95-702 Distributed Systems

Project 4

Assigned: Friday, March 21

Due: Friday April 4, 11:59pm

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

This project has 2 tasks.

Task 1 is a Java RMI project that builds on the RMI lab. You will build and deploy the dealing room system that is pictured on page 244 of the Fifth Edition of the Coulouris text.

Task 2 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, concurrency, event handling, remote interfaces, remote interface compilation, interface definition languages, mobile and cloud computing.

Task 1 A Dealing Room System using Java RMI

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

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

It is important that you study this system and understand how it works.

Your task is to solve a similar problem. You will implement the dealing room system that is pictured on page 244 of the Coulouris text.

You will use NetBeans for all of this work. We will not be using the DOS or Unix shells. In order to run the rmiregistry, we will make calls from within your server program as described below.

To simplify things a bit, we will not be using a security manager or a security policy in this project. In real life, of course, we would use security policies and managers. In an academic setting, we can skip this important step. Be sure to remove any hint of security management from your code.

Within NetBeans, your solution will be contained in exactly two projects. One project will be named StockRMIClientProject. It will contain all of the client side code. The

other project will be named StockRMIServerProject. It will contain all of the server side code.

There will be three types of clients in the dealing room system. These three clients will be found in the project named StockRMIClientProject.

The first client type will be represented by a class called StockPriceCallBack.java. This class will extend UnicastRemoteObject and implement the Notifiable interface. The Notifiable interface will define two methods (called notify and exit) and is shown below.

The StockPriceCallBack class is meant to provide callable methods to the server side and would normally run on the dealer's computer. Since it extends UnicastRemoteObject, it will be called with remote calls. This is an example where the client acts as a server. When the main routine of this class runs, it will prompt the user for a user name and then it will perform a lookup on the registry and will call the registerCallBack method on the stock service. The signature of the registerCallBack method is shown below. At that point, the server will have a remote reference to the client's callback methods. My screen looks like the following when running this class:

Enter user name:

StockDealer

Looking up the server in the registry

Creating a callback object to handle calls from the server.

Registering the callback with a name at the server.

Callback handler for StockDealer ready.

The second class that exists in the client side project is StockRMIClientPriceUpdate. This class has a main method that interacts with a user at the keyboard. The user enters lines of text in the following format: "<stockSym> <price>" or "<!>" to quit. So, an example interaction might look as follows:

Enter stock symbol and price or ! to quit.
IBM 102.00
USX 238.45
IBM 103.56
IBM 98.03
MIC 654.99
!

The user has exited by entering the "!" symbol. Each time the user enters a line of text, the server is called and the data is passed to a remote method named stockUpdate. The stock symbol and price are passed to the server and then, in turn, the stockUpdate method makes remote calls to all of the dealer computers that have registered an interest in that stock. In other words, the stockUpdate method will be calling the notify method on all clients that have registered interest in that particular stock. In the diagram in Coulouris, the StockRMIClientPriceUpdate represents an external source of events. The signature of the stockUpdate method will appear below. Note that there is no user name associated with this client.

The third class that exists in the client side project is named StockRMIClientSubscription. This class will also prompt for a user name and interact with a user at the keyboard. The user will enter lines in the following format: "S <stockSym>" or "U <stockSym>" or "<!>" to quit. So, an example interaction might look as follows:

Enter user name: StockDealer

Enter S for subscribe or U for unsubscribe followed by the stock symbol of interest.
Enter ! to quit.

S IBM
S MIC
S USX
U IBM
!

On the first line, this client (StockDealer) has subscribed to IBM using "S IBM". At that point, this client will begin to receive updates to IBM's stock price as they arrive from the external source of events (see StockRMIClientPriceUpdate described above.) As users subscribe to stocks, they receive updates on those stocks. When they unsubscribe, the updates for that particular stock stop arriving. The line "U IBM" means that the client would no longer like to be informed of updates to IBM's price. When the client exits (with '!'), the server is informed by a client side call on the server's deRegisterCallBack() method. The server then calls the client side exit() method and the client no longer receives call backs from the server.

In addition to these three classes, the client side project will also include two interfaces. The first is the Notifiable interface mentioned above. The second is the interface of the information provider and is called StockRMI.java. The implementation of this interface is in the server side project.

