Practical Exercises in Computer Networks and Distributed Systems

Distributed objects with Java RMI (WIP)

Object oriented programming and distributed programming blend nicely in Java RMI (Remote Method Invocation), this technology allows a Java client to call methods of objects installed at a Java Virtual Machine (JVM) other than the client's. These latter methods constitute a form of Server technology, consequently, Java RMI implements the Client/Server model. Of particular importance is the fact that the client and server JVMs may be running in separate Internet hosts. In this practical we detail how to deploy a simple C/S application where the Client accesses the Server by way of Java RMI, *i.e.,* by invoking the methods exported by the server. The purpose of the C/S application involved in this practical is to simply illustrate Java RMI by way of example, in a straightforward way, therefore we will not undertake any exhaustive analysis or design nor will we study the different customary architecture and design patterns at use in the realm of RMI.

Before working this practical I recommend that you study the introductory presentation to Java RMI that you may download from http://paloalto.unileon.es/ds/Lab/RMI/DS-StructureOfJavaRMI.pdf alongside the presentation that briefly describes the code used in the classes of this practical: http://paloalto.unileon.es/ds/Lab/RMI/DS-JavaRMI-Concepts.pdf

(So far, this document has no exercises nor conclusions sections)

Java RMI distributed objects and programming to interfaces

The Server program exports two remote methods that we will access from our Client, which as you may guess is also written in Java. When I was of devising this RMI introductory practical I considered a few short and simple illustrative algorithms to be encapsulated into two *remote* methods: I selected the generation of a Hash value to a long string and the calculation of the *old* factorial of an integer, you can see their signatures in the following Java interface file:

```
package sdrmiexample;

import java.rmi.*;

public interface SDRemoteObject extends Remote{

    public int longStringHash(String s) throws RemoteException;

    public long factorial(int x) throws RemoteException;

}
```

This Java interface file expresses the *remote interface of the remote object,* when we write the Server program we will have it

implement this interface, for the time being it suffices to observe that both methods may throw a RemoteException - *e.g.* an exception that happens when IP connectivity is lost between Client and Server. Glancing over the preceding Java interface we become convinced that we are in familiar ground, that is what we expect from Java RMI: programming against remote objects as though they were *local* objects, remember the DS transparencies that we studied in the introductory chapter to Distributed Systems? The access transparency applies to Java RMI alongside with others.

The SDRemoteObject Java interface is a purely abstract data type that lends itself to be included in our client program without yet having access to any specific instance of it, *i.e.,* we can say variable x is a reference to an instance of SDRemoteObject though we don't yet have that instance. This style of programming is known as *programming to interfaces* by contrast to the more familiar *programming to implementations.* All in all, we will program our client and server programs knowing just the remote interface implemented by the remote object, not the real object which, as you may have already guessed, resides in a remote host, in a separate memory space, in a separate JVM. Obviously, when our program is run, it will necessarily have to fetch a real instance of the remote object so that it may call its methods: The real object downloaded by the client is not the full remote object, it simply represents the real remote object, it's a proxy to the remote object and it's composed of two parts: the remote interface (A compiled Java interface) and the stub.

Java RMI distributed objects and programming to interfaces

A Java remote object is a Java object that can be accessed from anywhere in the network via RMI. Our example remote object has the two public methods mentioned above which constitute the Java interface also mentioned above (sdrmiexample.RemoteMethods). Since we want our example object to have methods accessible via RMI, we will write a class named RemoteMethodsImpl that will implement sdrmiexample.RemoteMethods, also, Java RMI requires that it extend UnicastRemoteObject, thereby allowing it to inherit the essential methods for becoming an effective remote object. I recommend that you read the source code to the ExampleMethodsImpl.java class and that you observe that this class implements the interface RemoteMethods.java and that each exported method throws RemoteException.

