Laboratory Practices
A distributed object-oriented chat based on RMI
(3 sessions)

Concurrency and Distributed Systems

Introduction

The aim of this practice is to introduce students to distributed systems and object-oriented design of distributed applications. The application selected to learn and practice with distributed systems is a distributed chat application. Although there are different standards and a multitude of chat applications on the market, our proposed chat application does not imitate any specific application, nor does it follow any standard, giving priority to the clarity and simplicity of its implementation. Specifically, in this practice we develop an object-oriented distributed chat based on Java-RMI. Therefore, RMI will be used as the underlying distributed middleware, but any other middleware that supports distributed objects could be used.

Once completed this practice, the student will have learned to:

- Identify the different components that make up a distributed application.
- Understand the function of the name service, which registers symbolic names for remote services or objects, and associates with each name the reference to the service/object. This mechanism allows clients to locate these registered services or objects.
- Introduce the Client/Server paradigm and use this paradigm to design and implement distributed applications.
- Analyze a complete distributed application.
- Learn how to compile and run simple distributed applications.

The estimated duration of this practice is three weeks. Each week you must attend a laboratory session where you will be able to discuss your progress and resolve any doubts you may have with your teacher. Please note that you will need to allocate some time from your personal work to complete the practicum. The proposed organization into sessions is as follows:

- Session 1.- activities 1, 2 and 3.
- Session 2.- activity 4 and start activity 5.
- Session 3.- complete activity 5.

This practice is prepared to be carried out on **Polilabs DSIC-Linux** systems.

Activity 1

In this first activity, we will launch the components that make up the Java-RMI-based object-oriented distributed chat application. Specifically, the application consists of three types of components, which must be launched in the same order in which they are introduced here:

1) A name server (*NameServer*), listening on a given port, through which it receives requests from servers (*ChatServer* in our case) to register remote objects (i.e. to associate a logical name to a given reference to a remote object); or it receives requests from clients (*ChatClient* in our case) to obtain the reference associated to a logical name.
2) At least one chat server (*ChatServer*), identified by a name (by default "TestServer"). We can launch several chat servers and register them on the same name server, if each one has a different unique logical name.
3) Any number of chat clients (*ChatClient*). Each client is identified by a unique name or nickname and will connect to the specified chat server.

Instructions

1. Enter **DSIC-LINUX** from Polilabs.
2. When you start the session, you will be in a machine named `ldsic-vdiNN`, where NN are two digits that identify the specific virtual machine. **DSIC has 20 virtual machines, ranging from `ldsic-vdi01` to `ldsic-vdi20`**. If we open a terminal (right mouse button on the screen background, option '`Open in a terminal`') we will have a terminal like the one in the following figure, in whose **prompt** appears:
   - Our user (example: `esarvil@upvnet.upv.es`)
   - The machine name (e.g. `ldsic-vdi01`)
   - The current directory (in this case `~/Escritorio`), where ~ corresponds to our home directory, which is also reached by executing the command `cd`

3. On your W drive, create the `csd` directory and download there the **DistributedChatRMI.tgz** file provided for the practice (available in the Lessons of practice 4 on the PoliformaT site of the subject), unzip it and access the newly created directory. Specifically, do the following:

```
$ cd $HOME/W
$ mkdir csd
$ cd csd
$ tar xvf DistributedChatRMI.tgz
$ cd DistributedChatRMI
```

2. In the DistributedChatRMI directory you will find the files `NameServer.java`, `ChatServer.java` and `ChatClient.java` corresponding to the application provided in this practice. Compile these files:

```
$ javac *java
```

We will now launch the different components of our application (`NameServer`, `ChatServer` and several `ChatClients`). To do this, we will launch each component on a different terminal (in total, we will have 4 terminals at the same time). It is recommended to resize each terminal and move them to have all 4 visible simultaneously.

IMPORTANT: we assume that on all terminals we are located in the directory where we have the result of the compilation (where the `.class` files are located).

3. On the first terminal launch the **name server** (*NameServer*), indicating the port number on which it will listen for incoming requests. For this practice we will use the port range 9000 - 9499. Only one component can listen on a port of a machine. Therefore, for different components, different ports will have to be used. For example, we can launch the name service on port 9000, as shown below:

```
esarvil@upvnet.upv.es@ldsic-vdi01:~/W/csd/DistributedChatRMI$ java NameServer port=9000
OK ==> 'NameServer' running at ldsic-vdi01.upvnet.upv.es:9000
I'm good boy, don't kill me please!
```

Note: the following are the allowed parameters for the **NameServer** application

```
USAGE HELP

    java NameServer [host=...] [port=...]


USAGE -- valid arguments:

   host = <Host name or address to be included into my object references>
   port = <Port number where to listen for connections>


Example:
   java NameServer port=10000
```

These parameters indicate: the *host* where the name server is launched (by default, it assumes *localhost*, i.e., the local machine); and the *port* number on which the name server will listen. If no port number is specified, *NameServer* will default to port 9000.

Note: when launching the name server, in case the port to be used is already occupied, a message "Port already in use" will appear, as shown in the following figure.

```
Error. port 9000 already in use.
Please specify different port using parameter 'port'
```

This means that a server has already been launched listening on that port of that virtual machine. For example, if we have already launched a component using that same port, or another student has launched a component in that port using the same virtual machine. To solve this problem, retry the java `NameServer` command with another port number, within the range indicated for the practice.

4. On the second terminal, launch the **chat server**, with the following command (assuming the name server has previously been launched on port 9000 on the local machine):

```
$ java ChatServer nsport=9000 serverName="TestServer"
```

Note: The following are the allowed parameters for the **ChatServer** application

```
USAGE HELP

    java ChatServer [nsHost=...] [nsPort=...] [serverName=...] [host=...]


USAGE -- valid arguments:

   nsHost = <host where name server is running>
   nsPort = <port where name server is listening>
   serverName = <ChatServer object name to bind/lookup into NameServer>
   host = <Host name or address to be included into my object references>


Examples:
   java ChatServer nshost=localhost  nsport=9010  serverName=test
   java ChatServer nshost=EVIRL-039-OK  nsport=9010  serverName=test
```

These parameters indicate[1] : the host (*nshost*) where the name server is launched (by default, it assumes *localhost*, i.e., the local machine); the port number (*nsport*) on which the name server listens (by default, it assumes port 9000); and the name (*serverName*) to assign to the chat server (by default "TestServer").

