Exercise 3

Remote Method Invocation

1 Goal

The goal of this exercise is to teach you how to program a distributed chat application using Java Remote Method Invocation (RMI). To that end you will reuse your existing implementation of our local chat and distribute it. In this new application you will be able to connect, via a chat client, to a chat server, the applications running on different hosts.

2 Guidelines

We present here in more detail the different indications that will help you to implement the RMI-based chat application.

The application will contain two main classes: (1) a chat server (`ChatServer`) and (2) a chat client (`RMIClient`). Different instances of a `RMIClient` (that will in turn create instances of `ChatClient`) can be launched on different hosts and will send messages to the `ChatServer`. As usual, the clients are able to see the list of the connected users as well as the different messages sent by them. The server is responsible for dispatching the messages to the different connected clients.

The messages are sent from the clients to the server using remote method invocation. There are different possibilities for the server to dispatch messages to its clients: (1) using the traditional producer/consumer pattern (see Section 2.3) or (2) using the RMI callbacks (see Section 2.4).

2.1 Applications Overview

2.1.1 RMI Client

The user should be able to launch the client application via the following command:

```
java ch.epfl.lpd.ids.RMIClient <rmiHostName> <rmiPort>
```

The same window as presented in Exercise 2 should appear (Figure 1).

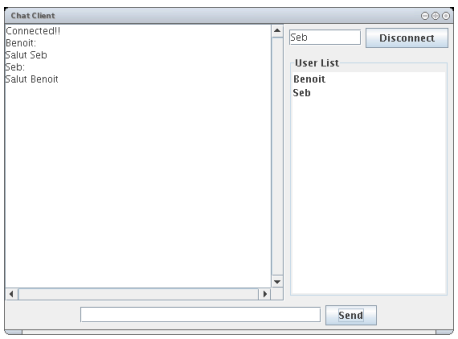

Figure 1: A Chat Client GUI

As usual, a user is able to connect/disconnect (via the `Connect` button), send messages (via the `Send` button) and receive messages as well as the list of the connected users.

2.1.2 Chat Server

The server should be started using the following command:

```
java ch.epfl.lpd.ids.server.ChatServer <rmiPort>
```

An information message saying that the server has been bound to the *rmi registry* should be printed out. The server maintains a list of its connected clients and is responsible for dispatching the messages it receives.

An application can bind or unbind only to a registry running on the same host for security reasons (see [3]).

2.2 Design Overview

As presented during the course, in order to use Java RMI, a `Remote` interface should be designed. This interface must be implemented by the component that is accessed via remote invocation (in our case, this is the chat server). In this case, the architecture of the applications should follow Figure 2.

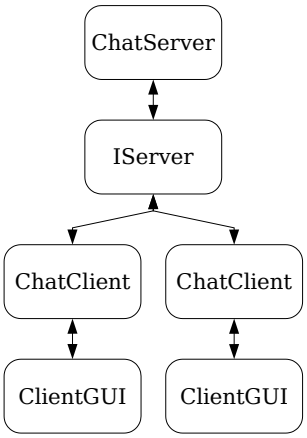

Figure 2: RMI Chat Architecture

The different instances of the `ChatClient` will communicate through the `IServer` to send/receive information to/from the `ChatServer`.

2.3 Enhancing your Previous Application

If you have well designed your previous application, the `IServer` interface should already be implemented. In this case, you just have to modify this interface in such a way that it extends the `java.rmi.Remote` interface. Once this is done, you have to make several modifications in both the `ChatServer` and the `ChatClient` classes.

First the `ChatServer` must be able to run as a standalone application. Then it should implements the modified `IServer` interface. As a consequence, it has to extend the `java.rmi.server.UnicastRemoteObject`. Finally, it must bind itself to the *rmi registry* in order to be accessible.

Regarding the `ChatClient`, it has to lookup the `IServer` object and as it downloads code from the *rmi registry*, it has to install a `RMISecurityManager`.

