Polytech'Grenoble

TP3 - RMI-based Distributed Applications

Polytech/INFO4, année 2018-2019
F. Boyer, O. Gruber

Organisation	Binome
Evaluation	Code-based
Advised time	3-4h per binome
Tools	JDK >=1.7, IDE (eclipse or equivalent)
Return	Compressed source files, in a file named Name1_Name2_RMI.zip (or .tar)
Contact	Fabienne.Boyer@imag.dfr, Olivier.Gruber@imag.fr

1 Preamble

This practical work aims at programming a Chat application over Java/RMI. For a good understanding of this work, you need to have in mind the RMI lecture. As additional reference guides, you may look at:

- https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/hello/hello-world.html
- https://docs.oracle.com/javase/tutorial/rmi/index.html

2 Baby step

As a baby-step, we will consider programming a Printer application over Java/RMI whose behavior can be sketched as follows:

- Client side:
  - The client gets the reference of the printer server, implementing the interface *IPrinter,* from the RMI registry
  - The client calls the method *printLine(String s)* on the server, the method *printLine* is defined by the IPrinter interface .
- Server side:
  - The server instantiates the class *Printer*, implementing the IPrinter interface
  - The method *printLine(String s)* prints a line on a local screen,
  - The server registers the *Printer* instance onto the RMI registry.

Steps to follow:

1. Start by determining which classes compose your application. Basically, you may consider four classes here: *Client, Server, Printer,* and *IPrinter*. The classes *Client* and *Server* respectively launch the client and server applications. The interface *IPrinter* defines the remote methods that can be invoked on any instance of the *Printer* class. The Printer class implements the methods defined in the *IPrinter* interface.
2. The second step is to determine which objects compose your application at runtime and how these objects are aliased. We ask you to draw on a paper the global distributed object graph, showing local objects and remote objects, and also including all the stubs and skeletons.
3. Now you are ready to program your application, so create a Java project under Eclipse. Create first the interface *IPrinter* and its corresponding implementation (*Printer* class). Then, define the *Server* and *Client* classes.
4. Regarding the Server class, you have the choice of embedding the *RMIregistry*, or to launch it as a standalone application (through the *rmiregistry* command). We advise you to embed the *RMIRegistry* within your server because this greatly simplifies the way you start and stop your application. A reminder of how to do this is given next page.
5. Start your server, and then start several clients. Check that things go properly.

Reminder

To embed the RMIregistry within your server, you create and start a registry on a given port using the following instruction:

```
Registry reg = LocateRegistry.createRegistry(port);
```

Once the registry is started, we may (re)bind names to remote references as follow (assuming *printer* references your instance of the Printer class):

```
reg.rebind("LinePrinter", printer);
```

Another option is to use the static method *rebind* of the Naming class, that takes as argument the *url* of the registry:

```
Naming.rebind("//" + host + ":" + port + "/" + "LinePrinter", printer);
```

At Client side, you may get the reference of the registry and then the remote reference to the printer as follows:

```
Registry reg = LocateRegistry.getRegistry(host, port);
printer = reg.lookup("LinePrinter");
```

Another option is to simply use the static method *lookup* of the Naming class that takes as argument the *url* of the registry:

```
printer = Naming.lookup("//" + host + ":" + port + "/" + "LinePrinter";
```

3 Chat server

We now consider programming a Chat Server over Java/RMI. This server manages one or more *chat rooms* where several participants in a given room may exchange messages.

The communication protocol of a participant with a Chat server is very simple; it consists of the following operations that may be called on the Chat server:

- Connect: allows a new participant to connect to a chat room
- Leave: allows a participant to leave a chat room
- Say: allows a participant to send a message to a given chat room
- Who: allows a participant to get the names of all participants currently connected to a chat room

To be able to send or receive messages to a chat room, a participant must be connected to that chat room. When a participant sends a message to a chat room, the Chat server delivers it to all participants that are currently connected to that chat room.

Chat Server – First Version: the chat server will manage a <u>single chat room</u>

In Eclipse, create a Java project for the Chat server and then follow these steps:

1. Firstly, determine what classes and interfaces compose your application. We advise you to consider six classes here: *Client, Server, ChatRoom / IChatRoom*, and *Participant / IParticipant*. The *Client* and *Server* classes allow respectively to launch the client and server applications. The *IChatRoom* interface defines the remote methods that can be invoked on a *ChatRoom* instance. The *IParticipant* interface defines the remote methods that can be invoked on a *Participant* instance.
2. As a second step, determine what objects compose your application at runtime and how the corresponding object graph (including object aliasing). As previously, we ask you to draw the global **distributed** object graph, showing local objects and remote objects, and also including all the stubs and skeletons.
3. Now define the *ChatRoom* and *Participant* classes implementing the *IChatRoom* and *IParticipant* interfaces defined as proposed below. Pay attention to the fact that any remote object may be invoked **concurrently**, as soon as it is remotely aliased from several clients.

```
Interface IChatRoom:
   String name();
   void connect(IParticipant p);
   void leave(IParticipant p);
   String[] who();
   void send(IParticipant p, String msg);

Interface IParticipant:
   String name();
    void receive(String name, String msg);
```

4. Once this is done, define your class *Server* and *Client*. The class *Client* uses the shell command line to interact with end users.
5. Finally, start one or more servers and launch several clients to test your application.

Packaged version - steps to follow:

We now focus on how to package and deploy our Chat application over distributed machines (client and server machines). In the previous versions, you launched the clients and the server under Eclipse, from the same Java project. This does not match the reality of distributed applications deployed in the real world. Firstly, clients and servers are usually started via shell scripts as standalone Java applications, independently from Eclipse or any other IDE. Secondly, the client part and the server part are delivered as separate Java Archive files (jar):

1. A *ChatClient.jar* that you can give to anyone who wants to launch the Client application allowing to connect to a given Chat server. Determine what classes should compose this archive.
2. A *ChatServer.jar* archive that you can give to anyone who wants to launch a Chat server. Notice that some classes need to be present in both archives.

To create the *ChatClient.jar* and *ChatServer.jar* archives under Eclipse:

- Select *Export* then *Create Jar*, do not select *Create Runnable Jar*.
- Check that your jars contains the expected files from the command: java tvf X.jar

To launch manually, from a shell, your server application*, you can use the following command that* assumes your class Server is in the Java package *chat.server:*

java -cp ChatServer.jar chat.server.Server <args>

To launch manually, from a shell, client applications*, you can use the following command that* assumes your class Client is in the Java package *chat.client:*

java -cp ChatClient.jar chat.client.Client <args>

Notice that you can start one or more clients, on one or more machines.

Chat Server – Second Version: the chat server will manage <u>multiple chat rooms</u>

Add the possibility to your class Server to manage multiple chat rooms.

Make sure that you server can survive to the crash of clients (hitting Ctrl-C for example)

Make sure that a client can resist to a temporary unavailable server (a server crash for example). The idea is that the client will wait for a new server to be restarted and will then reconnect to that new server.