# Laboratory practices

# An object oriented distributed chat using RMI

Concurrency and Distributed Systems

## Introduction

The goal of this assignment is to introduce students to distributed systems and distributed applications design using objects. We will be using RMI as the underlying distributed middleware, but any other distributed object support middleware could be used. The selected application to learn and practice distributed systems topics is a distributed Chat application. There are many chat applications in the computers market and there are some standards too. Our goal is neither to develop a fully-fledged commercial-like application nor to follow a particular standard, but to focus on simple and clear object oriented distributed systems. This practice also addresses some important topics on distributed systems not specifically targeted to object oriented systems, such as name services, and the client/server paradigm. You will be using Java as your programming language and RMI as the object support runtime.

Once you complete this practice, you will have learned to:

- Identify the different processes that build up a distributed application.
- Compile and execute simple distributed applications.
- Understand the name service role.
- Understand the client/server paradigm and its object-oriented extension to design and implement distributed applications.

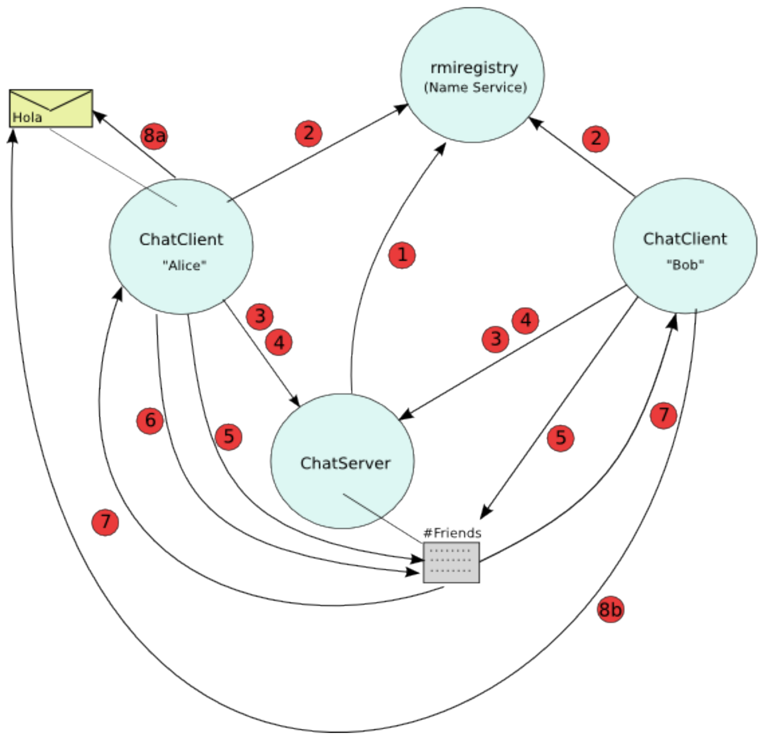
This lab practice must be developed by **groups of two students**.

You will need approximately three weeks to complete this practice. Each week you must attend one lab session, where the professor can help you with technical questions you might find. You might also need additional time to complete the practice. To that end you can use laboratory facilities during their free access periods of time. You could also use your own personal computer.

Along the practice you will notice that there are exercises to complete. It is recommended to solve them and to annotate their results in the provided boxes, to facilitate the future study of the content of the practice.

## A basic distributed Chat

The following diagram shows a basic setup for our distributed chat. In this case, you can find a name server (called *rmiregistry* in RMI), a ChatServer and two ChatClients (launched by users Alice and Bob). Those 4 processes build up one simple setup which is however more complex and interesting than a pure client/server scheme.



The diagram shows the sequence of steps that our application follows when users Alice and Bob join a chat channel named “#Friends” and start chatting. Alice sends a simple “Hello” message and both users receive it since they are connected to the channel.

Let us see which steps are taken. Each step is marked with a numbered red circle in the diagram:

1. ChatServer registers its main *ChatServer* object with the name server.
2. Both ChatClients look up for the *ChatServer* object using the name server. Notice that all three processes must agree on the *ChatServer* object name.
3. Chat clients connect to the *ChatServer*. For this, they create a local *ChatUser* object and register it in the server using the *connectUser* method of ChatServer.
4. Chat clients ask the list of channels, using the *listChannels* method of ChatServer.

