Solution 3

Remote Method Invocation

1 Overview

We give here the complete solution for Exercise 3. We first focus on enhancing Solution 2 to support RMI and then we modify Solution 2 in order to use the RMI callbacks.

2 Getting, Compiling and Running the Application

We present here the different steps to execute in order to run the RMI solution we provide.

2.1 Compilation

The complete source code is available for download at the exercise page of the course web site. The archive contains an ant (*http://ant.apache.org*) build file that can be used to compile the sources and generate the Java documentation. To do this, simply type in the *Ex3* directory[1]:

```
ant
```

You can notice in the `build.xml` file that we compile the stub of the server separately with a compiler called `rmic`. This supplementary compilation phase is not needed anymore with Java 1.5 for the classes that extend `java.rmi.server.UnicastRemoteObject`.

2.2 Starting the RMI Registry

Once the application is compiled, you have to start the *rmiregistry*. To that end you can follow two different approaches:

- (1) You assume that the RMI registry knows where to get the stub of the server as well as all the different classes that will be transferred between the client and the server.
- (2) You make no assumption on the RMI registry and hence the client and the server will have to specify where to those classes are.

The first case corresponds to setting the *CLASSPATH* variable before starting the RMI registry. Once you have setup the *CLASSPATH* correctly you can start the client and the server without any trouble. This solution has however a drawback in the sense that all the services that want to use this RMI registry must be in the classpath of this registry. This is not a very good solution.

In the second case, you do not specify any *CLASSPATH* variable before starting the registry and the server will be responsible to provide the registry with the needed classes. We have chosen this approach in our solution.

Hence, to start the registry, simply type:

```
rmiregisty [rmiportnumber] &
```

[1] Please use the latest version of ant.

2.3 Starting the Server and the Client

To start the server, execute the following in a console:

```
ant RMIServer [-Dportnumber=rmiportnumber]
```

If you take a look at the build file, you can see that we set the *-Djava.rmi.server.codebase* property to a directory that contains all the files that the client might need to download: `ChatServer_Stub.class`, `IMessage.class` and `IServer.class`. Please note that the `IMessage.class` and `IServer.class` files are not necessary as the client must have them in its classpath to correctly compile its code.

To start the client, type the following in a console:

```
ant RMIClient [-Dhostname=serverHostName] [-Dportnumber=rmiportnumber]
```

If we look at the build file, we can notice that we set the *-Djava.security.policy* parameter in order to accept a specific policy file. This file allows the `java.rmi.RMISecurityManager` of the `ChatClient` to connect to the registry and to read the files of the server (i.e., the `ChatServer_Stub.class`).

3 Design

The design as well as the usage of the new chat application is exactly the same as the one presented in Solution 2. Please have a look at it.

4 Modifications

We present here the different modifications done in specific classes in order to make use of Java RMI.

4.1 RMI Client Class

In the `RMIClient` class a `start()` method is used to lookup the server and to start the chat clients. The `main()` method calls the `start()` method, used to lookup the *ChatServer* service, and hence has to catch several exceptions (`java.rmi.NotBoundException`, `java.net.MalformedURLException` `java.rmi.RemoteException`). The lookup of the service is done via the `java.rmi.Naming.lookup()` method.

We also specify in the `main()` method a `java.rmi.RMISecurityManager` in order for the client to be able to download the stub of the server.

Finally, we have changed the constructor of this class to accept an additional argument, which is the port number of the RMI registry.

4.2 ChatClient Class

In this class, all the methods that include method calls on the server contain now a `try catch` clause in order to catch a potential `java.rmi.RemoteException`. These methods are: (1) `connect()`, (2) `disconnect()`, (3) `sendMessage()` and (4) `run()`.

4.3 ClientListFetcher Class

In the `run()` method, we have a `try catch` clause to handle possible `java.rmi.RemoteException` while retrieving the list of clients.

4.4 IMessage Interface

In order to be able to send messages from the `ChatClient` to the `ChatServer`, the `IMessage` interface must be serializable. To that end, it extends the `java.io.Serializable` interface.

4.5 IServer Interface

Now this interface has to extends the `java.rmi.Remote` class. Consequently, all the methods that are declared in `IServer` have to throw the `java.rmi.RemoteException`.

