95-702 Distributed Systems
Project 4
Assigned: Tuesday, June 28, 2011
Due: Tuesday, July 12, 11:59:59 PM

Project Topics: Synchronous and asynchronous Java RMI and Cryptography

This project has four tasks:

First, code is presented that acts as a scaffolding for the remaining parts. A complete Java RMI solution is presented and the student is asked to build and run the system using the directions provided. The student is taught exactly how to use the rmi registry and the rmic tool.

Second, using Java RMI, the student builds a distributed but synchronous chat server. Several clients must be shown to be running and chatting. The solution is modeled after the distributed whiteboard application that is found in the Coulouris text.

Third, the student is required to add a degree of “asynchronicity” to the chat server. This amounts to providing a call back handler that runs on each client. This improvement makes for a livelier user interaction and illustrates the importance of event handling.

Fourth, the student will implement a cryptographic protocol. The first task of this project (Task 0) is re-written to prevent an attacker from reading the parameters passed in a Java RMI call.

When completing these tasks, the student should reflect on synchronous and asynchronous calls, event handling, remote interfaces, remote interface compilation, interface definition languages, cryptography, cryptographic protocols and security.
Task 0 Simple Java RMI

Task 0 is a pre-requisite to Task 3. It is a very simple Java RMI application that illustrates the use of interfaces and the RMIC tool. If you complete Task 3 of this homework then there is no need to submit Task 0. In any case, Task 0 is a very useful starting point. Note that it is designed to work from the command prompt.

Create two directories from a DOS or Unix prompt. Name one directory 'Task0Server' and the other directory 'Task0Client'.

Within the server directory, save the following three classes.

```java
//************************************************************
// Calculator.java                Interface for a Calculator
import java.rmi.*;
public interface Calculator extends Remote {
    // this method will be called from remote clients
    int add (int x, int y) throws RemoteException;
}
//*************************
************************************
// CalculatorServant.java
// A Remote object class that implements Calculator.
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class CalculatorServant extends UnicastRemoteObject implements Calculator {
    public CalculatorServant() throws RemoteException {
    }
    public int add(int x, int y) throws RemoteException {
        System.out.println("Got request to add "+x+" and "+y);
    }
```

return x + y;
}
}

import java.rmi.*;
public class CalculatorServer {
    public static void main(String args[]){
        System.out.println("Calculator Server Running");
        try{
            // create the servant
            Calculator c = new CalculatorServant();
            System.out.println("Created Calculator object");
            System.out.println("Placing in registry");
            // publish to registry
            Naming.rebind("CoolCalculator", c);
            System.out.println("CalculatorServant object ready");
        }catch(Exception e) {
            System.out.println("CalculatorServer error main " + e.getMessage());
        }
    }
}

Compile all the code with the command "javac *.*". Run RMIC on the servant class with the command "rmic -v1.2 CalculatorServant". Copy the interface Calculator.java and the stub class to the client. The stub will be called CalculatorServant_Stub.class. Within the server directory, start the rmiregistry with the command
"rmiregistry &" (Unix) or "start rmiregistry (DOS)". Run the server with the command "java CalculatorServer".

In the client directory, enter the following program:

```
import java.rmi.*;
import java.rmi.server.*;
import java.io.*;
import java.util.StringTokenizer;

public class CalculatorClient {
    public static void main(String args[]) throws Exception {
        BufferedReader in =
            new BufferedReader(
                new InputStreamReader(System.in));
        // connect to the rmiregistry and get a remote reference to the Calculator
        // object.
        Calculator c = (Calculator) Naming.lookup("//localhost/CoolCalculator");
        System.out.println("Found calculator. Enter ! to quit");
        while(true) {
            try {
```
// prompt the user
System.out.print("client>");
// get a line
String line = in.readLine();
// if a "!" is entered just exit
if(line.equals("!")) System.exit(0);
// if it's not a return get the two integers and call add
// on the remote calculator.
if(!line.equals("")) {
    StringTokenizer st = new StringTokenizer(line);
    String v1 = st.nextToken();
    String v2 = st.nextToken();
    int i = Integer.parseInt(v1);
    int j = Integer.parseInt(v2);
    int sum = c.add(i,j);
    System.out.println(sum);
}
}

catch(RemoteException e) {
    System.out.println("allComments: " + e.getMessage());
}

}
}

Compile the code on the client and run with the command
"java CalculatorClient". If you are not able to complete Task 3, for partial credit, post
this code (Task 0) to Blackboard.
Chapter 5 of the Coulouris text contains a Java RMI case study. The code implements a distributed white board. Modify the code so that it acts as a distributed chat server. Rather than moving graphical objects about we would like to move simple text.

The execution of one client program follows:

C:>java MyChatClient
client> Hello There
   Hello There
   <client> This is cool
   Hello There
   This is cool
   <client> I'm talking to myself
   Hello There
   This is cool
   I'm talking to myself
   <client>!
C:>

The explanation mark means quit. In this example, there were no other clients running. Your solution will allow for more than one client. Note that every time a client sends a message to the server the client receives an entire list of comments previously made. This is, of course, not ideal. The client may already be in possession of earlier comments and so there is no need for this extra data to be transferred. However, for this first part of the project, this approach will do fine.