For example, we can launch the chat server (assuming we have previously launched the name server on port 9000 and on our local machine, in this case `ldsic-vdi01`), as shown below:

```
esarvil@upvnet.upv.es@ldsic-vdi01:~/W/csd/DistributedChatRMI$ java ChatServer
Channel '#Spain' created.
Channel '#Linux' created.
Channel '#Friends' created.
OK ==> 'TestServer' Running at ldsic-vdi01.upvnet.upv.es:9001
```

If all goes well, a message will appear indicating that the server ("TestServer" in this case) is active, listening on the first free port it has found from port 9000 onwards (in the example, it is listening on port 9001), and that it has created certain channels (in this case, the #Spain, #Linux and #Friends channels).

Important: in case the name server has not been previously launched, or has been launched on a different host and/or port than the one indicated in the ChatServer call, a connection error will appear, similar to the following figure. You will have to repeat steps 3 and 4 accordingly.

```
esarvil@upvnet.upv.es@ldsic-vdi01:~/W/csd/DistributedChatRMI$ java ChatServer
Channel '#Spain' created.
Channel '#Linux' created.
Channel '#Friends' created.
Error connecting to NameServer: java.rmi.ConnectException: Connection refused to host: localhost; nested exception is:
        java.net.ConnectException: Conexión rehusada (Connection refused)
```

5. On the third and fourth terminals launch both **chat clients**, with the following command (assuming that the name server has previously been launched on port 9000 on the local machine, and that ChatServer has been registered with the default name "TestServer"):

   ```
   $ java ChatClient nsport=9000 serverName="TestServer"
   ```

[1] The *host* parameter will not be used in the development of this practice on the DSIC-Linux lab machines. This parameter allows, in the case of using a machine that has more than one IP, to specify the specific IP to be registered in the reference.

Note: The following are the allowed parameters for the **ChatClient** application

```
    java ChatClient [nsHost=...] [nsPort=...] [serverName=...] [host=...] [nick=...] [channel=...]


USAGE -- valid arguments:

   nsHost = <host where name server is running>
   nsPort = <port where name server is listening>
   serverName = <ChatServer object name to bind/lookup into NameServer>
   host = <Host name or address to be included into my object references>
   nick = <User name to use when connecting to a ChatServer>
   channel = <Channel to auto-join when program starts>


Examples:
   java ChatClient
   java ChatClient nsport=9010
   java -cp lib/*:.: ChatClient nshost=EVIRL-039-OK  nsport=9010 nick=pau
```

These parameters indicate: the machine (*nshost*) where the name server has been launched (by default, it assumes *localhost*); the port number (*nsport*) on which the name server listens (by default, it assumes port 9000); the logical name (*serverName*) to query the name server for the associated object reference (by default, "TestServer"); the *host* or own address (by default, localhost); the *nick* or user name to connect with; and the *channel* to join automatically when the application is launched. All these parameters are optional.

For example, we can launch the chat server (assuming we have previously launched the name server on port 9000 and on our local machine, in this case `ldsic-vdi01`), as shown below:

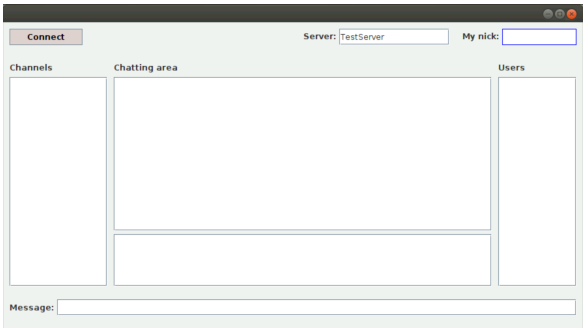
```
esarvil@upvnet.upv.es@ldsic-vdi01:~/W/csd/DistributedChatRMI$ java ChatClient
OK ==> 'ChatClient' running at ldsic-vdi01.upvnet.upv.es:9002
```

A message will appear indicating that ChatClient is active on the first free port it has found starting from port 9000 (in the example, it is active on port 9002). The client is the only component with a graphical interface, the operation of which is explained in the next section. Possible error messages (e.g. if it is unable to contact the name server) are displayed in this interface.

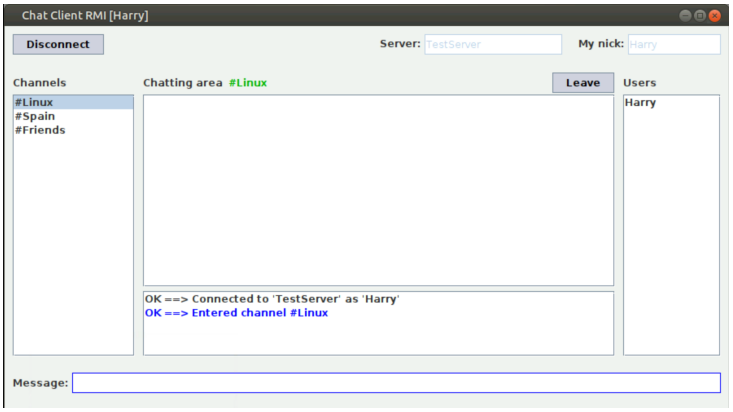
Note: you can launch as many clients as you want, but we recommend a minimum of three clients to check that the chat application works properly, so that two clients talk to each other within the same channel (and spread the information between them) and a third client is attached to a different channel (and does not receive messages from the other channels). For each new client, another terminal would be opened and the ChatClient component would be launched there.

Verification

If everything has worked correctly, for each ChatClient launched, a graphical interface similar to the following image will be displayed.



By assigning a name in "My nick" to the user and pressing the *Connect* button, it will connect to the chat service provided by the server (in this case, called "TestServer"), showing us the list of channels. The following figure corresponds to user Harry's client after connecting to the TestServer chat server and joining the #Linux channel. A *Disconnect* button appears to leave the chat, as well as a *Leave* button to leave the channel.