4.6 ChatServer Class

This class has changed the most. First, it has to extend `java.rmi.server.UnicastRemoteObject` in order to receive incoming calls from the clients. Consequently, the constructor of the `ChatServer` class has to throw a `java.rmi.RemoteException`.

The second major change comes from the fact that now the `ChatServer` is a standalone program and hence has to define a `main()` method. In this main method, we create the `ChatServer` instance and we bind this instance to its name into the RMI registry. This is done via a call to `java.rmi.Naming.rebind()` method.

5 RMI Callbacks

The code containing the classes for the RMI callbacks are available in the callbacks directory. You can launch the client and the server as mentioned in Section 2. Please note that you do not need to start the RMI registry. It is started when launching the `ChatServer`.

To use the RMI callbacks, we had to change several classes. We present in the following those changes.

5.1 Major Changes

The `RMIClient` class does not exist anymore. The class that starts the client is now directly `ChatClient`. Consequently, it is not possible anymore to start multiple clients from the same JVM. This is not a limitation of the RMI callbacks but only a design choice. Accordingly, the `IWindowManager` interface has been removed.

The `ChatServer` has now references to its `ChatClient` directly (via the `IChatClientListener` interface). Hence no more threads are used in the `ChatClient` and no more blocking queues are used in the `ChatServer` (because the server will directly call its clients when new messages are sent or new clients connect).

5.2 IChatClientListener Interface

A new interface, `IChatClientListener` is used to provide the interface between the `ChatServer` and the `ChatClient`. This interface extends the `java.rmi.Remote` interface and describes the methods the server can call on the clients. A stub will be created during compilation for the implementation of this interface (i.e., for `ChatClient`).

5.3 ChatClient Class

As previously explained this class does not need to implement `java.lang.Runnable` anymore, as its methods will be called asynchronously by the server when new messages are sent or new clients connect. To that end, this class implements the new `IChatClientListener` interface such that the server can remotely access it. Consequently, it implements the `newListOfClients()` method as well as the `newMessage()` method.

This class now defines a `main()` method used to start the application and creates its own `java.rmi.RMISecurityManager`. In that security manager, we overload the `checkPermission()` methods in order to allow everything. We could, of course, implement more restrictive methods that would grant only the mandatory rights for the application to work.

Finally, we export the `IChatClientListener` via the `java.rmi.server.UnicastRemoteObject` class in order for the `ChatServer` to be able to call methods on this client.

5.4 IServer Interface

The `getMessage()` and `getListOfClients()` method have been removed from this interface. Moreover, the `connect()` and `disconnect()` methods have been changed to give a reference to a `IChatClientListener` as a parameter.

5.5 ChatServer Class

The `ChatServer` manages now references to `IChatClientListener` and not anymore to `InternalClients`. It manages also a list of user names of the clients (please note that we could have incorporated a `getUsername()` method in the `IChatClientListener` in order to get rid of this list).

Moreover we implement the changed `connect()` and `disconnect()` methods. In each of them, we send to the connected clients the list of their colleagues.

The `sendMessage()` method now directly sends the new message to the connected clients.

Finally, the `main()` method correctly sets the `java.rmi.RMISecurityManager` (in allowing, for instance, everything, by re-implementing the `checkPermission()` methods), creates the registry (with the `java.rmi.registry.LocateRegistry.createRegistry()` method) and binds the *ChatService* to this registry.

6 Security and Policy File

As presented in Section 2, if your application needs to download byte code you have to set a `java.lang.SecurityManager` and specific security policy.

To achieve that goal, either you set a default `java.rmi.RMISecurityManager` in the `ChatClient` and then specify a *java.policy* file to grant the correct permissions or you re-implement the pertinent (e.g., `checkPermissions()`) methods of the `java.rmi.RMISecurityManager` (see the implementation of the `ChatClient` of the callbacks directory).

7 Starting the RMI Registry Automatically

If you want to start automatically a RMI registry when you launch your `ChatServer` this is possible via the `createRegistry()` method of the `java.rmi.registry.LocateRegistry` class.

References

[1] Java Remote Method Invocation Tutorial. *http://java.sun.com/docs/books/tutorial/rmi/*.

[2] Getting Started With Java RMI. *http://java.sun.com/j2se/1.4.2/docs/guide/rmi/getstart.doc.html*.

[3] Java RMI Codebase Howto. *http://java.sun.com/j2se/1.4.2/docs/guide/rmi/codebase.html*.