Now that we have our remote object ready, we move on to writing the Server program which will actually instantiate our brand new remote object. The server program creates an instance of SDRemoteObjectImpl(), the remote object and proceeds to register it in a remote object directory known as Java RMI registry. The registry (Its command name is *rmiregistry* and it comes bundled with the JDK you installed earlier when you setup your Linux for Distributed Systems) must be running on the same host where the remote objects are instantiated; rmiregistry will receive a request from the server indicating it to associate a name with the remote object. The server must remain in execution for the whole remote object lifetime. The static method rebind() of the Naming class does the job of contacting the registry and requesting the remote object registration. From that moment on, a client may lookup the registry for the publicly known-name of the remote object, then the client will be handed a copy of the remote object known as *stub* which will act, on the client side, as a proxy for the real remote object. Chapter 5 of the Distributed Systems textbook by Coulouris *et al.* provides an extensive explanation of these processes and the underlying structures.

Procedure for setting the Java RMI example to work

On the following sections we will detail how to set all the parts of this project to work and, where necessary we will offer further technical details since the description above is rather shallow.

We will download the compressed source tree to a base folder named /users/tomas/server and /users/tomas/client,

decompress it and move the client and server classes where appropriate:

```
$ cd /users/tomas

$ mkdir client
$ mkdir server
$ mkdir server/sdrmiexample
$ mkdir client/sdrmiexample

$ wget http://paloalto.unileon.es/ds/Lab/RMI/sdrmiexample.zip

$ unzip sdrmiexample.zip

$ cp Server.java SDRemoteObject.java SDRemoteObjectImpl.java
     server/sdrmiexample

$ cp Client.java SDRemoteObject.java client/sdrmiexample

$ cp all.policy client
```

Now we will compile the server and the client:

```
$ cd /users/tomas/client/sdrmiexample
$ javac *.java

$ cd /users/tomas/server/sdrmiexample
$ javac *.java
```

We need to copy the public interface binary file (SDRemoteObject.class) to a web server so that, when the client requests the *stub class* it will be properly accessible. Let's assume that you are using a Linux Ubuntu distribution and that you want to have an Apache Web server installed at your system[1]:

```
$ sudo su
Password: ...

# apt-get update
...

# apt-get upgrade
...
```

[1] From this point on, the sharp symbol (#) represents the root shell prompt, *i.e.*, the prompt presented by the shell after we have successfully switched to user root. Please, do not mistake it for the symbol representing a commented line in Bourne shell scripts.

```
# apt-get install apache2
...
```

If you need further assistance in installing Apache Web Server, you may refer to the following URL: https://help.ubuntu.com/10.04/serverguide/httpd.html. Let's assume a successful installation, then, restart the web service in your host:

```
# /etc/init.d/apache2 restart
```

Assuming that the base folder for web documents in your Apache installation is at: /var/www/html, we will create a folder named after the package name for the SDRemoteObject.class and copy it there (Obviously, if your specific installation results in a different base documents folder, you will have to modify the following commands accordingly):

```
# mkdir /var/www/html/sdrmiexample
# cp /users/tomas/server/SDRemoteObject.class /var/www/html/sdrmiexample
```

I recommend that you check that the SDRemoteObject.class file is accessible from the web, assume that your IP address (The IP addresses used below are illustrating examples, not the IPs you must use) is 192.168.2.131, then, simply download it by executing the following command:

```
# cd /tmp
# wget  192.168.2.131/sdrmiexample/SDRemoteObject.class
```

You should have the file in your current directory now, otherwise something did not work properly, then you will have to check that the file is in the proper position, etc. After the download is successful, you have a guarantee that the class file is accessible via http and hat the client program will be able to access it; now, you can rm the class file:

```
# rm SDRemoteObject.class
```

Before our server program is run we need a rmiregistry properly running in our host so that requests from clients result in the remote object being properly located, we will start the rmiregistry by submitting the following command line -avoid copy/paste from this pdf file which could result in unexpected behavior when you submit the command line to the shell, download instead the shell script that contains the command line from:

```
http://paloalto.unileon.es/ds/Lab/RMI/start-registry.sh
```

```
# cd /
# rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false &
```

rmiregistry is run in the background and we passed it a JVM property set to false that instructs it to use the Java codebase property received from the Server as the only location to lookup the Stub objects (A Stub object represents the remote object on the client side). rmiregistry is a binary program that spawns a JVM and the rmiregistry mainframe class.