Note: the user's nickname must be unique. If a nickname that is already in use is proposed, the server generates an error message. (`java.rmi.RemoteException: User exists`).

Once the connection is established, the server responds with the current list of channels, which is displayed in the left column. The channels are named #xxx, and the client can join a channel from the list by double-clicking on the desired name (or by clicking the *Join* button). When a user joins a channel, the list of users subscribed to that channel appears in the right column (in the figure above, Harry is the first user to subscribe to the #Linux channel).

The text area at the bottom allows you to write a message, which can be addressed to:

- A specific user, if we have previously selected the name of a user from the right column. In this case it is a **private message**. The name of the selected user will appear in the text "Chatting area".
- A channel, if we have previously selected the name of a channel from the left column. In this case it is a **public message**, as it is broadcast to all users subscribed to that channel. In the text "Chatting area" the name of the selected channel will appear.

The central area is the message area (not editable). In the upper part of the central area, public and private messages sent to us by other users are displayed. The lower part of the central area shows the messages coming from the ChatServer itself (e.g., connection message to the server, channel entry message, channel exit message), as well as any error messages that appear during the execution of the ChatServer.

After launching several clients (each from a different terminal, but all from your local machine), check that the chat is working correctly:

- On the terminal where you have launched the nameserver, you should see:
    - A "rebind" type message indicating that the ChatServer has registered with the name server, showing its logical name and the associated endpoint. In our example:

```
==> rebind (TestServer, {IChatServer, endpoint:[ldsic-vdi01.upvnet.upv.es:9001]})
```

    - Several "resolve" messages, one for each ChatClient that has been launched. This *resolve* message indicates that the name server has been requested to provide us with the reference associated with a logical name (TestServer in our case). In our example:

```
==> resolve (TestServer) --> {IChatServer, endpoint:[ldsic-vdi01.upvnet.upv.es:9001]})
```

- On the ChatServer terminal, a message of type `User 'Harry' connected` shall be displayed for each user connecting to the chat application, and a message of type `User 'Harry' disconnected` shall be displayed for each user disconnecting.

- On the terminal of each ChatClient client, a message will be displayed indicating the port that client is using. For example:

```
OK ==> 'ChatClient' running at ldsic-vdi01.upvnet.upv.es:9002
```

  Check that each client has been assigned a different port number.

- Using the graphical interface of each client, try sending both **messages to the channel** (e.g., double-click on channel #Linux to display "Chatting area #Linux"), and **private messages** to another user (e.g. double-click on user Sally to display "Chatting area Sally"). Also try joining and leaving channels. Note all messages displayed at the bottom of the "Chatting area".

Testing the use of the application

Once the basic functionality has been tested, different variants are proposed:

1st) Check the operation of the channels. To do this, check that when a channel is selected, the set of clients connected to that channel is updated correctly.

2nd) Launch more than one chat server, giving each one a different name. For example, you can include your username (e.g. "TestServer-esarvil").

```
$java ChatServer nsport=9000 serverName="TestServer-esarvil"
```

Verify that a new "rebind" message associated with this launched chat server is displayed on the name server terminal.

3rd) Check the operation of your other chat servers by launching additional clients that connect to them. For example:

```
$java ChatClient nsport=9000 serverName="TestServer-esarvil"
```

Check that a new "resolve" message associated with this launched chat client is displayed on the name server terminal. And note that in the graphical interface we will be connecting to this new server.

4th) Check the dissemination of public messages (sent to the channel). Check that each server has its own set of channels, although the channel names may be the same. Also make sure that the user's registration/deregistration is broadcast to all the users of the corresponding chat, but does not affect the rest of the servers.

Questions

a) Does a user connecting to a channel receive the previous conversation in that channel?
b) Are the messages persistent? No
c) How many components have you launched? How many ports have been used? Are there components sharing the same port number?
d) What exactly do the "rebind" and "resolve" messages on the name server terminal mean? How many messages of each type did you get and what do they represent?

Activity 2

In the previous activity all the application components have been deployed on the same machine, for convenience and ease of deployment. In this second activity we will check that the components of our application can reside on any machine. To do so, we will deploy the different components in different virtual machines.

To launch from our machine commands on a different machine we will use the `ssh` (secure Shell) command, which allows secure access to another machine, so that all information transferred is encrypted.

In this activity we will launch each component of the application on a different machine and we will check that the application works correctly, regardless of the physical location of the different components. In particular, we will see that the registration and deregistration of users is broadcasted and that both public and private messages are delivered correctly.

Instructions

In the description of the instructions that follow, it is assumed that we are initially connected to the `ldsic-vdi01` machine; that we will launch the name server on the `ldsic-vdi11` machine and port 9000; the ChatServer server, with logical name "TestServer-esarvil" on the `ldsic-vdi12` machine; a ChatClient client on the `ldsic-vdi13` machine; and another ChatClient client on the `ldsic-vdi14` machine. The following table shows the description of the machines and ports on which each component of the application was launched in the example described in these instructions.

Component	**Running on machine**	**Port**	**Orden utilizada para lanzar a ejecución el componente**
NameServer	`ldsic-vdi11`	9000	`java NameServer port=9000`
ChatServer (`TestServer-esarvil`)	`ldsic-vdi12`	9001	`java ChatServer nsHost="ldsic-vdi11" nsPort=9000 serverName="TestServer-esarvil"`
ChatClient	`ldsic-vdi13`	9001	`java ChatClient nsHost="ldsic-vdi11" nsPort=9000 serverName="TestServer-esarvil" nick="Alice"`
ChatClient	`ldsic-vdi14`	9001	`java ChatServer nsHost="ldsic-vdi11" nsPort=9000 serverName="TestServer-esarvil" nick="Bob"`

Each student should make the appropriate modifications to the instructions indicated, to suit his or her initial connection machine and the machines where he or she wishes to launch the various components. To facilitate this task, the student should complete **Table 1** (above) to clearly detail on which machine and port each component of the application resides.

