Distributed Systems – Labs

Java RMI

Learning Outcomes:

1. Be able to create remote objects using Java RMI.
2. Be able to use the rmiregistry naming service.
3. Be able to register and look up remote objects.
4. Be able to invoke remote methods.
5. Be able to explain the difference between remote and local objects.
6. Be able to describe the role of the interface, implementation, stub, client and server.
7. Be able to establish which classes are required on the client side and which classes are required on the server side.

Tasks

1. Extract T1\Hello.java (the interface), T1\HelloImpl.java (the implementation), T1\HelloClient.java (the client) and T1\HelloServer.java (the server). Compile each of these classes.

2. Open two new command prompts in the same directory where you have your compiled code. In one of these, start the naming service as shown. This is a process that runs constantly waiting for clients to request the stub for some remote object. Before you start the `rmiregistry` you must set the classpath of that dos session to include the current directory. This is done so that the registry can locate the stub class file. This would previously have been done as a switch to the java command.

   ```
   :/> set CLASSPATH=%CLASSPATH%;.

   :/> rmiregistry
   ```

3. At one of the other command prompts, start the server. The server will create your remote object (note the server is NOT itself the remote object), and register it with the naming service.

   ```
   :/> java -classpath . HelloServer
   ```

   Look at the code for the remote object (HelloImpl.java). Also look at the interface (Hello.java). Only methods listed in the interface (more correctly the remote interface) can be invoked remotely.

4. At your third command prompt, start the client. Look at the code to see how it performs lookup, receives the stub (notice how the stub can be referenced as a Hello (the interface)), and invokes the remote method.

```
:/> java -classpath . HelloClient
```

5. Change your client so that it can take the location of the name service (previously localhost) as a command line argument. Recompile your client and use it to connect to the remote object of the person sitting next to you.

   You will find that your code will not work because the version of the compiled code being used by your neighbour is different to your own. Take a copy of their compiled stub class and their compiled interface but not the implementation and put it in your directory. See if this makes your application work.

6. Extract T6\Bank1.java, T6\Bank1Impl.java, T6\Bank1Client.java and T6\Bank1Server.java.

   This application creates a set of server side remote objects which are invoked by the client.

   Compile and run the example. Examine the code to understand what is going on.

7. Using class T7\Result and making the minor modifications that will ensure that objects of this class are serialisable, make method getResults available via an RMI interface. This method should return a *Vector* containing initialised Result objects that are set up by a server program (also to be written by you) and made available via an implementation object placed in the RMI registry by the server. The server should store two Result objects in the Vector contained within the implementation object. Access this implementation object via a client program and use the methods of the Result class to display the surname and examination mark for each of the two Result object.

   Note: Have a look at the Bank example from the Task 6.

8. Extract, compile and examine files from T8 subdirectory. They are the samples for Security manager.

Further Tasks

1. ### Callbacks

Extract, compile and run the following example. This example uses a callback object to allow the server to call methods on the client.

FT1\Client.java

FT1\Server.java

FT1\LogManager.java

FT1\Listener.java

FT1\ListenerImpl.java

FT1\MessageServer.java

FT1\MessageServerImpl.java

2. ### Security

We are now going to add security to our application to control how methods can be called on our remote objects on both the client and server sides. Java provides an inbuilt security mechanism which we will employ.

Java security managers, when employed, prevent any access to remote objects through any ports. In order to allow restricted access, we can use policy files which identify clearly which ports are open and to whom they are open.

Extract the new FT2\Client.java and FT2\Server.java. Check out the new line of code that instructs the client and server to run with a full security manager in operation.

Extract the FT2\client.policy and FT2\server.policy files. These contain the permissions given to connecting code. In this case, both policy files give almost full permissions, but this can easily be restricted.

Recompile the Client and Server with all the other code from step 1. When running the code now, you must give the names of your policy files, as shown:

```
:/> java -classpath . -
Djava.security.policy=server.policy Server

:/> java -classpath . -Djava.

security.policy=client.policy Client
```

3. Obviously in any distributed application, the client and server code would not be on the same machine, let alone in the same directory. To simulate this we are going to create separate client and server directories.
Create two folders named client and server.

Put the following files in the client folder (note these are compiled files):

- Client.class
- LogManager.class
- Listener.class
- ListenerImpl.class
- ListenerImpl_Stub.class
- MessageServer.class
- MessageServerImpl_Stub.class
- client.policy

Put the following files in the server folder:

- Server.class
- LogManager.class
- MessageServer.class
- MessageServerImpl.class
- MessageServerImpl_Stub.class
- Listener.class
- ListenerImpl_Stub.class
- server.policy

Note how the client side does not require the implementation for the server side remote object, and the server side does not need the implementation for the client side remote object.

Start the `rmiregistry` from the server directory. Run the client and server in the same way as at step 2 above.

4. ## Dynamic Class Loading

Dynamic class loading is the facility that allows Java to look at runtime for a class that it requires - unlike other programming languages where all the code is linked into an executable following compilation.

Java can load classes from anywhere - for example from any part of your class path on your local machine or your network. It can also load classes over the web. This is what we will do.

If the server implementation is changed and a new stub class is generated the stub class will have to be distributed to all clients. This is undesirable. A better approach would be to make the new stub class available online and whenever a

client starts up it automatically loads the new class from the web server.

In order to do this, follow steps 5 - 8 below.

5. Extract the following web server software FT5\new_ws.ZIP. Unzip the contents into a new directory named ws. In this directory you will find a batch file named `run.bat`. Execute this. Open a web browser and point it to http://localhost:8008/. This will show you the contents of the htdocs directory.

6. Remove MessageServerImpl_Stub.class from the client directory - this is the class that will be dynamically loaded from the web. Put it in the htdocs directory of your web server.

   Also put a copy of LogManager.class, Listener.class and MessageServer.class in the htdocs directory. These are referenced by the stub class and as such are required here.

7. Start your `rmiregistry` from the root of your c: drive. Do not set the `classpath`. The `rmiregistry` will get the stub class from the web server.

8. Restart the server with the following command:

   :/> java -classpath . -Djava.security.policy=server.policy -Djava.rmi.server.codebase=http://localhost:8008/ Server

   Note how we give the `codebase` for the stub so that this can be provided to the `rmiregistry`. Observe the log for the web server (this is echoed to the terminal window for the web server). Notice how requests were made for each of our classes.

9. Start the client in the same way as before. Observe once again how the client made a request to the web server for the stub class i.e. it was dynamically loaded from the web server.

Further Reading

1) Chapter 5 Introduction to Network Programming in Java by Graba.

2) Java tutorial section on Java RMI:

http://docs.oracle.com/javase/tutorial/rmi/