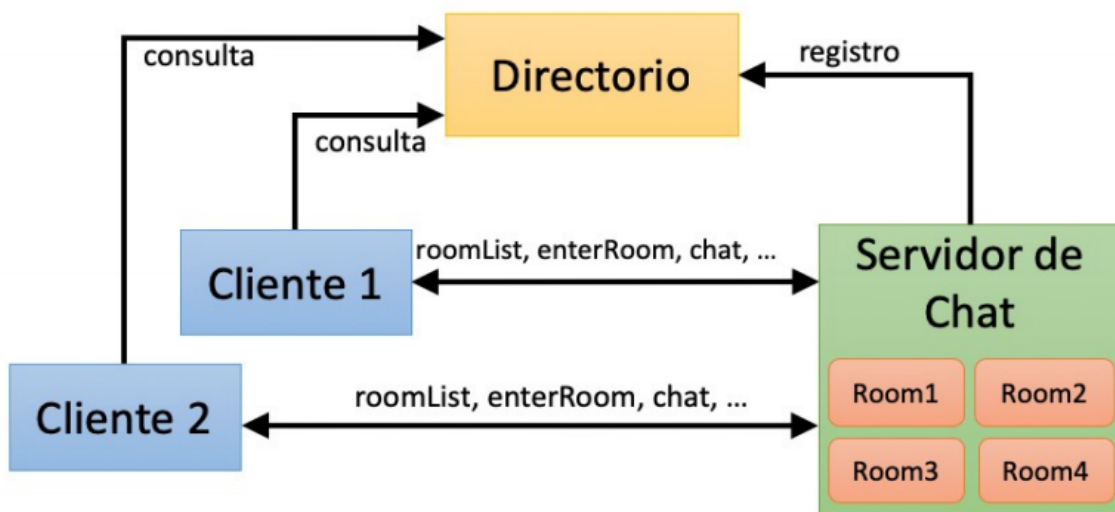


# DOCUMENTACIÓN PRACTICA FINAL

## NanoChat 2020/21

Redes de Comunicaciones



## **ÍNDICE.**

<b>1.- INTRODUCCIÓN.</b>	<b>2</b>
1.1.- Identificador de protocolo y tipo de mensajes a implementar.	2
<b>2.- DISEÑO DE LOS PROTOCOLOS.</b>	<b>3</b>
2.1.- Formato de los mensajes UDP con el directorio.	3
2.2.- Formato e implementación de los mensajes TCP - Lenguaje de Marcas.	4
2.3.- Ejemplos de intercambio de mensajes TCP.	7
2.4.- Autómatas de protocolo.	10
2.4.1.- Autómata del Servidor de Chat.	10
2.4.2.- Autómata del Cliente.	10
2.4.3.- Autómata del Directorio.	11
<b>3.- DETALLES SOBRE LOS PRINCIPALES ASPECTOS DE LA IMPLEMENTACIÓN.</b>	<b>11</b>
3.1.- Mecanismo de gestión de salas.	11
3.2.- Mejoras adicionales implementadas.	12
3.2.1.- Notificar a los demás usuarios de la sala las entradas/salidas de otros usuarios en las salas conforme se vayan produciendo.	12
3.2.2.- Posibilidad de crear nuevas salas en el servidor.	13
3.2.3.- Ver el histórico de mensajes enviados a la sala.	14
3.2.4.- Capturar la excepción que se produce en el servidor de chat cuando un usuario aborta su conexión y tratarla adecuadamente.	15
3.2.5.- Modificar la clase Directory y DirectoryThread para que acepte como parámetro la IP de la interfaz donde va a estar corriendo ese servicio.	16
<b>4.-CONCLUSIÓN.</b>	<b>18</b>

## 1.- INTRODUCCIÓN.

La realización de esta práctica consiste en la programación de una aplicación llamada *NanoChat*, la cual va a permitir mediante una entidad denominada **directorío** que una entidad denominada **servidor de chat** registre en él su dirección de red. Posteriormente otra entidad denominada **cliente** podrá consultar en el directorío cuales son los servidores de chat disponibles y obtener su dirección de red. Una vez la obtienen se puede conectar a ellos e interactuar, permitiendo la comunicación entre distintos clientes a través de él.

Esta práctica tiene dos partes diferenciadas, la comunicación entre '*directorío-servidor*' y '*directorío-cliente*' la cual se implementa usando el protocolo de transporte **UDP**, y la segunda parte que involucra la comunicación entre '*servidor de chat-cliente*' y '*cliente-servidor de chat*' que se implementa utilizando el protocolo de transporte **TCP**.

Esta comunicación se lleva a cabo en los siguientes pasos:

1. Un servidor de chat le indica al directorío que se quiere registrar como servidor y se registra.
2. Un cliente cualquiera solicita al directorío un servidor de chat disponible para su protocolo.
3. Una vez el cliente ha obtenido la dirección del servidor de chat a través del directorío y se conecta a él, podrá realizar las siguientes acciones:
  - a. Registrarse con un nick.
  - b. Solicitar la lista de salas disponibles.
  - c. Entrar a una sala específica.
  - d. Crear nuevas salas.
  - e. Solicitar una lista de los comandos disponibles y su descripción.
  - f. Desconectarse del servidor.
4. En el caso de que el usuario se haya conectado, registrado con un nick y entrado a una sala, podrá realizar las siguientes acciones:
  - a. Enviar un mensaje a los otros usuarios de la sala.
  - b. Solicitar información acerca de la sala actual.
  - c. Solicitar el historial de mensajes enviados en la sala desde que se creó.
  - d. Salir de la sala.

### 1.1.- Identificador de protocolo y tipo de mensajes a implementar.

El identificador de protocolo que nos corresponde se obtiene al sumar el número de DNI de los miembros del grupo y aplicar el módulo 128, puesto que la codificación del protocolo en nuestros mensajes UDP se va a implementar como un solo byte.

Puesto que a pesar de haber realizado la práctica de manera individual, inicialmente estaba previsto realizarla en pareja con el compañero Adrian Balsalobre Vera, [adrian.balsalobrev@um.es](mailto:adrian.balsalobrev@um.es) con DNI 48756821, y siendo mi DNI 48753907, obtenemos:

$$48756821 + 48753907 = 97.510.728 \text{ MOD } 128 = 72$$

-Como el protocolo es **PAR**, entonces el formato de los mensajes a implementar será el **lenguaje de marcas**.

## 2.- DISEÑO DE LOS PROTOCOLOS.

### 2.1.- Formato de los mensajes UDP con el directorio.

Empleamos el protocolo **UDP** para la comunicación entre el servidor-directorio y cliente-directorio. Los servidores de chat enviarán mensajes para poder registrarse en el directorio, y los clientes consultarán al directorio cuales son los servidores de chat disponibles.