These two interfaces appears below and on the next page:

```
Notifiable.java
/**
 * Notifiable is an interface with two methods:
 * The method notify(String stockSym, double price) is called by the
 * server and informs the callback client that a change in the stock price has
 * occurred.
 * The method exit() is also called by the server and tells the callback client
 * that it should cease listening for this client. The user has exited the
 * system.
 *
 * This interface will be implemented on the client side.
 * The implementation will be done by an object that extends
 * the UnicastRemoteObject. The interface will be available on the
 * client and the server. */
import java.rmi.*;
public interface Notifiable extends Remote {
    public void notify(String stockSym, double price) throws RemoteException;
    public void exit() throws RemoteException;
}
```

```
StockRMI.java
/** This is the interface to the remote stock service.
 *  There are three types of clients of this service.
 *
 *  One client will be calling stockUpdate() to report on new prices for various
 *  stocks.
 *
 *  A second client will be calling subscribe() and unsubscribe() to register
 *  interest in or remove interest from a particular stock. This client will also
 *  call deRegisterCallBack() when it terminates.
 *
 *  A third type of client will be calling the registerCallBack() method so that it
 *  may receive call backs from the service when stock prices change.
 *
 *  The last two clients are associated with a single human user. So, for example,
 *  a user might use a call to subscribe to register interest in a particular stock.
 *  At that point, the stock service will begin to make calls back to the client
 *  when that stock price changes.
 *
 */
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface StockRMI extends Remote {
     public boolean subscribe(String user, String stockSym)throws RemoteException ;
     boolean unSubscribe(String user, String stockSym) throws RemoteException ;
     public void stockUpdate(String stockSym, double price) throws RemoteException;
     public void registerCallBack(Notifiable remoteClient, String user) throws
          RemoteException;
     public void deRegisterCallBack(String user) throws RemoteException;
}
```

On the server side, within the project named StockRMIServerProject, there will be two classes and two interfaces. The interfaces, StockRMI and Notifiable, are as shown above. The two classes are StockRMIServant and StockRMIServer.

The StockRMIServer class will start up the rmiregistry and bind the servant to the registry with the following commands:

```
Registry registry = LocateRegistry.createRegistry( 1099 );
Naming.rebind("stockService", stockService);
```

The StockRMIServant class will extend UnicastRemoteObject and will implement StockRMI. In my solution, I used two TreeMaps. One was used to associate a user id with a remote reference and the other was used to associate a stock symbol with its current price and a list of users that are interested in that stock.

```
/* Given a stock, get a list of users that are interested in that stock. */
private static Map stocks = new TreeMap();
/* Given a user, get the remote object reference to its callback method. */
private static Map users = new TreeMap();
```

You will need to implement the StockRMI methods in the servant. The two TreeMaps shown above may be useful.

In my solution, I made good use of the following method, which is called by the server when it is time to terminate the thread that is handling callbacks on the client.

```
public void exit() throws RemoteException{
    try{
        UnicastRemoteObject.unexportObject(this, true);
        System.out.println("StockPriceCallBack exiting.");
    }
    catch(Exception e){ System.out.println("Exception thrown" + e); }
}
```

Be sure to document your methods well. That is, using Java comments, describe how each method works. You will need to make some reasonable assumptions and come to some reasonable decisions along the way.

You may assume that the interactive users are friendly. That is, unlike a real-world system, you may assume that the user input is always correct and that no one is trying to trick or abuse the system. But, in many other respects, your system should behave in a reasonable way. For example, suppose a user enters:

S IBM
S IBM

Then, when a price update arrives for IBM, your system should only make one report on IBM's stock price. Not two. Likewise, your system should be able to handle price update requests on stocks that no one has subscribed an interest in.

Clients should be able to come and go. If a client exits with (!), then the associated callback process should terminate automatically. This is due to a call on exit() from the server.

Submission requirements for Task 1:

- Follow the project submission guide on the course schedule
- Include screen shots showing your system at work.
- Submit your solution as two Netbeans projects in a single zip file called P4T1.zip.

Task 2 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 (GAE).

The application is of your own design. It can be simple, but should fetch information from some 3rd party source and 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 artwork display application. Your application should do something similarly simple but useful (but you should not reuse our ideas!).

Your web application on GAE should fetch information from some 3rd party API. In Project 1 we experimented with screen scraping, so that is not allowed in this project. Rather, you must find an API that provides data via XML or JSON. One good place to look for such APIs is programmableweb.com.

We recommend avoiding APIs that require authentication. Many APIs will require you get a key (e.g. Flickr used in the Android lab), but avoid the headache of APIs that require authentication via OAuth or other schemes. But if you are brave, go ahead...

Two restrictions:

- As mentioned in class, one thing you cannot do is display the weather. That has been done too many times and is too boring.
- Don't use Flickr, for we have already done that.

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 on Google App Engine. The web application is where the business logic for your application should be implemented (including fetching information from the 3rd party API).

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 (if using an HttpServlet) or uses JAX-RS / Jersey.

2.2. Receives an HTTP request from the native Android application

2.3. Executes business logic appropriate to your application. This includes fetching XML or JSON information from some 3rd party API and processing the response.

- -10 if get the weather
- -10 if use Flickr
- -10 if screen scrape instead of fetching XML or JSON via a published API

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.)

- -5 if information beyond what is needed is passed on to the Android app, forcing the mobile app to do more computing than is necessary.

Submission requirements for Task 2:

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 – Submit both Netbeans projects and screen shots as defined in the project rubrics sent out for earlier projects

Task2 – Submit both Eclipse project folders (Android and GAE projects) and the document showing how you have met the task requirements using the project submission rubric.