Project Topics: Synchronous and asynchronous Java RMI and Native Android

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 Android lab we will do on October 28. It will make requests to a web service and process results. We will introduce and demonstrate this task on Friday October 28 when we discuss Android. It will also be due with the other 2 tasks on November 4, so it would be smart to start on Tasks 1 and 2 in the meantime.

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

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:


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 builds on Project 1 Task 4 in which you built an Animal Letter Video game and the AndroidInterestingPictureLab. This time you are to build a native Android application that will make use of two RESTful web services: Flickr.com and Freebase.com in order to create a simple game that shows pictures of animals that start each letter of the alphabet.

Starting point – We recommend that you start with the AndroidInterestingPictureLab project. Once you have that working, you can create a copy of the project in Eclipse by (a) right clicking on the project and choosing "copy" then (b) right clicking again and choosing "paste". Give the new project the name "Fall2011Project4Task3".

Mapping letters to animals – Earlier, you created your own map of letters to animals. This time you should use Freebase.com to do that mapping. Freebase is a large knowledge base of structured data. Go to their web site and play around with some queries to get a feel for the data. Freebase allows, therefore, the ability to do a query such as "give me a list of animals that start with 'a'" and it will do so.

For example, try the query:

https://www.googleapis.com/freebase/v1/search?&limit=1&indent=true&type=/biology/animal&prefix=true&query=c

Dealing with JSON – Freebase takes JSON queries, and returns a JSON response. JSON is similar to XML in that it is a text-based structured data exchange standard. It essentially uses the JavaScript object literal notation. It is a standard that is very popular, especially with passing to and from browsers. It can be difficult to work with in Java, however, because of the strong-typed nature of Java.

Freebase-java is an open source project that makes it easier to make Freebase queries and work with the resulting JSON response:

http://code.google.com/p/freebase-java/w/list

Freebase-java makes it easy to (a) create the MQL query (b) do the Freebase request and (c) process the returned JSON.

Be sure to read the "GettingStarted" and "JSON" pages to understand how to create and work with JSON.

Using Freebase-java – Download and unzip the Freebase-java package. You then need to add the package to your Fall2011Project4Task3 project in Eclipse. To do this:

1. Create a folder called 'lib' in you project root folder. (I.e. in the Fall2011Project4Task3 folder. You do this from Windows Explorer or the MacOS Finder)
2. Copy the freebase-java-1.0.0.jar file into the lib folder.
3. Within Eclipse, right click on the Fall2011Project4Task3 project folder and choose "Refresh". At this point you should be able to see the lib folder in your project.

4. Right click on the freebase-java-1.0.0.jar file and do Build Path -> Add to Build Path. This will create a folder called "Referenced Libraries" in your project.

**JSON.simple Dependency** – Freebase-java has a dependency on JSON.simple. You need to download this jar and install it in the same way.


2. Move json_simple-1.1.jar to the same lib directory.

3. In Eclipse, "refresh", then add json_simple-1.1.jar to the build path.

**Freebase query** – The following freebase query works well for the letter 'c':

```json
{
  "name": null,
  "type": "/biology/animal",
  "name~": "c*",
  "limit": 1
}
```

See the Freebase MQL documentation to understand the query.

It is recommended that you limit the responses to 1. With one response, the JSON response parsing works. Without it, or with a higher limit, it throws an exception. If you can get it working with a larger number of responses, then let us know how you did so. For in that case, you don't have to use only the first animal returned, but can randomly choose from a set of animals and therefore get more variation in your application.

**Failure Model** – Your application's failure model should detect failure and degrade gracefully for the user. Account for if the Freebase query fails, and if the Freebase succeeds and either of the Flickr requests fail.

Submit your Eclipse project, and screen shots of your Android application working.

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

Zipped Eclipse project folder
Screen shots

Submit a single zip file.