El formato empleado para estos mensajes será el formato binario multiformato, para esta comunicación dispondremos de los siguientes mensajes con el siguiente formato:

**2.1.1.- Solicitud de registro del servidor en el directorio:** Mensajes enviados por el servidor de chat al directorio, para registrarse en él como servidor disponible.

<b>OPCODE</b> (1 byte)	<b>PUERTO</b> (4 bytes)	<b>PROTOCOLO</b> (1 byte)
---------------------------	----------------------------	------------------------------

En nuestro caso OPCODE es *OPCODE\_REGISTRO\_SERVER* = 1. Ejemplo:

1	6969	72
---	------	----

**2.1.2.- Confirmación de registro del servidor en el directorio:** Mensaje que le envía el directorio a un servidor de chat cuando lo ha registrado con éxito.

<b>OPCODE</b> (1 byte)
---------------------------

En nuestro caso OPCODE es *OPCODE\_OK\_REGISTRO\_SERVER* = 2. Ejemplo:

2
---

**2.1.3.- Consulta de un cliente al directorio:** Estos mensajes son enviados por el cliente al directorio para preguntarle cuál es la dirección de un servidor para un protocolo dado, que se registró previamente.

<b>OPCODE</b> (1 byte)	<b>PROTOCOLO</b> (1 byte)
---------------------------	------------------------------

En nuestro caso OPCODE es *OPCODE\_CONSULTA* = 3. Ejemplo:

3	72
---	----

**2.1.4.- Información sobre el servidor de chat:** Este mensaje es enviado por el directorio al cliente, con la información correspondiente al servidor de chat por el que le consultó en caso de que existiese registrado en el directorio.

<b>OPCODE</b> (1 byte)	<b>Dirección IP</b> (4 bytes)	<b>PUERTO</b> (4 bytes)
---------------------------	----------------------------------	----------------------------

En nuestro caso OPCODE es *OPCODE\_DATOS\_SERVER* = 4. Ejemplo:

4	127.0.0.1	6969
---	-----------	------

**2.1.5.- Servidor de chat no disponible:** Mensaje enviado por el directorio a un cliente que solicitó un servidor de chat especificando un protocolo mediante una consulta y en el directorio no hay ningún servidor para dicho protocolo.

<b>OPCODE</b> (1 byte)
---------------------------

En nuestro caso OPCODE es *OPCODE\_SERVER\_NOTFOUND* = 5. Ejemplo:

5
---

## **2.2.- Formato e implementación de los mensajes TCP - Lenguaje de Marcas.**

En nuestro caso para la codificación de los mensajes **TCP** entre el cliente y el servidor, vamos a emplear un lenguaje de marcas, para ello vamos a definir los siguiente mensajes:

**2.2.1.- NCOperationCodeMessage:** En este tipo de mensajes únicamente se envía un código de operación que describe una acción básica que se va a llevar a cabo:

```
<message>
  <operation> OP_CODE </operation>
</message>
```

Donde **OP\_CODE** es un código para codificar un mensaje en el programa, puede ser:

- OP\_NICK\_OK**: Para confirmar que el registro del nick de un usuario ha sido correcto.
- OP\_NICK\_DUP**: Para indicarle a un usuario que el nick que ha introducido ya está registrado
- OP\_GET\_ROOMS**: Para solicitarle al servidor cuales son las salas disponibles.
- OP\_ENTER\_ROOM\_OK**: Para confirmarle al cliente el acceso a una sala correctamente.
- OP\_GET\_ROOM\_INFO**: Para solicitar información de la sala en la que nos encontramos.
- OP\_GET\_HISTORY**: Para solicitar el historial de la sala en la que nos encontramos.
- OP\_ENTER\_ROOM\_FAIL**: Para indicar al cliente que no ha sido posible acceder a una sala.

**-OP\_CREATE\_ROOM\_OK:** Para confirmar al cliente que la creación de la sala que ha solicitado se ha realizado con éxito.

**-OP\_CREATE\_ROOM\_FAIL:** Para indicar al cliente que no ha sido posible crear la sala que ha solicitado.

**2.2.2.- NCListRoomsMessage:** Este mensaje será creado y enviado por el servidor para informar a un cliente que está en estado 'REGISTRATION' sobre cuales son las salas disponibles en el servidor, los usuarios que las componen y la fecha en la que se envió el último mensaje.

También es usado para informar a un cliente que está en estado 'IN\_ROOM' acerca de la información de la sala en la que se encuentra. En este caso habrá que asegurarse que la lista de salas de la cual nos está ofreciendo información en este caso es únicamente una (únicamente podrá aparecer un conjunto de etiquetas <room></room>).

```
<message>
  <operation> OP_CODE </operation>
  <roomList>
    <room>
      <roomName> Sala1 </roomName>
      <nick> Usuario1 </nick>
      <nick> Usuario2 </nick>
      <time> Fecha </time>
    </room>
    <room>
      <roomName> Sala2 </roomName>
      <nick> Usuario3 </nick>
      <nick> Usuario4 </nick>
      <nick> Usuario5 </nick>
      <time> Fecha </time>
    </room>
  </roomList>
</message>
```

Donde **OP\_CODE** es un código para codificar un mensaje en el programa, puede ser:

**-OP\_ROOM\_LIST:** Para solicitar la información de las salas disponibles en el servidor.

**-OP\_ROOM\_INFO:** Para solicitar la información de la sala en la que se encuentra el usuario.

**2.2.3.- NCRoomMessage:** Este mensaje es utilizado cuando además de un código de operación para indicar la acción a realizar, necesitamos enviar también algún dato asociado a la operación que queremos realizar.

```
<message>
  <operation> OP_CODE </operation>
  <name> Texto </name>
</message>
```

Donde **OP\_CODE** es un código para codificar un mensaje en el programa, puede ser:

**-OP\_NICK:** Para indicar que un usuario se quiere registrar en el servidor, y en el campo name indica el nombre con el que se desea registrar.

**-OP\_ENTER\_ROOM:** Para indicar que un usuario quiere acceder a una sala de chat, en el campo name indica el nombre de la sala a la que se desea unir.

**-OP\_EXIT\_ROOM:** Para indicar que un usuario quiere salirse de una sala de chat, en el campo name indica el nombre de la sala de la que se quiere ir.

En este caso también se podría haber usado el tipo de mensaje '*NCOperationCodeMessage*' ya que el nombre de la sala no es obligatorio que se lo envíe el cliente al servidor, puesto que este ya sabe en qué sala se encuentra y como solo puede estar en una sala a la vez, puede saber cual es el nombre de la sala que se quiere salir.