2.4 Using Java RMI Callbacks

It is possible to use RMI callbacks instead of the traditional producer/consumer pattern. Indeed, we can imagine that each time the server receives a message from a client, it calls a method on

each of the clients to inform them that a new message has been received. The clients will not *pull* the messages from the server anymore but the server will *push* the messages to them.

Using callbacks have several advantages: (1) less data structures are involved and (2) less threads are needed (smaller chances of deadlocks). However, in this case, the `ChatClient` has to implement an interface through which the server will be able to communicate with it. The new design is presented in Figure 3.

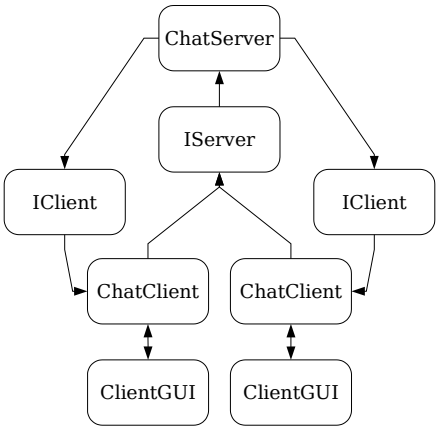

Figure 3: RMI Chat Architecture with Callbacks

In addition to the different steps presented in Section 2.3, the `ChatClient` has to export itself via the `UnicastRemoteObject.exportObject()` method. This allows other distributed components to communicate with the `ChatClient`. As the `ChatClient` will download the code of the `ChatServer` it has to instantiate a `RMISecurityManager`.

2.5 Compiling your Applications

The compilation of your different application is done as usual. However, you now need to compile also the stub and skeleton of your remote object. To that end you need to use the special *rmic* compiler.

For compiling the stub of the `ChatServer`: `rmic ch.epfl.lpd.ids.server.ChatServer`.

If you use the callbacks alternative, you will also need to compile the stub for the `ChatClient`: `rmic ch.epfl.lpd.ids.client.ChatClient`.

Have a look at the option of the *rmic* compiler if you must set a specific classpath.

2.6 Running your Applications

2.6.1 RMI registry

Before being able to run your different applications as presented in Section 2.1, you have to start the *rmi registry* on the computer that will host the server. The following command achieves this goal: `rmiregistry&`.

You will see in the solutions of the exercise that it is possible to get rid of this execution step and start the registry directly in the `ChatServer` application.

2.6.2 Security Policy

By default, the `RMISecurityManager` installed at the client side do not allow the client to connect to or resolve an IP address. So, in order for the `RMIClient` to connect to the `rmi registry` as well as to the `ChatServer`, it has to specify a *policy* file to its security manager.

This file contains the following:

```
 grant {
    permission java.net.SocketPermission "*:1024-65535",
        "connect";
};
```

The *policy* file has to be given to the virtual machine when starting the application. The new command for launching the server will be: `java -Djava.security.policy=java.policy ch.epfl.lpd.ids.RMIClient <rmiHostName> <rmiPort>`.

We will see in the solutions of the exercise that is it possible to specify the *policy* directly in the `ChatClient` application.

Please note that if you use callbacks, the same issues must apply for the `ChatServer` application.

3 Due

You have to give, for May 2^{nd}, the complete source code of your application. You can send your code, **without** the compiled classes, in a **.zip** file archive (or **.tgz**) to the following email address: *Sebastien.Baehni@epfl.ch*. The archive **has** to contain a script file used for compiling your code (either on Windows or on Linux). Your email should arrive before noon and its subject has to be: *IDS EX3*. Also, please put in the content of your mail the firstname and name of your partners.

The application has to run correctly (i.e, without bugs) for you to receive 0.33 bonus points. If the code is well designed, well written and generic, you will receive additionally 0.33 points. Finally, if the code is well documented (i.e., javadoc preferably) you will get the final 0.33 points of the bonus.

References

[1] Java API Documentation. *http://java.sun.com/j2se/1.5.0/docs/api*.

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

[3] Java Remote Method Invocation Getting Started.
http://java.sun.com/j2se/1.4.2/docs/guide/rmi/getstart.doc.html.