Now, we can execute the server so that it is ready to receive client requests, the remote object will be called "Aslan" (Download the Shell script from the following URL and make sure you edit the **command_server.sh** file and modify the IP addresses according to your system's IP):

http://paloalto.unileon.es/ds/Lab/RMI/start-server.sh

```
# cd /users/tomas/server
# java -Djava.rmi.server.hostname=192.168.2.131
-Djava.rmi.server.codebase=http://192.168.2.131/
sdrmiexample.Server  Aslan
```

The preceding command assumes that the web server that offers the stubs runs within the same host where the RMI server runs, but, this is not required whatsoever, *i.e.,* the stubs may be served from elsewhere. The codebase property set specifies where the binary interface files that comprise the stubs can be found in the web; the hostname property specifies on which IP address should our remote object listen for requests in case your host system is multihomed (Several network interface cards configured with different IP addresses). The server program should printout the following message in your terminal:

```
------------------------------------------------
- Remote objects instantiated -
- Remote objects registered (Java RMI registry) -
- Server running -
------------------------------------------------
```

We are now ready to fully check the project by executing the Client program. This program accesses a running instance of Java rmiregistry, it will use it for looking up our Aslan object, our remote object, finally, it will execute the two remote methods exported by the remote object whose names are longStringHash() and factorial(). Proceed according to the following command sequence (Download the Shell script from the following URL and make sure you edit the **command_client.sh** file and modify the server's IP address according to your server's IP):

http://paloalto.unileon.es/ds/Lab/RMI/start-client.sh

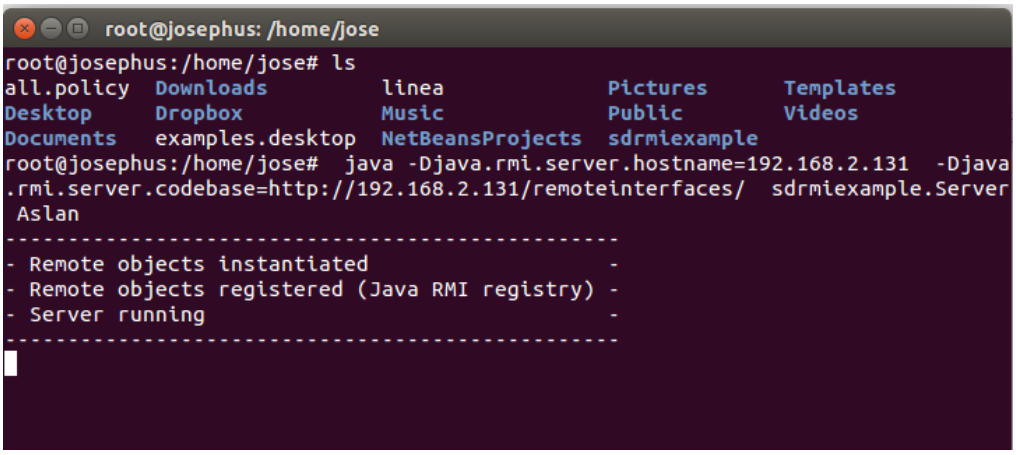
```
# cd /users/tomas/client
# java -Djava.security.policy="all.policy" sdrmiexample.Client
192.168.2.131 Aslan
```

We have started the client class passing it the java property that explicitly allows it to run free of any security restrictions (You may want to take a look at the contents of the security policy file all.policy). The IP address is the rmiregistry's IP and Aslan is the name of the remote object that we want looked up. If no problems arise, the client should printout the following message sequence on your console:

```
jose@josephus:~/client$ java -Djava.security.policy="all.policy"
sdrmiexample.Client 192.168.2.131 Aslan
registry found
stub lookup done
Hash of Hello world from Java RMI!!! = 23
Computing 7!  = 5040
jose@josephus:~/client$
```

On the server screen, the remote methods will printout the following messages as requests from the clients arrive:

```
------------------------------------------------
- Remote objects instantiated                  -
- Remote objects registered (Java RMI registry) -
- Server running                               -
------------------------------------------------
Thread name:  RMI TCP Connection(6)-192.168.2.131
Thread name:  RMI TCP Connection(6)-192.168.2.131
```



Since the server process is an RMI server, the RMI runtime will keep it running indefinitely, until you send it a kill signal (Via ctrl-C, for example) *–or until it crashes!*