**-OP\_NOTIFY\_ENTER:** Para el servidor indicarle a todos los usuarios de una sala que un nuevo usuario se ha unido a la sala, en el campo name indica el nombre del usuario que se acaba de unir.

**-OP\_NOTIFY\_EXIT:** Para el servidor indicarle a todos los usuarios de una sala que un usuario se ha salido de la sala, en el campo name indica el nombre del usuario que se acaba de salir.

**-OP\_CREATE\_ROOM:** Para indicar que un usuario quiere crear una sala de chat, en el campo name indica el nombre de la sala que desea crear.

**2.2.4.- NCMessage:** Este mensaje se emplea para codificar la información necesaria (nombre del usuario que envía el mensaje, el propio mensaje y la fecha) cuando un cliente desea enviar un mensaje de texto al resto de clientes de la sala en la que se encuentra. Este tipo de mensaje se lo envía un cliente al servidor, y el servidor se encargará de reenviarlo a todos los demás usuarios de la sala en la que se encuentra el cliente.

```
<message>
  <operation> OP_CODE </operation>
  <nick> Usuario1 </nick>
  <text> String </text>
  <time> Fecha </time>
</message>
```

En este caso **OP\_CODE** siempre será **OP\_MESSAGE**.

**2.2.5.- NCHistoryMessage:** Este mensaje se emplea para codificar la información necesaria que le tiene que enviar el servidor de chat, a un cliente que ha solicitado el historial de mensajes que se han enviado en la sala desde que se creó. En este caso la información a codificar será la lista con todos los mensajes de texto que se han enviado.

```
<message>
  <operation> OP_CODE </operation>
  <text> String 1 </text>
  <text> String 2 </text>
  ...
  <text> String n </text>
</message>
```

En este caso **OP\_CODE** siempre será **OP\_HISTORY**.

\*\*puesto que esto se trata de una mejora esto se explicará más en detalle en el [apartado 3.3.3](#).

### 2.3.- Ejemplos de intercambio de mensajes TCP.

PETICIÓN usuario que se quiere registrar.	RESPUESTA del servidor (una u otra).
<pre>&lt;message&gt; &lt;operation&gt;Nick&lt;/operation&gt; &lt;name&gt;pedro&lt;/name&gt; &lt;/message&gt;</pre>	<pre>&lt;message&gt; &lt;operation&gt;Nick Correcto&lt;/operation&gt; &lt;/message&gt;</pre>
	<pre>&lt;message&gt; &lt;operation&gt;Nick Duplicado&lt;/operation&gt; &lt;/message&gt;</pre>

PETICIÓN de un usuario que quiere saber cuales son las salas de chat disponibles.	RESPUESTA del servidor.
<pre>&lt;message&gt; &lt;operation&gt;Get Rooms&lt;/operation&gt; &lt;/message&gt;</pre>	<pre>&lt;message&gt; &lt;operation&gt;Room List&lt;/operation&gt; &lt;RoomList&gt; &lt;Room&gt; &lt;RoomName&gt;room_b&lt;/RoomName&gt; &lt;Nick&gt;alba&lt;/Nick&gt; &lt;Time&gt;1617531439614&lt;/Time&gt; &lt;/Room&gt; &lt;Room&gt; &lt;RoomName&gt;room_a&lt;/RoomName&gt; &lt;Nick&gt;luis&lt;/Nick&gt; &lt;Nick&gt;pepe&lt;/Nick&gt; &lt;Time&gt;1617531389881&lt;/Time&gt; &lt;/Room&gt; &lt;/RoomList&gt; &lt;/message&gt;</pre>

PETICIÓN de un usuario que se encuentra en una sala, para obtener información de esa misma sala.	RESPUESTA del servidor.
<pre>&lt;message&gt; &lt;operation&gt;Get Room Info&lt;/operation&gt; &lt;/message&gt;</pre>	<pre>&lt;message&gt; &lt;operation&gt;Room Info&lt;/operation&gt; &lt;RoomList&gt;</pre>



	<pre> &lt;Room&gt; &lt;RoomName&gt;room_a&lt;/RoomName&gt; &lt;Nick&gt;luis&lt;/Nick&gt; &lt;Nick&gt;pedro&lt;/Nick&gt; &lt;Nick&gt;pepe&lt;/Nick&gt; &lt;Time&gt;1617531389881&lt;/Time&gt; &lt;/Room&gt; &lt;/RoomList&gt; &lt;/message&gt; </pre>
--	--

<b>PETICIÓN</b> de un usuario para entrar en una sala específica. En el caso de que la sala no exista se creará y entrará en ella.	<b>RESPUESTA</b> del servidor (una u otra).
<pre> &lt;message&gt; &lt;operation&gt;Enter Room&lt;/operation&gt; &lt;name&gt;room_a&lt;/name&gt; &lt;/message&gt; </pre>	<pre> &lt;message&gt; &lt;operation&gt;Enter Room OK&lt;/operation&gt; &lt;/message&gt; </pre> <p>En el caso de que el acceso sea correcto el servidor de chat enviará el siguiente mensaje a los usuarios que ya se encontraban en la sala para informarles de que el nuevo usuario se unió.</p> <pre> &lt;message&gt; &lt;operation&gt;Notify Enter&lt;/operation&gt; &lt;name&gt;luis&lt;/name&gt; &lt;/message&gt; </pre>
	<pre> &lt;message&gt; &lt;operation&gt;Enter Room Fail&lt;/operation&gt; &lt;/message&gt; </pre>

<b>PETICIÓN</b> de un usuario para enviar un mensaje de texto al resto de usuarios de una sala.	<b>RESPUESTA</b> del servidor a todos los usuarios de la sala menos al que lo envió.
<pre> &lt;message&gt; &lt;operation&gt;Message&lt;/operation&gt; &lt;Nick&gt;pedro&lt;/Nick&gt; &lt;Text&gt;Hola!&lt;/Text&gt; &lt;Time&gt;0&lt;/Time&gt; &lt;/message&gt; </pre>	<pre> &lt;message&gt; &lt;operation&gt;Message&lt;/operation&gt; &lt;Nick&gt;pedro&lt;/Nick&gt; &lt;Text&gt;Hola! &lt;/Text&gt; &lt;Time&gt;1617532669038&lt;/Time&gt; &lt;/message&gt; </pre>