5. Chat clients join channel “#Friends”. For this, they obtain first the *ChatChannel* object of the requested channel (using the *getChannel* method of *ChatServer*) and they join to this channel.
   Notice that the first user to join the channel also receives a notification when the second one joins. This channel notification is not drawn in the diagram.
6. Alice sends a simple chat message to the channel. For this, *ChatChannel* invokes *ChatUser* to broadcast messages.
7. As a result of Alice’s message, the channel broadcasts it to all connected users.
8. When users receive the message, they ask for the message content. 8a results in local invocation, whereas 8b is a remote invocation.

Notice that even though *ChatClients’* role is mainly to act as chat clients, they also act as servers, since they hold object implementations which are invoked remotely. Specifically, *ChatChannels* invoke *ChatUsers* to broadcast messages, and anyone who needs to see a given message content must invoke the message owner for its contents. In our case, *ChatClients* ask user Alice for message content “Hello”. This object invocation pattern is not common among most chat environments but it is rich enough for us to understand object oriented distributed applications.

## Laboratory facilities and recommended setup

You can use whichever software you prefer to handle java projects (BlueJ, eclipse, etc.) or you could just use a simple text editor, the standard command line compiler and the java runtime. Our recommendation is for you to use the simple command line tools, since even if you use an IDE to help in development tasks, there are certain actions that must be done using command line tools.

**Given our laboratory facilities for this practice, we recommend that you use Windows, and the *java*, *javac* and *rmiregistry* command line tools.**

Our laboratories have computers with a configured firewall, which blocks most ports. We can use port **9000-9499** for our laboratory practices. Some of our programs and the *rmiregistry* tool itself need to be configured to use these ports accordingly.

## Software provided

For this practice, you can find a package named **DistributedChat.jar**. Please download it, unpack it, and check that it contains the java source files. Check that you can compile them all (`javac *.java`) and that you can execute the basic programs.

**NOTE:** In the laboratory, ***javac*** program is at:

"**C:\Archivos de programa\java\jdk1.8.0_71\bin**"

The package contains two ready to run programs. These programs are **ChatClient.java** and **ChatServer.java**. The other files are interfaces and classes needed by these programs and altogether constitute a basic distributed chat.

To start the *ChatServer* and *ChatClients*, you need an instance of ***rmiregistry*** already running so that our chat programs can use it as a name server. Name services make it possible to register a symbolic name for a remote object and associate it with a reference, so that clients can locate the object. To start *rmiregistry*, simply execute it inside a console.

```
rmiregistry 9000
```

**NOTE:** In the laboratory, ***rmiregistry*** program is at:
"**C:\Archivos de programa\java\jdk1.8.0_71\bin**"

Parameter 9000 means to start *rmiregistry* at port 9000. If this parameter is not provided, *rmiregistry* will bind itself to its default 1099 port number. Note that you must use ports within the range 9000-9499 inside our laboratories.

**NOTA**: If you execute this order and an error "Port already in use" appears, it might be possible that another student group has already started an *rmiregistry* instance on the same virtual machine. In this case, just restart *rmiregistry* using a higher port number, for example, 9100.

To start a *ChatServer*, the following command should be executed **in the same directory where its java binary classes are located** (parameters starting with "ns", such as *nsport*, refer to the name server):

```
java ChatServer nsport=9000 myport=9001
```

This command starts a basic *ChatServer* at port 9001, which will connect to local port 9000 when it needs to connect to the *rmiregistry*. Default local port for *ChatServer* and *ChatClient* is 9001, and expected port for *rmiregistry* is 9000, so the command line provided above has the same effect as this simpler one:

```
java ChatServer
```

To start a *ChatClient*, the same parameters can be provided, plus an optional additional parameter (*nshost*) to specify the host where the name server is running. For example:

```
java ChatClient nshost=192.168.1.1 nsport=9000 myport=9002
```

With these parameters, this *ChatClient* will try to find a running *rmiregistry* at the computer 192.168.1.1 at port 9000. This example assumes that *rmiregistry* is running on that machine, but in general we will have to find out the IP of our machine, which can be done easily using e.g. the command **ipconfig** in a terminal (in Windows).