As in the previous application, we will launch each component in a different terminal (in total, we will have 4 terminals at the same time). It is recommended to resize each terminal and move them to have all 4 visible simultaneously. In each terminal, we will have to connect to a different machine (via ssh) and place ourselves in the directory where we have the executables of our application (e.g. directory `W/csd/DistributedChatRMI`), as shown in step 1.

1. From the **first terminal** (we assume that you are initially connected to `ldsic-vdi01`), connect to another virtual machine (for example, to `ldsic-vdi11`). Accept the prompt (`Are you sure you want to continue connecting? yes`) and enter your user password. Once connected, go to the `DistributedChatRMI` directory, and launch the **name server**

(NameServer). If the port to be used is already busy, enter another port number within the range 9000 - 9499. In this example port 9000 has been used.

```
$ssh -X ldsic-vdi11
$cd W/csd/DistributeChatRMI
$java NameServer port=9000
```

```
esarvil@upvnet.upv.es@ldsic-vdi01:~$ ssh -X ldsic-vdi11
The authenticity of host 'ldsic-vdi11 (172.23.2.241)' can't be established.
ECDSA key fingerprint is SHA256:Yeh69+xM/bexVHnDwbgh3HNNCW4NnLPst5uWkjsgDn0.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'ldsic-vdi11,172.23.2.241' (ECDSA) to the list of known hosts.
esarvil@upvnet.upv.es@ldsic-vdi11's password:
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.8.0-63-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage
Last login: Thu Oct 28 13:31:56 2021 from 172.23.2.242
esarvil@upvnet.upv.es@ldsic-vdi11:~$ cd W/csd/DistributedChatRMI/
esarvil@upvnet.upv.es@ldsic-vdi11:~/W/csd/DistributedChatRMI$ java NameServer port=9000
OK ==> 'NameServer' running at ldsic-vdi11.upvnet.upv.es:9000
I'm good boy, don't kill me please!
```

2. From the **second terminal**, connect to another virtual machine (for example, to `ldsic-vdi12`). Accept the request (`yes`) and enter your user password. Once connected, go to the `DistributedChatRMI` directory, and launch the **chat server** (ChatServer). You must specify the host name and port number on which you have previously launched your name server. In addition, you must specify a logical name for ChatServer, preferably including your username as a suffix. In this example, "TestServer-esarvil" has been used as the logical name.

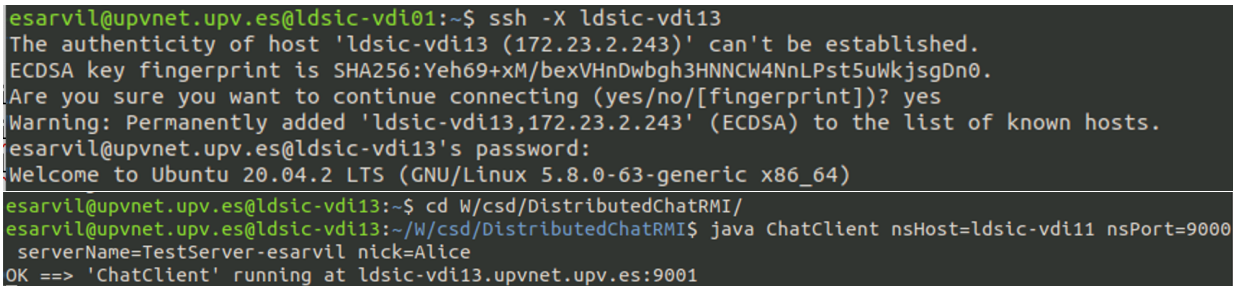
```
$ssh -X ldsic-vdi12
$cd W/csd/DistributeChatRMI
$java ChatServer nsHost="ldsic-vdi11" nsport=9000
      serverName="TestServer-esarvil"
```

```
esarvil@upvnet.upv.es@ldsic-vdi07:~/W/csd/DistributedChatRMI$ ssh -X ldsic-vdi12
The authenticity of host 'ldsic-vdi12 (172.23.2.242)' can't be established.
ECDSA key fingerprint is SHA256:Yeh69+xM/bexVHnDwbgh3HNNCW4NnLPst5uWkjsgDn0.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'ldsic-vdi12,172.23.2.242' (ECDSA) to the list of known hosts.
esarvil@upvnet.upv.es@ldsic-vdi12's password:
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.8.0-63-generic x86_64)
esarvil@upvnet.upv.es@ldsic-vdi12:~$ cd W/csd/DistributedChatRMI/
esarvil@upvnet.upv.es@ldsic-vdi12:~/W/csd/DistributedChatRMI$ java ChatServer nsHost="ldsic-vdi11" nsPort=9000
serverName="TestServer-esarvil"
Channel '#Spain' created.
Channel '#Linux' created.
Channel '#Friends' created.
OK ==> 'TestServer-esarvil' Running at ldsic-vdi12.upvnet.upv.es:9001
```

3. From the **third terminal**, connect to another virtual machine (for example, to `ldsic-vdi13`), go to the `DistributedChatRMI` directory and launch a first **chat client** (ChatClient). You must specify the host name and port number on which you have previously launched your name server, as well as the logical name under which you have registered ChatServer (e.g., "TestServer-esarvil"). You can also specify a nickname for the client (e.g., Alice). When the graphical interface appears, click the Connect button to connect to the chat application.