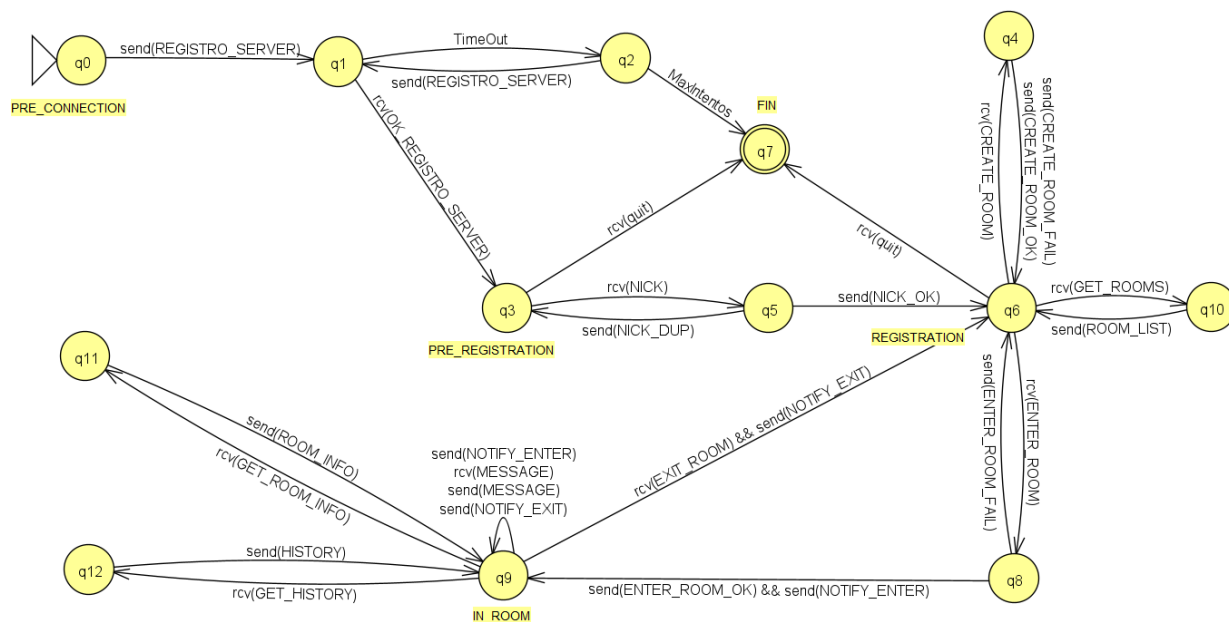
<b>PETICIÓN</b> de un usuario solicitando el historial de mensajes de la sala en la que se encuentra actualmente.	
<pre>&lt;message&gt; &lt;operation&gt;Get History&lt;/operation&gt; &lt;/message&gt;</pre>	
<b>RESPUESTA</b> del servidor.	
<pre>&lt;message&gt; &lt;operation&gt;History&lt;/operation&gt; &lt;Text&gt;* User 'pedro' joined the room&lt;/Text&gt; &lt;Text&gt;* User 'luis' joined the room&lt;/Text&gt; &lt;Text&gt;[Sun Apr 04 12:31:22 CEST 2021] luis: Hola!&lt;/Text&gt; &lt;Text&gt;* User 'juan' joined the room&lt;/Text&gt; &lt;Text&gt;[Sun Apr 04 12:32:48 CEST 2021] juan: Buenos Dias!&lt;/Text&gt; &lt;Text&gt;[Sun Apr 04 12:33:18 CEST 2021] pedro: Como estais?&lt;/Text&gt; &lt;Text&gt;* User 'juan' left the room&lt;/Text&gt; &lt;/message&gt;</pre>	

<b>PETICIÓN</b> de un usuario para salir de la sala en la que se encuentra actualmente.	<b>RESPUESTA</b> el usuario que solicitó salir de la sala no obtiene respuesta del servidor. Se toma como que esta solicitud siempre será exitosa.  Pero el resto de usuarios de la sala reciben el siguiente mensaje que les envía el servidor para informales que dicho usuario salió.
	<pre>&lt;message&gt; &lt;operation&gt;Exit Room&lt;/operation&gt; &lt;name&gt;RoomA&lt;/name&gt; &lt;/message&gt;</pre> <pre>&lt;message&gt; &lt;operation&gt;Notify Exit&lt;/operation&gt; &lt;name&gt;luis&lt;/name&gt; &lt;/message&gt;</pre>

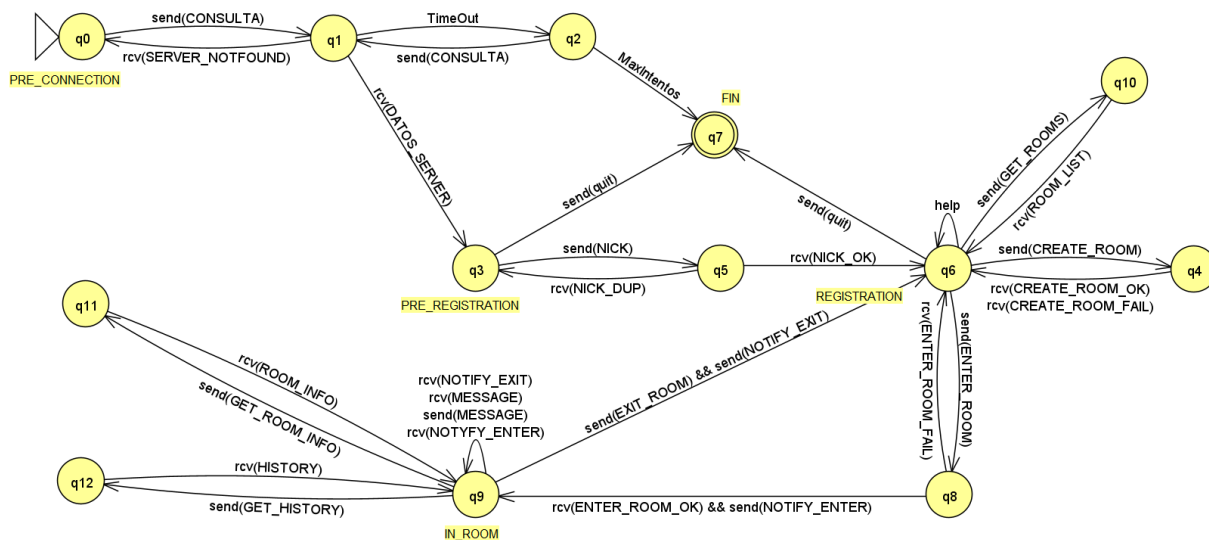
<b>PETICIÓN</b> de un usuario para crear una nueva sala específica. En el caso de que la sala ya exista no se creará y devolverá error.	<b>RESPUESTA</b> del servidor (una u otra).
<pre>&lt;message&gt; &lt;operation&gt;Create Room&lt;/operation&gt; &lt;name&gt;room_b&lt;/name&gt; &lt;/message&gt;</pre>	<pre>&lt;message&gt; &lt;operation&gt;Create Room OK&lt;/operation&gt; &lt;/message&gt;</pre>
	<pre>&lt;message&gt; &lt;operation&gt;Create Room Fail&lt;/operation&gt; &lt;/message&gt;</pre>

