rmi to locate the remote server and then invokes the appropriate method returning a structure consistent with the interface class file.
  - String servobjname= “rmi://”+ip+”/”+servicename
  - Classname x=(Classname) Naming.lookup(servobjname);
  - Result = x.method()
- It is important to note that the client will use the server stub to marshall data to be sent to the remote object and to collect and return data from the remote object.

RMI Classes (5)

- As you will soon see, it is necessary to move copies of some classes from the server to the client.
- It would be nice to have this happen automatically, but when this is done, there is an issue of security.
  - The RMISecurityManger class from java.rmi provides assistance in this process
  - The RMISecurityManger requires that a client.policy file be created that defines what kinds of connections are allowable for the client.

Conventions

- A client server pair will ultimately reside on different machines. Developing the two applications in different directories helps to insure that all the right pieces moved to all the right places
- By convention, the remote interface defines the primary name for the components
  - Given an interface called “Office”
  - The server program would be called OfficeServer.java
  - The class implementing the interface would be called OfficeImpl.java
  - The client program calling the methods would be OfficeClient.java
  - The stub produced by rmic will be called OfficeImpl_Stub.java

Defining remote interfaces

- Defining a remote interface is a matter of defining the methods along with the parameters and returns.
- Use the keyword interface and extend remote.
- Be sure to indicate that it throws a RemoteException
- For example:

```
// Time.java  (interface)
import java.rmi.*;
public interface Time extends Remote{
  public String getSTime()
   throws RemoteException;
}
```

Building an RMI server(1)

- Assume the interface already defined is used
- An implementation is built that provides the method definitions

```
public class TimeImpl extends UnicastRemoteObject
  implements Time
{private String stime;
 public TimeImpl() throws RemoteException
      {try
            {Date SD = new Date();
            stime = SD.toString();
            System.err.println(stime);}
      catch(Exception e)
            {e.printStackTrace();
            System.exit(1);}}
  public String getSTime() throws RemoteException
      {return stime;}
}
```

- Define a server that binds the object to a name

Building an RMI server(2)

- The server proper includes a main

```
public static void main (String args[])
   {try{ System.err.println("Starting
  Server...");
         TimeImpl tmp = new TimeImpl();
         String RMIObj = "//localhost/TS";
         Naming.rebind(RMIObj, tmp);
         System.err.println("...Server Started");
         }
   catch (Exception e)
         {
         System.out.println(“Server:" +e);
         }
   }
```

Building an RMI client

- The RMI client assumes the existence of the Impl_Stub class and the Interface Class
- These may be dynamically downloaded or manually moved to the location of the client
- With these two pieces in place, the client simply looks up and then invokes the remote method

```
 try
    {String RMethod = "//"+ip+"/TS";
    Time tmp = (Time) Naming.lookup(RMethod);
    String t = tmp.getSTime();   }
 catch (java.rmi.ConnectException ce)
    {System.err.println("connection refused");
    System.exit(1);}
```

- Note the exception catch

Putting the Piece Together

- Compile the Interface, the Implementation of the interface, and the server using javac
- Use the RMI compiler (rmic) to create the implementation stub. Assuming JDK1.2,

  rmic –v1.2 xyzImpl

  - Where xyzImpl is the particular implementation class file
- Move copies of the xyz class file and the xyzImpl_stub class file to the client directory
- Compile the client using javac

Running the server/client

- When the server is run, the first thing it will do is connect to the rmiregistry, so before running the server, start the rmiregistry program.
- Start it in the directory where the server is positioned with its classes.
  - start rmiregistry
- The start command in windows will run rmiregistry in a separate DOS window
- Start the server either from within freejava (or other IDE) or at the DOS prompt
- Run the client which will connect and invoke the remote method

An Example and Test

- Try all of the steps above on the example provided.
- Test it a couple of ways to assure yourself it is working
  - Try to run the server without running rmiregistry
  - Run the client without running rmiregistry
  - Run rmiregistry, and then run the client without running the server
  - Finally, try everything in the correct sequence

An Exercise

- Redefine the interface in the example provided to provide the time in two different forms.
- Add a method in the interface to set a variable on the server – try something simple like adding a name that says who the person providing the time service is.
- Keep in mind that the changes you make in the interface definition will have to be reflected in the implementation
- Also keep in mind that both of these files have to be moved to the client directory and the client will need to be recompiled