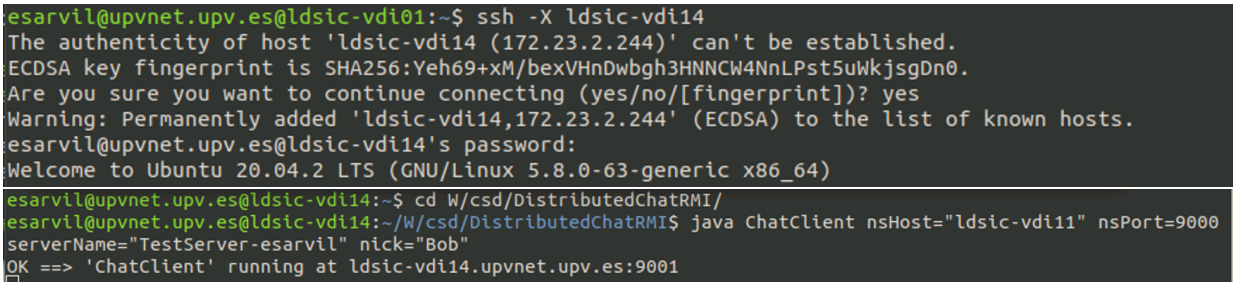
```
$ssh -X ldsic-vdi13
$cd W/csd/DistributeChatRMI
```

```
$java ChatClient nsHost="ldsic-vdi11" nsPort=9000
      serverName="TestServer-esarvil" nick="Alice"
```



4. From the **fourth terminal**, connect to another virtual machine (e.g., to `ldsic-vdi14`), and launch a second **chat client** (ChatClient), like the previous step. Use a different nickname for this client (e.g., Bob). When the graphical interface appears, click the Connect button to proceed with the connection to the chat application.

```
$ssh -X ldsic-vdi14
$cd W/csd/DistributeChatRMI
$java ChatClient nsHost="ldsic-vdi11" nsport=9000
      serverName="TestServer-esarvil" nick="Bob"
```



To make it easier for you, in each step of the instructions, fill in the line of the table corresponding to the terminal indicated, as appropriate:

Terminal	Component	Running on machine	Port	Command used to launch the component into execution
1º	NameServer	`ldsic-vdi`	9000	java NameServer
2º	ChatServer: `TestServer-`	`ldsic-vdi`		java ChatServer nsHost="ldsic-vdi11.upvnet.upv.es" nsport=9000 serverName="TestServer-squigra"
3º	ChatClient	`ldsic-vdi`		java ChatClient nsHost="ldsic-vdi11.upvnet.upv.es" nsport=9000 serverName="TestServer-squigra" nick="Alex"
4º	ChatClient	`ldsic-vdi`		java ChatClient nsHost="ldsic-vdi11.upvnet.upv.es" nsport=9000 serverName="TestServer-squigra" nick="Alex"

Table 1. To be completed by the student.

Verification

If everything has worked correctly, after launching each component from a different virtual machine, we will have the following:

- On the terminal where the name server has been launched, at least one message of type "rebind" associated to our ChatServer and several messages of type "resolve" associated to our clients should appear. Check that the addresses and port numbers indicated match those noted in the table you have previously filled in.
- On the ChatServer terminal, a "User XXX connected" message should appear for each user connected to our application. Check that the connected users match those listed in your table.
- Using the graphical interface of each client, check that the clients can join and leave the channels, as well as send each other both messages to the channel and private messages.

Activity 3

Once the deployment of components on different machines has been tested, different scenarios are proposed:

1. Launch the ChatServer and clients locally, using the NameServer launched by your lab practice teacher. To do this, the teacher is required to inform in advance of the virtual machine name and port number on which he/she has launched his/her NameServer. In this way, all students in the class will connect to the same NameServer. It is recommended to add the login of each student as a suffix to the logical name of the ChatServer (e.g., "TestServer-esarvil", where esarvil is a student login), to register the chat server properly on the name server, without interfering with that of other students.

2. Now use the NameServer launched by another student and launch your own ChatServer and clients to use the chat application. In this case only the ChatServers are required to have a different unique name among those connected to the same name server.

3. Finally, launch the clients locally, using other students' ChatServers, to check that messages are still being broadcast correctly to remote clients. The NameServer to be used can the teacher's NameServer, or the other student's NameServer from the previous step.

Activity 4

In this activity we will analyze the design of the distributed chat application provided in this practice, in order to understand its inner workings. This knowledge will serve as a basis for activity 5.

In the source code provided we find:
- The java files corresponding to the components used in the previous activities (*NameServer*, *ChatServer*, *ChatClient*), as well as the *ChatRobot.java* file, which will be used in activity 5.
- A set of directories grouping different files, according to their functionality:

- **Faces**: contains the remote interfaces (i.e., they define the interface of the remotely accessible objects). Therefore, they describe the interaction between the fundamental objects of the application:
  - `IChatChannel`: interface that provides a channel of the chat. A channel maintains the set of connected users, which allows the broadcasting of messages.
  - `IChatMessage`: interface that provides a chat message.
  - `IChatServer`: interface that provides a ChatServer.

    - `IChatUser`: interface provided by a chat user (e.g., to be notified of messages).
    - `INameServer`: interface that provides a name server to its clients.
    - MessageListener: NON-remote interface that defines the behavior of a client to 'listen' to messages.

- **Impl**: contains the java classes that implement the previous remote interfaces.
    - `ChatChannelImpl`: implements IChatChannel.
    - `ChatMessageImpl`: implements IChatMessage
    - `ChatServerImpl`: implements IChatServer
    - `ChatUserImpl`: implements IChatUser
    - `NameServerImpl`: implements INameServer

- **Lib**: external library that can be used to improve the GUI appearance.
- **Ui**: contains the code required to implement a client's GUI.
- **Utils**: contains utility classes that facilitate the implementation of the application.
- **Utils_rmi**: contains utility classes to simplify the configuration of the components, allowing, for example, the definition of the arguments they receive and their default values.

To understand how the application works, it is essential to analyze the interfaces in the **Faces** directory, as well as the **ChatServer** and **ChatClient** code. The classes that implement these interfaces (contained in the **Impl** directory) may be of interest to consult very specific aspects. The rest of the code (*lib*, *ui*, *utils* and *utils.rmi* directories) is not necessary to understand how chat works. The following figure summarizes the basics of these interfaces and classes:

Interface	- Methods	Classes implementing it	

IChatChannel	`IChatUser[] join(IChatUser)`
	`void leave(IChatUser)`
	`String getName()`
	`void sendMessage(IChatMessage)`
		ChatChannelImpl	Maintains channel name and set of channel users.
			Allows to add (*join*) / remove (*leave*) channel users.
			Implements message broadcasting (*sendMessage*) in the channel, iterating over users.

IChatMessage	`IChatUser getSender()`
	`Remote getDestination()`
	`String getText()`
	`Boolean isPrivate()`
		ChatMessageImpl	Keeps sender, destination and message text.
			Indicates whether the message is private.