## 2.4.- Autómatas de protocolo.

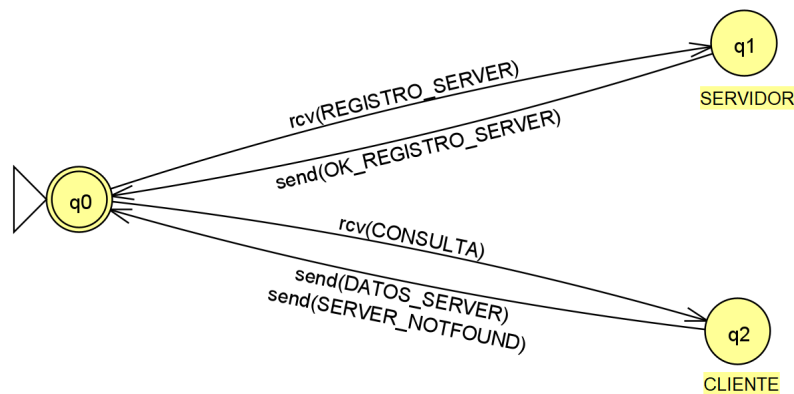
### 2.4.1.- Autómata del Servidor de Chat.



### 2.4.2.- Autómata del Cliente.



### 2.4.3.- Autómata del Directorio.



## 3.- DETALLES SOBRE LOS PRINCIPALES ASPECTOS DE LA IMPLEMENTACIÓN.

### 3.1.- Mecanismo de gestión de salas.

Cuando un usuario ya se encuentra registrado en un servidor de chat, a través del comando *'roomlist'* tiene la posibilidad de observar cuales son las salas de chat que se encuentran disponibles. Según mi implementación cuando un servidor de chat está recién creado, solamente existe una única sala con el nombre *'room\_a'*. Los usuarios podrán acceder y salir de las salas empleando los comandos *enter <nombre\_sala>* y *exit*, respectivamente.

Los usuarios también tendrán la posibilidad de crear nuevas salas (explicado en el [apartado 3.3.2](#)), esto ocurrirá en el caso de que emplee el comando *create <nombre\_sala>* y en el servidor aun no haya ninguna sala creada con el nombre *'nombre\_sala'*, en ese caso el servidor la creará, si por el contrario ya existe una sala con el nombre especificado, informará mediante un mensaje al usuario que la creación de la sala no se ha podido llevar a cabo.

Para facilitar la implementación de los comandos *'enter'* y *'create'*, y puesto que el nombre de las salas es el identificador de las mismas, se pueden usar todos los caracteres pero en el caso de ser caracteres se tomarán como su representación en minúscula.

Además, la implementación de estos comandos será *'case insensitive'*, el nombre de la sala que se le pase a estos comandos será transformada a minúscula, por lo que si suponemos que existen las salas *'room\_a'* y *'room\_b'* si introducimos el comando *'enter Room\_A'*, el servidor nos introducirá a la sala *'room\_a'*. De igual forma si introducimos el comando *'enter Room\_B'*, el servidor nos dirá que no se ha podido crear la sala (ya existe una con ese nombre).

Segun mi implementación, también cabe la posibilidad de borrar salas, pero esto no se realizará mediante ningún comando, en mi caso, esto ocurrirá cuando una sala se quede vacía, es decir, cuando de dentro de una sala salgan todos los usuarios que se encontraban en ella, en el momento que se sale el último, el servidor la eliminará de la lista de salas disponibles.

## 3.2.- Mejoras adicionales implementadas.

### 3.2.1.- Notificar a los demás usuarios de la sala las entradas/salidas de otros usuarios en las salas conforme se vayan produciendo.

Resulta interesante, siendo un cliente de chat saber cuando alguien se une o se sale de la sala en la que te encuentras enviando mensajes, de ahí la motivación de implementar dicha mejora. Para esto se emplean mensajes de tipo *NCRoomMessage*, ya que así en el campo de operación, le podremos indicar mediante un *opcode* u otro si vamos a notificar una entrada o una salida y mediante el campo *'name'* el nombre del usuario que se une o se sale de la sala.

La idea de funcionamiento consiste en que cuando el servidor recibe el comando de que un usuario desea entrar en una sala, además de responderle a ese cliente indicando que ha entrado correctamente a la sala, también envía al resto de usuarios de la sala un mensaje de los que hemos indicado anteriormente, en ese caso como opcode tendrá *OP\_NOTIFY\_ENTER*.

De forma similar ocurre cuando el servidor recibe un mensaje para salir de una sala por parte de un usuario, el servidor se encargará de enviarle al resto de usuarios de la sala un mensaje de los que hemos indicado anteriormente, en ese caso como opcode tendrá *OP\_NOTIFY\_EXIT*.

```
NanoChat shell
For help, type 'help'
* Connecting to the directory...
* Connected to /127.0.0.1:6969
(nanoChat) nick pedro
* Your nickname is now pedro
(nanoChat) enter RoomB
* You have joined the room 'RoomB'
(nanoChat-room) * Message received from server...
* User 'luis' joined the room
(nanoChat-room) * Message received from server...
[Fri Apr 02 16:35:44 CEST 2021] luis: Hola!
(nanoChat-room) * Message received from server...
[Fri Apr 02 16:36:03 CEST 2021] luis: Adios!
(nanoChat-room) * Message received from server...
* User 'luis' left the room
(nanoChat-room)
```

Usuario 1: pedro

```
NanoChat shell
For help, type 'help'
* Connecting to the directory...
* Connected to /127.0.0.1:6969
(nanoChat) nick luis
* Your nickname is now luis
(nanoChat) enter RoomB
* You have joined the room 'RoomB'
(nanoChat-room) send Hola!
(nanoChat-room) send Adios!
(nanoChat-room) exit
* Your are out of the room
(nanoChat)
```

Usuario 2: luis

Cambios realizados en las clases:

**-NCRoomManager.java:** Añadir los metodos abstractos *notifyEnterMessage(String u)* y *notifyExitMessage(String u)* para que todas sus subclases los tengan que implementar.

**-NCRoom.java:** Implementar los metodos indicados anteriormente, la implementacion es prácticamente igual a la de la función *broadcastMessage()*, simplemente se basa en coger la lista de usuarios de la sala y para cada uno enviarle un mensaje con el codigo de operacion *OP\_NOTIFY\_ENTER* o *OP\_NOTIFY\_EXIT* respectivamente y el nombre del usuario que está entrando o saliendo.