If the process *rmiregistry* has been launched on the local machine, it is not necessary to provide the host name. Thus, we could just use:

```
java ChatClient myport=9002
```

This *ChatClient* will accept connections on port 9002. If more than one client or server run on the same physical computer, we need to specify a different port number other than the default 9001, using a different port for each running client.

There is an additional parameter for both *ChatClient* and *ChatServer*, which is the *ChatServer* object name. It defaults to “TestServer”. You can override this name using the “server=” argument to both programs. For instance, the following commands start a *ChatServer* and a *ChatClient* that will use a *ChatServer* object named “NewServer”

```
java ChatServer server=NewServer
java ChatClient server=NewServer myport=8002
```

**Important: In this practice we assume that we always run *rmiregistry* and *ChatServer* at the same computer.**

## Activity 1: start chatting

This initial activity teaches you how to chat. ☺

You already know how to chat, don't you? But you have to learn how to chat using our distributed chat. Moreover, you must learn how the distributed object oriented chat is built and which object invocations occur when you perform the basic chatting activities.

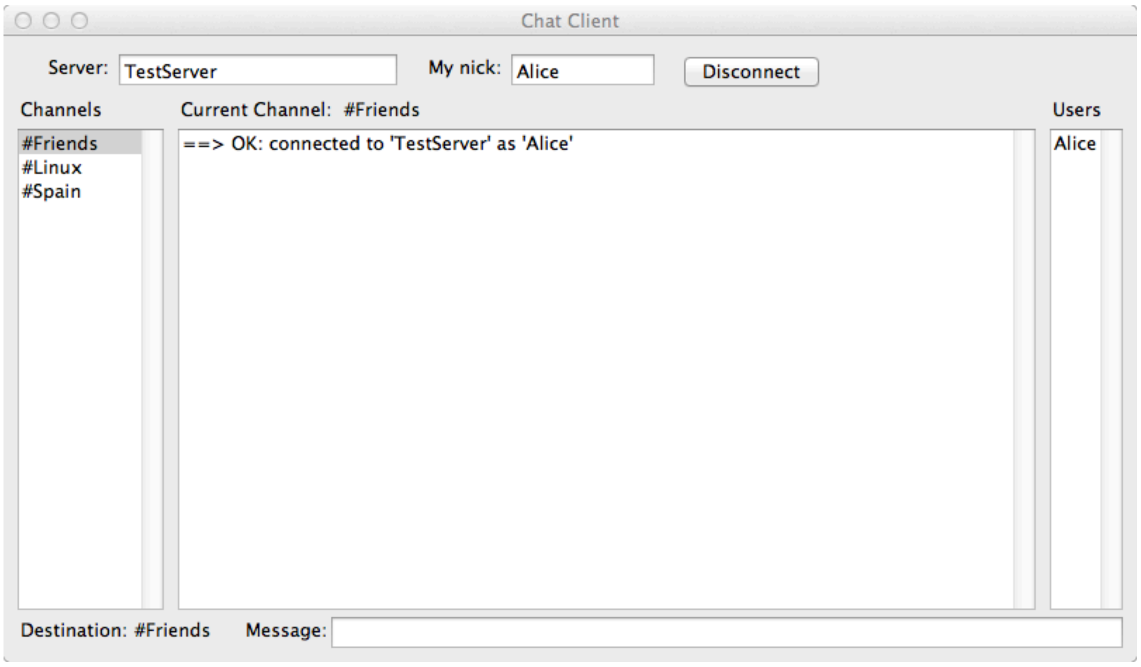
To practice chatting we need at least 2 people (for example: Alice and Bob). You must follow the steps explained previously, that is:

- Step 1: start rmiregistry
- Step 2: start a ChatServer
- Step 3: start a ChatClient for Alice. In this case, start this first *ChatClient* at the same computer where the other programs are running. Since our ChatServer uses port number 9001, you must start Alice's chat client at a different port (e.g. port 9002).
- Step 4: start a ChatClient for Bob. In this step, a second chat client is started, this time for Bob. This client can be run either in the same machine where the previous processes have been launched (using a different port number, for example 9003), or in a different one.
- Step 5: chat a little bit.