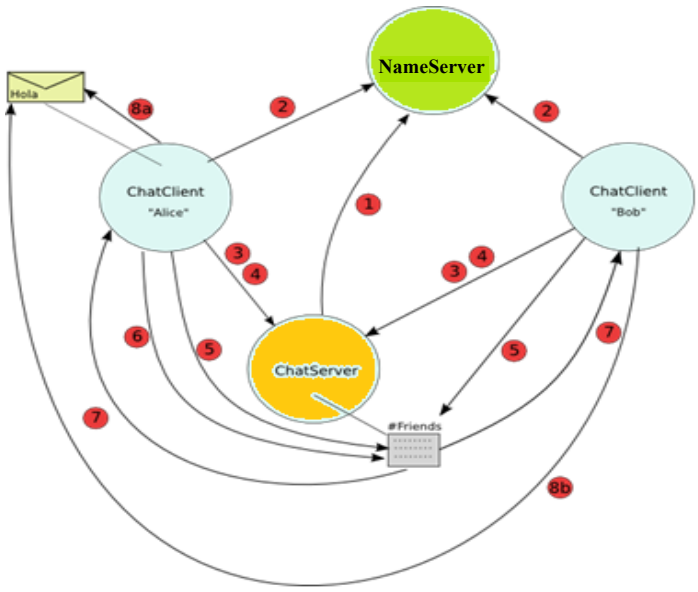
IChatServer	`IChatChannel createChannel(String)`
	`IChatChannels[] listChannels()`
	`IChatChannel getChannel(String)`
	`IChatUser getUser(String)`
	`boolean connectUser(IChatUser)`
	`boolean disconnectUser(IChatUser)`
		ChatServerImpl	Allows to create a channel.
			Maintains the list of *channels* and *users*.
			Returns a channel or a user from the name.
			Allows to connect or disconnect a user to the chat application.

IChatUser	`String getNick()` `void sendMessage(IchatMessage)`	ChatUserImpl	Represents the user. Keeps his/her nickname. Allows to send a message to the user

INameServer	`InameServer getNameServer(String,int)`
	`void bind(String, Remote)`
	`void rebind(String, Remote)`
	`void unbind(String)`
	`Remote lookup(String)`
		NameServerImpl	Allows to easily locate the NameServer.
			Maintains pairs (name, Remote reference).
			Allows adding new entries (*bind*, *rebind*), deleting entries (*unbind*), and searching by name (*lookup*).

MessageListener	`void messageArrived(IchatMessage)`	ChatClient ChatRobot	Allows the reception of messages.

The following diagram shows a basic layout of the distributed chat. In this case, there is a NameServer, a ChatServer and two ChatClients (corresponding to the users Alice and Bob). These 4 processes form a simple system that is, however, more complex and interesting than a pure client/server scheme.



The diagram shows the ordered steps our application follows when users Alice and Bob join a chat channel named "#Friends" and start chatting. Alice sends a "Hello" message and both users receive it when they are connected to the channel.

The steps followed are detailed below. Each step is marked with a numbered red circle on the diagram:

1. ChatServer registers its main object "ChatServerImpl" in the name server, with its logical name (by default "TestServer").
2. The two ChatClients look up the "ChatServerImpl" object using the name server and get its IChatServer proxy. Note that the three processes must agree on the logical name of the "ChatServerImpl" object, which in this example is "TestServer".
3. The Chat clients connect to the ChatServer. For this purpose, they create a local *ChatUserImpl* object and register it on the server, using the *connectUser* method of IChatServer.
4. The Chat clients ask for the list of channels, using the *listChannels* method of IChatServer.
5. The chat clients join the "#Friends" channel. To do this, they first get the *IChatChannel* proxy of the requested channel (using the *getChannel* method of IChatServer) and join that channel, using the *join* method of IChatChannel.
    a. Note that the first user to join the channel also receives a notification when the second client joins. This channel notification is not drawn in the diagram.
6. Alice sends a single chat message to the channel, using the *sendMessage* method of IChatChannel.
7. From that send, the channel broadcasts the message to all connected users (to the channel). To do so, *ChatChannelImpl* invokes the *sendMessage* method of *IChatUser* of each user of the channel to retransmit the messages.
8. The users, when they receive the message, parse its content using the **messageArrived** method. 8a represents a local invocation, while 8b is a remote invocation.

Note that, although the role of ChatClients is mainly to act as chat clients, they also act as servers, since they have objects (*ChatUserImpl*) that are invoked remotely. *ChatChannelImpl* invokes methods of *IChatUser* to broadcast messages, of which their references (not the message content) are sent. Anyone who needs to see the content of a given message must invoke methods on the message reference. This

object invocation pattern is not common in most chat environments but is complete enough to understand distributed object-oriented applications.

Questions

a) In which order should the different components (NameServer, ChatServer, ChatClient) be started?
b) How does a ChatClient locate the ChatServer from its name?
c) Assuming that a ChatClient knows the location of the NameServer (*nshost*, *nsport*) and the name of the ChatServer (*serverName*), indicate what sequence of methods ChatClient invokes to get the list of channels.
d) Note that both IChatChannel and IChatUser define the method `void sendMessage(IChatMessage)`. What is the difference between the two implementations?
e) Which remote objects does ChatServer define?
f) Indicate whether a ChatClient creates any remote object and whether it registers it in the name server.

Activity 5

In this activity we will complete the content of the ChatRobot class, in order to implement a "bot" of our chat application. This ChatRobot will be a process that:

- Receives command line arguments: *nshost*, *nsport*, *serverName*, *channelName*, *nick*
  - *nshost* and *nsport* are used to locate the NameServer.
  - *channelName* indicates which channel to connect to.
  - *nick* indicates the nick or name that the ChatRobot will have as a chat user.
- Once launched, it will not need any further interaction with the user launching the program. Thus, unlike a ChatClient, ChatRobot does not have a graphical interface and does not allow interaction through the console. The only possible external action is to stop its execution using Ctrl-C.
- You will have to connect to the server and channel indicated using the nickname provided, and monitor all incoming messages that arrive:
  - If it is the notice that a new client X has connected to that channel, ChatRobot should send a public "Hello X" greeting message through the channel.
  - If it is a private message from a ChatClient, it should reply with another private message stating "I am robot 'Nick', and the only answer I have learned so far is that 1+1 = 2".