**-NCServerThread.java:** En el método *run()*, cuando recibimos un mensaje con el código *OP\_ENTER\_ROOM*, llamamos al método *notifyEntry(user)*. De igual forma en la función *processRoomMessages()* si recibimos un mensaje con el código *OP\_EXIT\_ROOM*, llamamos al método *notifyExit(user)*.

Dichos métodos, declarados en esta misma clase se encargan de pedirle al *roomManager* correspondiente que ejecute los métodos de su clase *notifyEnterMessage(u)* y *notifyExitMessage(u)* respectivamente, explicados anteriormente.

**-NCController.java:** En la función *processIncommingMessage()* debemos analizar el mensaje que nos llega, en caso de que tenga el código *OP\_NOTIFY\_ENTER* o *OP\_NOTIFY\_EXIT*, lo que haremos será mostrar la información correspondiente por pantalla, extrayendo de mensaje, el nick del usuario que está realizando la acción.

### 3.2.2.- Posibilidad de crear nuevas salas en el servidor.

Por defecto, cuando se inicia el servidor la única sala que existe es 'room\_a', lo interesante es que cada usuario pueda crear las que quiera para poder entrar con los usuarios que lo deseen a chatear de forma más 'privada'.

Cuando un usuario está ya registrado y teclea el comando *create <room>*, en el caso de que la sala 'room' no exista el servidor creará una nueva sala con dicho nombre e informará al usuario que se ha realizado con éxito, en el caso de que ya exista una sala con ese nombre, lo que hará será informar al usuario que no se ha podido crear.

```
NanoChat shell
For help, type 'help'
* Connecting to the directory...
* Connected to /127.0.0.1:6969
(nanoChat) nick luis
* Your nickname is now luis
(nanoChat) roomlist
Room Name: room_a      Members (0) : Last message: not yet
(nanoChat) roomlist
Room Name: room_a      Members (0) : Last message: not yet
Room Name: room_b      Members (0) : Last message: not yet
(nanoChat)
```

Usuario 1: luis

```
NanoChat shell
For help, type 'help'
* Connecting to the directory...
* Connected to /127.0.0.1:6969
(nanoChat) nick pedro
* Your nickname is now pedro
(nanoChat) create room_b
* You have created the room 'room_b'
(nanoChat) create room_b
* It wasn't possible create the room 'room_b'
(nanoChat)
```

Usuario 2: pedro

Como se puede observar inicialmente llega el usuario *luis* y hace un listado de las salas y como hemos indicado inicialmente solo existe la sala 'room\_a', posteriormente llega el usuario *pedro* y introduce el comando *create 'room\_b'*, creando así la sala 'room\_b'. Después *luis* vuelve a solicitar un listado con las salas disponibles y podemos observar cómo ahora aparece la nueva sala 'room\_b' creada previamente por el usuario *pedro*. Por último, *pedro* vuelve a intentar crear la sala 'room\_b', pero esta vez como ya existe en el servidor este le indica un error.

Cambios realizados en las clases:

**-NCMessage.java:** Añadir los nuevos opcodes y añadir los nuevos case en la función *readMessageFromSocket()*.

**-NCCommands.java:** Añadir el nuevo comando *COM\_HISTORY* y añadir su correspondiente entrada en *\_valid\_user\_commands*, *\_valid\_user\_commands\_str* y *\_valid\_user\_commands\_help*.

**-NCShell.java:** Añadir al case de *readGeneralCommandFromStdIn()*, el comando *COM\_HISTORY*. También en el case de *validateCommandArguments()* para obligar a que cuando se use este comando se le tenga que pasar un argumento obligatoriamente.

**-NCServerManager.java:** Adaptar la función *registerRoomManager()*, para que antes de crear una sala compruebe si ya existe, en ese caso devolverá *false*, en caso de que no exista la creará introduciéndola al mapa de salas y devolverá *true*.

**-NCServerThread.java:** Añadir el case para el opcode *OP\_CREATE\_ROOM* en la función *run()* y en ese caso, solicitarle al server manager que cree la sala mediante el método

*registerRoomManager()* explicado anteriormente, construir el mensaje de tipo *NCOperationCodeMessage* y enviarlo al cliente por el socket como respuesta a esta solicitud, indicando si el proceso se ha realizado con éxito o no.

**-NCConector.java:** Crear la función *createRoom()* la cual se encarga de crear y enviar al servidor el mensaje *NCRoomMessage* para solicitarle que cree una sala que tenga como nombre el valor del campo '*name*' del mensaje.

Esta misma función recibirá la respuesta del servidor, en caso de que el opcode sea *OP\_CREATE\_ROOM\_OK* devolverá *true*, en cualquier otro caso devolverá *false*.

**-NCController.java:** Añadir al *case* de la función *processCommand()* el comando *COM\_CREATE*, en caso de que nos llegue llamaremos a la función *createRoom()* que hemos creado en esta misma clase, la cual se encarga de solicitarle al *NCConector* que envíe al servidor la petición para crear la sala, llamando a la función explicada anteriormente *createRoom()*. En función de lo que este le devuelva mostrará por pantalla al usuario el mensaje de si se ha realizado el registro correctamente o no.

### 3.2.3.- Ver el histórico de mensajes enviados a la sala.

La idea de esta mejora consiste en que para cada sala existente del servidor mantener una lista con todos los mensajes que han sido enviados en la misma por los distintos usuarios que han entrado, así cualquier usuario ya sea que llegue nuevo a la sala, o ya perteneciente a la misma, con el mando '*history*' puede consultar y ver todo el historial de mensajes de forma ordenada y con la fecha y hora de las mismas.

```
NanoChat shell
For help, type 'help'
* Connecting to the directory...
* Connected to /127.0.0.1:6969
(nanoChat) nick juan
* Your nickname is now juan
(nanoChat) enter RoomB
* You have joined the room 'RoomB'
(nanoChat-room) history
* HISTORY:
* User 'pedro' joined the room
* User 'luis' joined the room
[Fri Apr 02 21:58:36 CEST 2021] luis: Hola!
[Fri Apr 02 21:58:51 CEST 2021] pedro: Buenos días Luis!
[Fri Apr 02 21:59:11 CEST 2021] pedro: Me voy ya, hasta luego.
* User 'pedro' left the room
[Fri Apr 02 21:59:23 CEST 2021] luis: Adios, luis
* User 'juan' joined the room
(nanoChat-room)
```

La implementación de esta mejora consiste en cuando un usuario introduce el comando '*history*' emite una petición al servidor, mediante un mensaje *NCOperationCodeMessage* usando como *opcode=OP\_GET\_HISTORY*, entonces el servidor cuando reciba este mensaje, conseguirá el listado de mensajes que se han enviado desde que la sala se creó, y posteriormente creará un mensaje de tipo *NCHistoryMessage*, el cual contendrá todos estos mensajes, una vez llegue la respuesta al cliente este lo interpretará, y se lo mostrará al usuario por pantalla.

