95-702 Distributed Systems
Project 4

Assigned: Saturday, October 20

Due: Saturday November 3, 11:59:59 PM

Project Topics: Java RMI and a distributed, Mobile to Cloud application

This project has 3 tasks.

Tasks 1 and 2 of this project build upon the Java RMI lab that was done a few weeks ago.

Task 1 uses Java RMI to build 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.

Task 2 requires adding 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.

Task 3 will build on the Google App Engine and Android labs. You will design and build a simple mobile application of your own design that will communicate with a server application in the *cloud.*

When completing these tasks, the student should reflect on synchronous and asynchronous calls, event handling, remote interfaces, remote interface compilation, interface definition languages, mobile and cloud computing.

Task 1 Synchronous Java RMI

Chapter 5 of the Coulouris text contains a Java RMI case study. The code implements a distributed white board. The code can be found at:

http://www.cdk5.net/wp/extra-material/source-code-for-programs-in-the-book

Modify the code so that it acts as a distributed chat server. Rather than moving graphical objects, 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 in 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, provide 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 screens. 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, include a few screen shots showing two or more clients talking.

Task 3 Mobile to Cloud Application

Design and build a distributed application that works between a mobile phone and the cloud. Specifically, develop a native Android application that communicates with a web application deployed to Google App Engine.

The application is of your own design. It can be simple, but should do something of at least marginal value. For example, we have assigned projects that implement generating hash values, a calculator, a palindrome checker, and an Olympic picture application. Your application should do something similarly simple but useful (but you should not reuse our ideas!).

Users access your application via a native Android application. You do **not** need to have a browser-based interface. The Android application should communicate with your web application deployed using Google App Engine. The web application is where the business logic for your application should be implemented.

In detail, your application should satisfy the following requirements:

1. Implement a native Android application

1.1. Has at least two different kinds of views in your Layout (TextView, EditText, ImageView, etc.)

1.2. Requires input from the user

1.3. Makes an HTTP request (using an appropriate HTTP method) to your web app

1.4. Receives and parses an XML or JSON formatted reply from the web app

1.5. Displays new information to the user

1.6. Is repeatable (I.e. the user can repeatedly reuse the application without restarting it.)

2. Implement a web application, deployed to Google App Engine

2.1. Uses the MVC design pattern

2.2. Receives an HTTP request from the native Android application

2.3. Executes business logic appropriate to your application

2.4. Replies to the Android application with an XML or JSON formatted response.
The schema of the response can be of your own design.
Alternatively, you can adopt a standard schema that is appropriate to your application.
(E.g. Common Alerting Protocol if your application deals with emergency alerts.)

Submission requirements for Task 3:

1. Your Android application Eclipse project folder, zipped.
2. Your Google App Engine Eclipse project folder, zipped
3. A document describing how you have met each of the requirements (1.1 – 2.4) above. See the provided example (Project4Writeup.pdf) for the content and style of this document.

   Your write up will guide the TAs in grading your application. Because each student's application will be different, you are responsible for making it clear to the TAs how you have met these requirements, and it is in your best interest to do so. You will lose points if you don't make it clear how you have met the requirements.

If you have questions, please post them to the course Blackboard forum for Project 4 and the TAs and instructors will respond.

Summary

Task1

Two directories:

Task1Server and Task1Client

Screen shots

Task2

Two directories:

Task2Server and Task2Client

Screen shots

Task3

2 Zipped Eclipse project folders (Android and GAE projects)

Document showing how you have met the task requirements.

Zip these all together and submit a single zip file.