For the implementation of this process, the skeleton of the ChatRobot class is available, where only the declaration of the class and its main method appear. The following table breaks down this activity into a series of steps that will allow the functionality to be completed progressively. Each step:

- Describes the desired functionality in that step.
- Details the instructions to achieve it.
- Details the mechanism to validate the functionality.
- Raises questions that help to complete the instructions or to understand the design decisions made in each step.

ChatRobot has some similarities with ChatClient, so the ChatClient code can be consulted as a reference to facilitate the resolution of the different steps.

Step	Description	Instructions	Checking
1	Get reference to NameServer	Get host and port of NameServer from command line. Get the reference with `INameServer.getNameServer`	Print the reference with `RemoteUtils.remote2String`
2	Get reference to ChatServer	Get the name of the ChatServer from the `conf` object. Get the reference with the `lookup` operation of the `INameServer` object.	Print the reference with `RemoteUtils.remote2String`
3	Connect to ChatServer	Get the nickname from `conf`. Create a `ChatUserImpl` object to represent the robot. Register it to the server using the ChatServer's `connectUser` method.	Throw exception in case of error. If there are no exceptions, display message on screen indicating that it has connected.
4	Obtain the channel list	Use the `listChannels` method of the ChatServer to get the list. Check that the channel `channelName` specified in `conf` is in the list.	Print the channel list. Display the `conf` channel and the search result on the screen.
5	Join the channel	Get the channel with the `getChannel` method of the ChatServer. Use `join` and get the list of users.	Launch a pair of ChatClients (let's say A and B) connected to the channel where the bot will be launched. The bot should display the list of users (the nickname of the bot, A and B should be displayed). Check that the list matches the list displayed by A and B.
6	Send public message	Build message with `ChatMessageImpl`. Send the message over the channel.	Check that the clients connected to the channel receive it.
7	Send a private message	Build message with `ChatMessageImpl`. Send the message on the target client.	Check that the target client receives it (by selecting the ROBOT in its user list).
8	Receive and display messages	Ensure ChatRobot implements `MessageListener`. Get sender, destination, and message text. Determine whether it is private, from the server, or public.	Launch messages from clients and display the details of each message on screen.
9	Responding appropriately to messages	Respond appropriately to JOIN messages. Respond appropriately to private messages.	Verify performance according to specification.

Below are more detailed steps and instructions for you to perform in the ChatRobot code.

Step 5.1.- Get reference to the NameServer

The robot must collect from the command line the arguments `nshost`, `nsport`, `serverName`, `channelName`, and `nick`. The `ChatConfiguration` class is used for this.
Once the value of *nshost* and *nsport* are known, they must be used to access the NameServer, as a previous step to locate the ChatServer.

Instructions:

1. Get host and port of the NameServer from the command line. Note how ChatClient resolves that same issue using a *conf* variable of type ChatConfiguration.
2. Get the reference to the NameServer with `INameServer.getNameServer`. To better understand how to perform this step, you can also analyse the ChatClient code and see how it is resolved there.

Checking:

- Print the obtained reference. The `remote2String` method of the *RemoteUtils* class (in the *rmi-utils* directory) allows us to transform any remote reference into a String, which we can display on screen. It is interesting to check the information associated to the reference.

Questions:

a) What does the *conf* object do in ChatClient, how is it initialised, what are its default values?
b) What does the text printed from the reference indicate?

Step 5.2.- Get the reference to the ChatServer

When the ChatServer is started, it creates a remote object and registers it in the NameServer under a given name X. The robot receives in the serverName argument the name of the ChatServer to contact (let's assume it is X). To gain access to the corresponding remote object, the robot must ask the NameServer for the remote object called X. This procedure is like the one performed by ChatClient.

Instructions:

1. Get the name of the ChatServer provided in the serverName argument when launching ChatRobot from the command line.
2. Get the reference to the ChatServer with the INameServer `lookup` operation.
3. The obtained reference must be stored in a variable of type IChatServer. Possible exceptions must be handled (e.g., cannot find nameserver, cannot find object, etc.).

Checking:

- Print the message indicating the possible exception, or if everything went well the value of the obtained reference (using `remote2String`).

Questions:

a) What happens if we have not given the name of the server as an argument?
b) What happens if the name we are looking for does not exist?
c) Why do we not need to know the host and port where ChatServer is located?

Step 5.3.- Connect to the ChatServer

Once the bot has access to the server, it needs to connect to the server. In this way, ChatServer can:

- Validate the nick provided (each chat user's nick must be unique and cannot be empty).
- Get a reference to the corresponding IChatUser: each chat user implements a remote object and passes the reference to ChatServer, so that ChatServer can invoke certain operations on each registered user.

To register, ChatRobot must invoke the `connectUser` method of IChatServer, passing as argument an object of type ChatUser (containing the nickname of the robot).

Instructions:

1. Create a `ChatUserImpl` object to represent the robot (with the nickname specified in the command line).
2. Register that object on the server using the `connectUser` method of IChatServer.

Checking:

- Check for possible exceptions (e.g., server not found, nickname is empty, etc.), and if there are no exceptions display a message indicating a successful connection.
- Launch NameServer, a ChatServer and a client. Connect the client to a channel. Now launch your ChatRobot on that same channel and see if that connection affects the client.

Questions:

a) What if the nick is already being used by another chat user?
b) In the ChatClient code, `this` is used as the second argument of the ChatUserImpl constructor. Can the robot do the same?
c) When launching the ChatRobot, what can we observe in the client interface? Why?

Step 5.4.- Get the channel list

A user can connect to any channel (and subsequently disconnect). ChatServerImpl implements the `listChannels` method, which returns the list of available channels. This information should be obtained before attempting to connect to a channel. Specifically:

- ChatClient needs the list of channels to display them on screen and allow the user to choose the desired channel.
- ChatRobot receives the channel to connect to as an argument (*channelName*) but should verify that the value of that argument appears in the channel list.