Cambios realizados en las clases:

**-NCHistoryMessage.java:** Crear e implementar toda la clase y sus funciones para poder manejar este tipo de mensajes.

**-NCMessage.java:** Añadir los nuevos opcodes, añadir los nuevos *case* en la función *readMessageFromSocket()* y crear la nueva función *makeHistoryMessage()*.



**-NCRoomManager.java:** Añadir el atributo `LinkedList<String> history` para almacenar todos los mensajes de la sala. También hay que añadir los métodos `addMessageHistory(String message)` y `getMessageHistory()`, los cuales nos servirán para añadir un nuevo mensaje a la lista y para que nos devuelvan una copia de la misma, respectivamente.

**-NCRoom.java:** Modificar las funciones `broadcastMessage()`, `notifyEnterMessage()` y `notifyExitMessage()` añadiéndoles una llamada a la función `addMessageHistory()` pasándole como parámetro el mensaje que queremos añadir a la lista.

**-NCCommands.java:** Añadir el nuevo comando `COM_HISTORY` y añadir su correspondiente entrada en `_valid_user_commands`, `_valid_user_commands_str` y `_valid_user_commands_help`.

**-NCShell.java:** Añadir al case de `readChatCommandFromStdIn()`, el comando `COM_HISTORY`, para obligar a que este comando solo se pueda usar cuando estás dentro de una sala.

**-NCServerManager.java:** Añadir el método `getMessageHistory(String room)`, el cual se encarga de obtener el objeto de la sala que se le pasa como parámetro y llamar al método `getMessageHistory()` de la clase `NCRoomManager` obteniendo así la lista de mensajes de la sala.

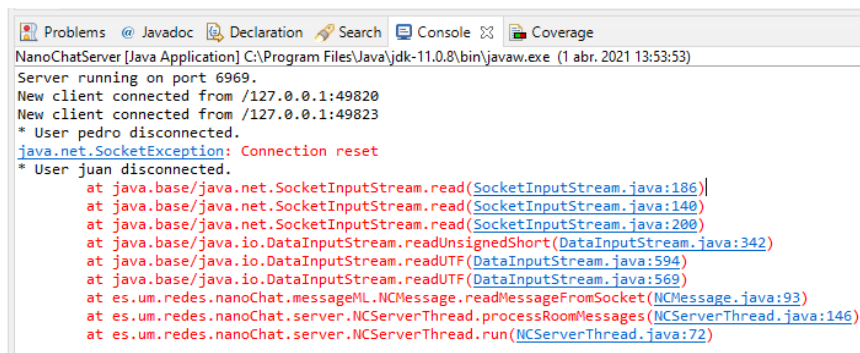
**-NCServerThread.java:** Añadir el case para el opcode `OP_GET_HISTORY` en la función `processRoomMessages()` y en ese caso, solicitarle al server manager que nos devuelva una lista con el historial de mensajes de la sala mediante el método `getMessageHistory()` explicado anteriormente, construir el mensaje de tipo `NCHistoryMessage` y enviarlo al cliente por el socket.

**-NCConnector.java:** Crear la función `getHistory()` la cual se encarga de crear y enviar al servidor el mensaje `NCOperationCodeMessage` para solicitarle al servidor el historial. Esta misma función recibirá el mensaje de tipo `NCHistoryMessage` que le habrá enviado el servidor, y devuelve una lista con los mensajes.

**-NCController.java:** Añadir al case de la función `processRoomCommand()` el comando `COM_HISTORY`, en caso de que nos llegue llamaremos a la función `getAndShowHistory()` que hemos creado en esta misma clase, la cual se encarga de solicitarle al `NCConnector` la lista de mensajes, llamando a la función explicada anteriormente `getHistory()`. Una vez tiene la lista con los mensajes se encarga de mostrarlos por pantalla.

### 3.2.4.- Capturar la excepción que se produce en el servidor de chat cuando un usuario aborta su conexión y tratarla adecuadamente.

Tal y como estaba programada inicialmente la práctica ocurría que si un cliente entraba a una sala de chat y estando dentro forzaba su salida, es decir, el cliente cerraba el programa, esto hacía que se produjese una excepción en la función `processRoomMessages()`, de la clase `ServerThread`, y como no se trataba por defecto el servidor dejaba de funcionar mostrando el siguiente error:



```

Problems  @ Javadoc  Declaration  Search  Console  Coverage
NanoChatServer [Java Application] C:\Program Files\Java\jdk-11.0.8\bin\javaw.exe (1 abr. 2021 13:53:53)
Server running on port 6969.
New client connected from /127.0.0.1:49820
New client connected from /127.0.0.1:49823
* User pedro disconnected.
java.net.SocketException: Connection reset
* User juan disconnected.
    at java.base/java.net.SocketInputStream.read(SocketInputStream.java:186)
    at java.base/java.net.SocketInputStream.read(SocketInputStream.java:140)
    at java.base/java.net.SocketInputStream.read(SocketInputStream.java:200)
    at java.base/java.io.DataInputStream.readUnsignedShort(DataInputStream.java:342)
    at java.base/java.io.DataInputStream.readUTF(DataInputStream.java:594)
    at java.base/java.io.DataInputStream.readUTF(DataInputStream.java:569)
    at es.um.redes.nanoChat.messageML.NCHistoryMessage.readMessageFromSocket(NCHistoryMessage.java:93)
    at es.um.redes.nanoChat.server.NCServerThread.processRoomMessages(NCServerThread.java:146)
    at es.um.redes.nanoChat.server.NCServerThread.run(NCServerThread.java:72)
  
```



Como esto no es demasiado correcto, puesto que porque un cliente aborte su conexión, no debería ocurrir que deje de funcionar un servidor de chat completo con el resto de sus clientes, he realizado las siguientes modificaciones para solucionarlo:

Cambios realizados en las clases:

**-NCServerThead.java:** En la función *processRoomMessages()* eliminar el *try/catch* que se utilizaba para tratar las excepciones que pudiesen producir sus instrucciones, y añadir en su declaración *'throws IOException'* de forma que si se produce alguna, de estas, será controlada por el manejador de la función que esté llamando a *processRoomMessages()* que en este caso será el método *run()* de cada hilo, y ya ahí el manejador se encargará de notificar que el usuario se ha desconectado, también se eliminará de la sala en la que se encontraba, eliminará al usuario del sistema y cerrará, si no lo está ya el socket que tenía para el.

De esta forma, desde el punto de vista del servidor, tanto si el cliente sale de forma exitosa (comandos *exit* y *quit*) como si no (el cliente aborta la conexión) seguirá funcionando correctamente y sin problemas.

