Sockets and RMI

CS151

Chris Pollett

Dec. 5, 2005.

Outline

- Echo Server with Multiple Clients
- Client pull/Server push
- Remote Method Invocation
- Proxy Pattern

Echo Server with Multiple Clients

```
public class MultiEchoServer
{
    public static void main(String[] args)
    {
        try
        {
            ServerScocket s= new ServerSocket(8009);
            while(true){Socket incoming = s.accept();
                new ClientHandler(incoming).start();}
        }
        catch(Exception e){}
    }
}
```

ClientHandler

```
import java.io.*;
import java.net.*;
public class ClientHandler extends Thread
{
        protected Socket incoming;
        public ClientHandler(Socket incoming)
            {this.incoming =incoming;}
        public void run()
        {
            try
            {
               BufferedReader in = new BufferedReader(new
                                InputStreamReader(incoming.getInputStream()));
               PrintWriter out = new PrintWriter(new OutputStreamWriter(incoming.getOutputStream()));
               out.println("MultiEchoServer: Type bye to quit");
               out.flush();
               while(true)
               {
                  String in.readLine();
                  if(str==null) {break;}
                  else{out.println("echo:"+ str); out.flush(); if(str.trim().equals("bye") break;}
               }
               incoming.close();
            }
            catch(Exception e){e.printStackTrace();}
        }
}
```

EchoClient

```
import java.io.*;
import java.net.*;
public class EchoClient
{
	public static void main(String[] args)
	{
		try
		{
		  if(args.length >0) host =args[0];
		  else host =“localhost”;
		  Socket socket= new Socket(host, 8009);
		   BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
		   PrintWriter out = new PrintWriter(new OutputStreamWriter(socket.getOutputStream()));
		   out.println(“bye”); out.flush();
		   for(int i=1; i <= 10; i++)
			{ System.out.println(“Sending: line ” + i); out.println(“line” +i); out.flush();}
		   while(true){String str =in.readline(); if(str==null) break; else System.out.println(str);}
		}
		catch(Exception e){e.printStackTrace();}
	}
}
```

Connections from Applets

- The book gives an example of an applet making a connection back to a server on the machine it came from.
- This server maintains a counter for the page the applet lives on.
- The relevant code to do to make such a connection is:

```
URL url = getDocumentBase();
Socket t = new Socket(url.getHost(), port);
```

Broadcast Echo Server

- The book gives an example of clients communicating with each other through a server.
- The server consists of the class BroadcastEchoServer which spawns BroadcastClientHandler threads in a similar fashion as was done in the MultiEchoServer.
- This time the server keeps track of all the handlers in a HashSet activeClients. It also passes into the constructor of the handlers an integer ID.
- The BroadcastClientHandler has a synchronized method sendMessage which write a message to its socket's output stream.
- Now when a handler receives a string from the client, the handler, iterates through activeClients and calls the sendMessage method of each handler with the message it just got.

Client pull/Server Push

- There are two strategies for handling real time updates over a network:
  - client pull: the client periodically contacts the server to receive current information
  - server push: the server notifies clients whenever a value has changed.
- The book gives implementations of a real time stock quote ticker in terms of both set-ups.

Remote Method Invocation

- Java Remote Method Invocation is a simplified mechanism for objects to communicate with each other over the web.
- It is a stripped down version of a programming language non-specific protocol known as CORBA.
- The key participants in the RMI architecture are:
  - The Server: -- an object that provides services to objects residing on remote hosts
  - Service Contract: -- An interface that defines the services provided by the server
  - Client: -- an object that uses the services provided by the server
  - Stub: -- an object that resides on the same host as the client and serves as a proxy for the remote server
  - Skeleton: -- an object that resides on the same host as the server and serves as a proxy for the client.

Using RMI

0. Get the rmi registry running on the server.: rmiregistry&
1. Define an interface for the remote object.

```
public interface Contract extends Remote
{//Remote is in java.rmi
        public void method1(…) throws
                RemoteException;
        //other methods
}
```

2. Implement the contract in some class on the server:

```
public class ServiceProvider extends UnicastRemoteObject
        implements Contract {/*code*/}
```

More Using RMI

3. Create an instance of the server and register that server to the RMI registry:

```
Contract server = new ServiceProvider(..);
Naming.rebind(name, server);
```

4. Generate the stub and skeleton classes, using the rmi compiler:

```
rmic  ServiceProvider
```

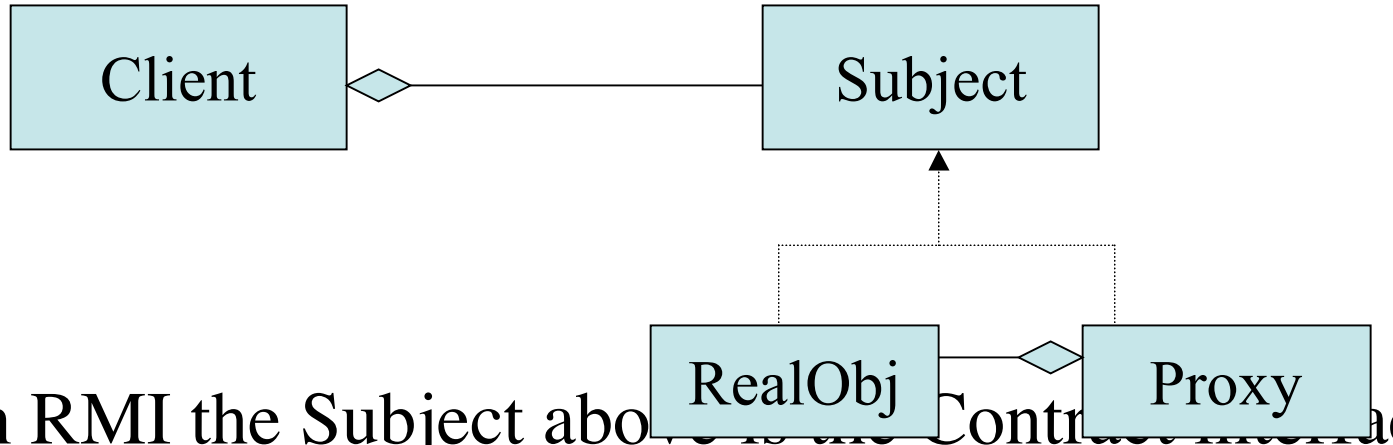
This generates two files ServiceProvider_stub.class and ServiceProvider_Skel.class. The former should be on the client machine the latter on the server. These class files actually handle the communication over the internet.

5. Develop a client that uses the services provided by the Contract interface:

```
Remote remoteObj = Naming.lookup(name);
 /*looking up rmi names is done with a URL like rmi://host:port/name where
        name is the name of the service on the server */
Contract serverObj = (Contract)remoteObj;
…
serverObj.method1(..);
```

Proxy Pattern

RMI uses something called the proxy pattern:



In RMI the Subject above is the Contract Interface, the RealObj is the ServiceProvider and the proxy is the ServiceProvider_Stub.