Instructions:

1. Get the list of channels using the `listChannels` method of IChatServer.
2. Check that the channel provided as an argument on the command line (i.e., *channelName*) also appears in the channel list.

Checking:

- Print the channel list.
- Display the value of *channelName* (command line argument) and the search result in the channel list.

Questions:

a) How do you get the channel names from what the `listChannels` method of IChatServer returns?

Step 5.5.- Join channel

To join a channel, you must first obtain the reference to the corresponding channel (with the `getChannel` method of IChatServer) and invoke the `join` method on that reference. When connecting to a channel, the list of users in the channel is obtained, which will include the user who has asked to join the channel.

Instructions:

1. Get the channel with the `getChannel` method of IChatServer.
2. Use the `join` method on the channel and get the list of users. Display the list on the screen.

Checking:

- Launch one or more ChatClients and join them to a given channel. You must keep these clients active for the rest of the following steps.
- Launch your ChatRobot on the same channel. In the list of users displayed by the bot, both the launched clients' nicknames and the bot's nickname should appear as users. In addition, each client's interface should display the bot's nickname in the list of users.

Questions:

a) When a customer joins the channel, the channel notifies the other users of the channel. What method is used for this?

Step 5.6.- Send public message

In this step, the necessary instructions for the robot to send messages to the channel it is connected to will be added to the ChatRobot. To do this, the robot will construct an object of type ChatMessageImpl and send it to the channel, and the channel will take care of broadcasting the message to the rest of the users connected to the channel. When a client receives the message, it displays it in its GUI.

Instructions:

1. Construct a message using the `ChatMessageImpl` class. The content of the message can be simply in the style "Hello everyone".
2. Send the message over the channel, using the `sendMessage` method.

Checking:

- Launch your ChatRobot over the channel where multiple users are previously connected.
- Check that your robot's greeting message is received by all of them (i.e., it appears in the GUI of each user connected to that channel).

Questions:

a) What does the channel do to broadcast the message to the users of that channel?

Step 5.7.- Send private message

In the previous step, public messages (messages to the channel) were sent. But you can also send private messages, i.e., directly to another user. In this case, the ChatRobot will select, from the list of users it has received from the channel where it is connected, the first user in the list (other than the robot itself) and will send a private message in the style "Hello, I'm a bot".

Instructions:

1. Construct a message using the `ChatMessageImpl` class. The content of the message can be simply in the style "Hello, I am a bot".
2. Go through the list of users and send the first user in the list (other than the robot itself) the message previously constructed, using the `sendMessage` method on its corresponding IChatUser.

Checking:

- Launch your ChatRobot on the channel where several users are already connected.
- Check that your bot's "Hello, I'm a bot" greeting message only reaches the specified user, and

not the other users in the channel.

Questions:

a) How does the channel intervene in the sending of a private message?
b) How do you know if a message is private or public, and is the same message builder used for public (to the channel) and private messages?

Step 5.8.- Receiving and displaying messages

To receive messages, the `MessageListener` interface must be implemented. This interface consists of a single method, `void messageArrived(IchatMessage)`, whose code must be implemented in the classes that make use of this interface, as has been done in the ChatClient class. Therefore, the `messageArrived` method must be completed in the ChatRobot class to ensure that ChatRobot properly implements the `MessageListener` interface.

The code of the `messageArrived` method must decompose the message, obtaining its sender, destination, and specific text. To do so, the methods provided by the IChatMessage interface will be used to obtain the different components of the message (i.e., sender, destination, text), as well as to determine if the message is private, if it comes from the server, or if it is a public message (i.e., sent to the channel).

Instructions:

1. Verify that ChatRobot implements the `MessageListener` interface.
2. In the `messageArrived` method, get the sender, destination, and message text.
3. In that method, determine whether the message is private, coming from the server, or public.
4. Display all this information on the screen.

Checking:

- Launch your ChatRobot on the channel where several users are previously connected.
- Launch messages from multiple clients (both messages to the channel and private messages to the robot) and verify that the robot displays on its associated terminal the details of each message it has received.
- Have some clients leave the channel and join the channel. Check that the robot displays on its associated terminal the details of the connection and disconnection messages it receives from the server.

Questions:

a) When receiving a message, how do we differentiate whether it is a public or private message?
b) How can we know if a message has been sent by the server itself?
c) What is the difference between a public message and a private message with respect to the destination component?
d) Where is the `messageArrived` method invoked?

Step 5.9.- Respond appropriately to messages

The robot specification indicates that it should respond to two types of messages (and therefore ignore the rest):

- Messages corresponding to the registration of a user. These are public messages coming from the server and whose text starts with "JOIN". The reply must be a public message greeting the new user, as it was done in step 5.6.
- Private messages. The reply must be another private message, as in step 5.7.

Instructions:

1. Write in the `messageArrived` method of ChatRobot the necessary code to classify the types of messages (private/public, sent by the server or not, JOIN or other).
2. For a JOIN message, write the code needed to get the nickname of the new user. Construct a message with content "Hello 'nick'" and send it through the channel.
3. For a private message sent by a user, construct a message with a content like "I'm a robot, and the only answer I've learned so far is that 1+1 = 2" and send it to that user's ChatUser.

Checking:

- Launch the robot and connect it to a channel. Then launch client(s) and observe the result on the robot terminal and on the interface of the other clients.
- From one of the clients, send a private message to the robot and observe the response.

Questions:

a) Explain what objects and operations are invoked when a client connects to the #Friends channel, assuming that ChatRobot is already connected to that channel.

b) Assuming that NameServer is launched on machine A and ChatServer is launched on machine B, and that we launch ChatRobot on machine C and ChatClient on machine D, and that everyone knows the name server data (nsport and nshost), state for each object ChatServerImpl, ChatChannelImpl, ChatUserImpl, ChatMessageImpl, ChatClient, ChatRobot:
   - Which machine each one is on.
   - Whether a single instance or multiple instances of each object are used.
   - How instances can communicate with each other (how their references are obtained, and whether invocations are local or remote).