### **3.2.5.- Modificar la clase *Directory* y *DirectoryThead* para que acepte como parámetro la IP de la interfaz donde va a estar corriendo ese servicio.**

Resulta interesante poder probar y curiosear como la práctica que hemos programado puede servir realmente para comunicar a dos clientes que se encuentran en dos equipos distintos, incluso además el servidor y el directorio ni siquiera tendrían porque estar en la misma máquina.

En mi caso la prueba que he realizado es con tres ordenadores distintos, uno para tener corriendo los servicios del Directorio y del Servidor de Chat y otros dos equipos distintos en los que se ejecuta el Cliente de chat, y efectivamente podemos comprobar que ambos clientes se pueden comunicar entre ellos, no directamente, sino a través del equipo que actúa en este caso de Servidor de Chat y de Directorio.

Para poder realizar esto, además de exportar el proyecto (las clases *Directory*, *NanoChatServer* y *NanoChat*) como un archivo *'jar'* hemos tenido que instalar en los tres equipos java, para poder ejecutarlos.

Además, hemos tenido que realizar modificaciones en las clases *Directory* y *DirectoryThead* para que ahora la clase *Directory* tenga como parámetros obligatorios *-loss probability* y *-ip ip\_pc\_directory*. Este cambio es necesario, porque inicialmente el Directorio, cuando se creaba cogía la dirección por defecto (*localhost (127.0.0.1)*), y ahora la que va a coger es la que le pasemos como parámetro, así mientras sigamos desarrollando la práctica le pasaremos como parámetro *localhost*, para que siga funcionando todo en nuestra máquina, pero en el caso de querer realizar la prueba anteriormente comentada, le pasaremos la dirección ip, en mi caso privada (de mi subred (*192.169.0.109*)) que identifica el ordenador donde voy a poner a correr el directorio, esta ip deberá ser conocida por el servidor de chat y todos los clientes para que se puedan conectar al directorio.

```

pedro@pedro-PC: ~/Escritorio/redes
pedro@pedro-PC:~/Escritorio/redes$ java -jar Directory.jar -loss 0.0 -ip 192.168.0.109
Probability of corruption for received datagrams: 0.0
IP Directory: 192.168.0.109  PORT Directory: 6868
Directory starting...

pedro@pedro-PC: ~/Escritorio/redes
pedro@pedro-PC:~/Escritorio/redes$ java -jar NanoChatServer.jar 192.168.0.109
Server running on port 6969.
New client connected from /192.168.0.100:51661
New client connected from /192.168.0.113:63348
* User Cesar disconnected.
* User Maria disconnected.

```

*Equipo 1: Tiene el directorio y servidor de chat corriendo.*

```

C:\Users\Maria\Desktop>java -jar NanoChat.jar 192.168.0.109
NanoChat shell
For help, type 'help'
* Connecting to the directory...
* Connected to /192.168.0.109:6969
(nanoChat) nick Maria
* Your nickname is now Maria
(nanoChat) enter RoomB
* You have joined the room 'RoomB'
(nanoChat-room) * Message received from server...
* User 'Cesar' joined the room
(nanoChat-room) * Message received from server...
[Fri Apr 02 23:03:16 CEST 2021] Cesar: Hola
(nanoChat-room) send Hola!
(nanoChat-room) * Message received from server...
[Fri Apr 02 23:03:33 CEST 2021] Cesar: Adios
(nanoChat-room) send Hasta luego!
(nanoChat-room) exit
* Your are out of the room
(nanoChat) quit
Bye.

```

*Equipo 2: Cliente de chat.*

```

C:\Users\Cesar\Desktop>java -jar NanoChat.jar 192.168.0.109
NanoChat shell
For help, type 'help'
* Connecting to the directory...
* Connected to /192.168.0.109:6969
(nanoChat) nick Cesar
* Your nickname is now Cesar
(nanoChat) roomlist
Room Name: RoomB      Members (1) : Maria   Last message: not yet
Room Name: RoomA      Members (0) :         Last message: not yet
(nanoChat) enter RoomB
* You have joined the room 'RoomB'
(nanoChat-room) send Hola
(nanoChat-room) * Message received from server...
[Fri Apr 02 23:03:25 CEST 2021] Maria: Hola!
(nanoChat-room) send Adios
(nanoChat-room) * Message received from server...
[Fri Apr 02 23:03:51 CEST 2021] Maria: Hasta luego!
(nanoChat-room) * Message received from server...
* User 'Maria' left the room
(nanoChat-room) exit
* Your are out of the room
(nanoChat) quit
Bye.

```

*Equipo 3: Cliente de chat.*

Como podemos observar en los mensajes que nos reporta el servidor de chat que está corriendo en la dirección *192.169.0.109*, efectivamente se están realizando conexiones cliente desde otros equipos distintos, en este caso identificados por las direcciones *192.169.0.100* y *192.169.0.113*.

Cambios realizados en las clases:

**-DirectoryThead.java:** Modificar el constructor para que ahora también se le pase un *String directoryIP*, donde esa cadena contendrá la dirección IP del equipo donde va a estar corriendo el directorio. Esta cadena la usaremos para pasarla como parámetro al constructor de la clase *'InetSocketAddress(directoryIP, directoryPort)'*.

**-Directory.java:** Modificar la parte de tratamiento de los argumentos, antes si no se indicaba el parámetro *-loss*, tomaba por defecto la probabilidad a *0.0*, pero ahora por simplicidad en la comprobación de argumentos, vamos a obligar a indicar este parámetro, aunque la probabilidad que se le pase sea *0.0* y además vamos a obligar que se pase el parámetro *-ip*, cuyo valor vamos a leer en la variable *DIRECTORY\_IP*, y que posteriormente utilizaremos para pasarsela al constructor de la clase *DirectoryThead*.

#### **4.-CONCLUSIÓN.**

Mi opinión con respecto a la práctica es que inicialmente se hace bastante complicado familiarizarse con el código y entender todas las clases y métodos que lo componen, esto hace que el comienzo sea bastante costoso. En cambio cuando ya has implementado un par de comandos, te has enfrentado a todos los problemas que te suelen aparecer, de cosas que no funcionan, excepciones que no sabes de donde saltan y donde está el error, etc, a partir de ese momento, ya conoces al dedillo el proyecto, sabes como es la comunicación entre las distintas clases del proyecto, entiendes perfectamente cómo está estructurado y que parte hace cada cosa, por lo que a partir de ahí ya se vuelve todo mucho más sencillo y resulta muy satisfactorio continuar con la práctica, esto hace que te enganches y en mi caso que no me costase dedicarle demasiado tiempo el mismo día o la misma semana y que en mi caso que cuando terminé la parte obligatoria continuase implementando mejoras.