To start a ChatClient that connects to the Chat server of a different machine (e.g. the computer of the neighboring group), you have to indicate the IP of the machine where the *rmiregisty* of the neighboring group is running. For example, let us assume that this machine is 192.168.1.2 and that their rmiregistry is using port 9000. Therefore, you should start the new ChatClient as follows:

```
java ChatClient nshost=192.168.1.2 nsport=9000 myport=9004
```

The following figure shows the *ChatClient* window. You can see three areas in it. There is a top line for the server parameters. Then, there is a central scrollable area (divided in three parts) and a third area at the bottom, where you can type messages.



==> OK: connected to 'TestServer' as 'Alice'

The top line has 2 text boxes, one for the *ChatServer* name and the other where you must type in your nickname. Alice and Bob must type in their different nicknames. Once you have typed in both fields you can connect to the *ChatServer*. If connection succeeds, you will see a success message.

The scrollable area is divided in 3 areas from left to right. The leftmost area will contain the server channel list. The rightmost area will contain the channel user list and the biggest central area will contain incoming messages from both your current channel and any private messages you could receive.

Bottom line has a *Destination* label and a text box for you to type messages. *Destination* will contain the name of the channel or the user that will receive your messages. You can select a channel or a user by double-clicking on their names.

Once you double-click onto a channel you join that channel and you also leave any other channel you joined before. If you double-click onto a user you don't leave your current channel, but you just change your message destination, so that the following messages will be private messages for a particular user.

## Questions

1) *When* ChatServer *registers its main object* ChatServer *in the name server, which actions are done? Which interfaces have been extended or implemented?*

2) *When* ChatClient *looks for* ChatServer *using the name server, which actions are done? Which object name is used? What does* ChatClient *get from the name server?*

*3) When Chat clients connect to* ChatServer*, which proxies are used? Which methods of those proxies? Is any object sent as a method parameter? Why for?*

*4) When Chat clients ask for the list of channels, which proxies are used? Which methods of those proxies? Is any object sent as a parameter or returned value of these methods? Why for?*

*5) When a Chat client joins a channel, which proxies do the chat client to do this action? Which methods of these proxies?*

*6) When a Chat client joins a channel, he/she notifies the other users of the channel. For this, which proxies are used? Which methods of these proxies? Is any object sent as a method parameter? Which one and what for?*

*7) If Alice sends a simple message to the channel, which actions are done? Which objects are used? Is any new object created? If so, what is it used for?*

*8) When the channel broadcasts Alice's message to all users connected to the channel, which actions are done? Is any object sent as a method parameter?*

*9) Users, on receiving the message, ask for its content. Which actions are done? Which differences are there between a local invocation and a remote invocation?*

*10) Which objects will be created and which invocations will be done when Bob user sends a private message to Alice user?*

## Activity 2

You have to implement a *ChatRobot*. The *ChatRobot* is a process that connects to a given server and connects to a fixed channel specified as a parameter. Then, each time a user connects to that channel, our *ChatRobot* must greet him or her sending a "Hello *nick*" message to the channel, where *nick* is the nick of the user joining the channel.

To implement the *ChatRobot*, complete file **ChatRobot.java**.

### Questions

1) Describe which objects and operations are invoked each time a user connects to the #Friends channel, assuming that ChatRobot is already connected to this channel.

2) Justify the sentence "if you start more than one ChatClient or ChatRobot application in the same machine, they all must have different *myport* values. If there is only one application per machine, you do not need to modify this value".

3) The location of the name server (*nsport*, *nshost*) must be known by everyone, but clients do not need to know the host or the port corresponding to the ChatServer. Justify why.