Two or more users must be able to use the system to converse at the same time. Notes on rmic are available on the course slides.
You are required to separate your files into two different directories.
One will be called Task1Client and the other will be called Task1Server.
Copy relevant files to the two directories and, to simplify this effort, do not use a
security manager. Be sure to remove any reference to a security manager from the
code provided on the slides.

Post your documented java files to Blackboard. Also, to show that you have a working
system, post to Blackboard a few console screen shots showing two or more clients
talking.

So that the registry has access to necessary files, be sure to start your rmiregistry
from within your server directory.

In DOS, use the command "start rmiregistry".
In Unix use the command "rmiregistry &".

Task 2 Asynchronous Java RMI and Distributed Event Handling

The problem with the solution to Task 1 is that the client waits until the user enters a
line of text before contacting the server. It would be far better to allow the server to
make calls on the clients whenever any user enters a line of text. This is the popular
publish-subscribe design pattern. This is also called the observer pattern. It is simple
to implement using Java RMI.

Create two directories. One directory will be named Task2Server and the other will be
named Task2Client.

Write one client program called ReaderClient.java and another called WriterClient.java.
Both of these will be stored in the Task2Client directory. WriterClient.java will read
from the console (one line at a time) and send each comment to the server. It
"writes" to the server. Unlike the previous exercise, WriterClient will not read from the
server at all. That is, unlike the solution in Task 1, it will not bother to read comments
posted by others from the server and write them to the console. It simply reads from
the user and writes to the server.
ReaderClient.java will run in a separate console window and wait for calls from the server. It "reads" from the server. These calls from the server will pass to the reader recent comments that have been entered. This client will call the rmi registry to get a remote reference to the CommentList. This client will then need to call a registration method on the server. The registration method will be passed a remote reference to an object that lives on the client and whose class extends UnicastRemoteObject.

Note that the server makes a bind call on the registry but the ReaderClient does not. The server learns about the ReaderClient because the ReaderClient calls the registration method on the server. The ReaderClient does use the registry but only to look up the location of the server.

A working solution will have at least six open console scenes. One will be for the rmiregistry. The second will be for the CommentListServer. The third and fourth will be for user input (two executions of WriterClient.java) from two different users. The fifth and sixth will be two separate executions of ReaderClient (again, for two different users). These last two consoles will show the content of the comment list.

We will not use a security manager so feel free to copy the server side stubs to the client and the client stub to the server. Nor will we concern ourselves with concurrency issues.

So that the registry has access to necessary files, be sure to start your rmiregistry from within your server directory.

The client stub will be generated from running rmic on the ReaderClient.class file. The command is "rmic -v1.2 ReaderClient" without the ".class".

In your solution, you must identify the person who made the comment on every Reader that displays the comment. This will require the client to collect the user's unique screen name. How exactly that is done is up to you.
Post all of your documented source code to Blackboard. Also, to show that you have a working system, post to Blackboard a few DOS or Unix screen shots showing two or more clients talking.

**Task 3 A Simple Cryptographic Protocol**

Rewrite Task 0 so that the client uses the public key of the server to encrypt a symmetric key. This encrypted symmetric key will be transmitted to the server. It will then be used to encrypt and decrypt the operands in the call to the add method.

We will use RSA for public key encryption and Blowfish for symmetric key encryption.

It is required that you use the java keytool command to generate the server's public and private key and a public key certificate (copied to the client). Example code (on the slides) demonstrates how to use Java's crypto API. Here is an overview of the protocol that we will implement:

This problem is a simplified version of that discussed on page 278, Scenario 3, of Coulouris. We will not be using a key distribution service. Otherwise, it's the same protocol as described there.

**Client**

1. Read the server's public key (Kpub) from a provided certificate.
2. Generate a symmetric Blowfish session key s.
3. Encrypt s with Kpub.
4. Call a method on the server, passing the encrypted key s.
5. Encrypt two integer parameters using the key s.
6. Call the add method on the server with the encrypted parameters.
7. Collect the response.
8. Decrypt the response with s.
The client will use a proxy design to encapsulate the cryptography.

Server

=====
1. Provide an initialization method that collects the encrypted key s from the client.
2. Decrypt the key with the server's private key (Kpriv).
3. On subsequent calls to add, use s to decrypt the passed parameters.
4. Use s to encrypt the response to the client.

The server will use a proxy design pattern to encapsulate the cryptography.
Again, there needs to be two distinct directories. Use the same naming scheme as above.

Summary and Point Distribution:
Task 0 (Optional if Task3 is complete and working)
   Two directories:
   Task0Server and Task0Client
Task1  33%
   Two directories:
   Task1Server and Task1Client
Task2  33%
   Two directories:
   Task2Server and Task2Client
Task3  33%
   Two directories:
   Task3Server and Task3Client

Submit a single zip file of all six directories including screen shots.