MEMORIA PROYECTO NANO CHAT

AÑO ACADÉMICO: 2020-2021

ASIGNATURA: Redes

ALUMNOS: David Fernández Expósito y

Pablo Tadeo Romero Orlowska

DNI: 49444688R (David) y 48665752Y (Pablo)

GRUPO: PCEO

ÍNDICE DE CONTENIDOS

1. INTRODUCCIÓN

En este documento memoria del proyecto NanoChat se especifica el diseño de los protocolos de las comunicaciones entre las distintas partes del proyecto de NanoChat (servidor de directorio, servidor de chat, cliente de chat). Se mostrará el diseño de los mensajes, que serán mensajes binarios multiformato en el caso de la comunicación con el servidor de directorio y de tipo lenguaje de marcas para la comunicación entre el servidor de chat y el cliente de chat. También se mostrará el diseño de los autómatas que modelan el comportamiento del servidor de chat, el servidor de directorio y el cliente de chat. Además, también se tratará la implementación de los formatos de los mensajes, la implementación del mecanismo de gestión de salas y las diferentes mejoras que se han implementado.

2. FORMATO DE LOS MENSAJES DEL PROTOCOLO DE COMUNICACIÓN CON EL DIRECTORIO

Para definir el protocolo de comunicación con el Directorio, vamos a utilizar mensajes binarios multiformato. El valor que tome el campo “opcode” (código de operación) me indicará el tipo de mensaje y por tanto cuál es su formato, es decir, qué campos vienen a continuación.

Formatos de mensajes:

Formato: Control.

Opcode (1 byte)

Formato: OneParameter.

Opcode (1 byte)	Parámetro (1 byte)

Formato: TwoParameter.

Opcode (1 byte)	Parámetro1 (1 byte)	Parámetro2(4 bytes)

Formato: FiveParameter.

Opcode (1 byte)	Parámetro1 (1 byte)	Parámetro2 (1 byte)	Parámetro3 (1 byte)	Parámetro4 (1 byte)	Parámetro5 (4 bytes)

Tipo y descripción de los mensajes:

Mensaje: **registration (opcode = 1).**

Formato: TwoParameter.

Sentido de la comunicación: Servidor de chat → Servidor de directorio.

Descripción: Mensaje enviado por el servidor de chat al directorio para registrarse como servidor de chat para el protocolo indicado en el campo Protocol_ID (1 byte) en el puerto indicado en el campo Puerto (4 bytes). El valor asignado al opcode es 1.

Ejemplo:

Opcode (1 byte)	Protocol_ID (1 byte)	Puerto (4 bytes)
1	104	6969

Mensaje: **registrationOk (opcode = 2).**

Formato: Control.

Sentido de la comunicación: Directorio → Servidor de chat.

Descripción: Este mensaje lo envía el Directorio a un servidor de chat para confirmarle que su solicitud de registro se ha realizado correctamente. El valor asignado al opcode es 2.

Ejemplo:

Opcode (1 byte)
2

Mensaje: **query (opcode = 3).**

Formato: OneParameter.

Sentido de la comunicación: Cliente de chat → Directorio

Descripción: Este mensaje lo envía el Cliente de Chat al Directorio para consultar si existe algún Servidor de Chat registrado para el protocolo dado en el campo Protocol_ID, que es un entero (1 byte). El valor asignado al opcode es 3.

Ejemplo:

Opcode (1 byte)	Protocol_ID (1 byte)
3	104

Mensaje: **info (opcode = 4).**

Formato: FiveParameter.

Sentido de la comunicación: Directorio → Cliente de chat.

Descripción: Este mensaje lo envía el directorio al cliente de chat como respuesta satisfactoria a su consulta de si hay algún Servidor de chat para un protocolo determinado. El cliente recibe la IP del servidor de chat dividida en 4 campos de 1 byte (uno por cada número que forma la IP) y el puerto en otro campo de 4 bytes. El valor asignado para opcode es 4.

Ejemplo:

Opcode (1 byte)	IP_1 (1 byte)	IP_2 (1 byte)	IP_3 (1 byte)	IP_4 (1 byte)	Puerto (4 bytes)
4	127	0	0	1	6969

Mensaje: **no_info (opcode = 5).**

Formato: Control.

Sentido de la comunicación: Directorio → Cliente de chat.

Descripción: Este mensaje es un mensaje de respuesta no satisfactoria enviada del directorio al cliente tras una consulta de este último por un servidor de chat para un protocolo para el que no existe ningún servidor de chat registrado aún. El valor asignado para opcode es 5.

Ejemplo:

Opcode (1 byte)
5

3. FORMATO DE LOS MENSAJES DEL PROTOCOLO DE COMUNICACIÓN ENTRE CLIENTE DE CHAT Y SERVIDOR DE CHAT

Nuestro identificador de protocolo, calculado como la suma de nuestros DNI, y haciendo modulo 128 a esta suma, es 104, por lo que nos corresponde usar el formato de mensajes basado en Lenguaje de marcas.

Formatos de mensajes:

Formato: **RoomMessage**

```
<message>
    <operation>xxxx</operation>
    <name>xxxx</name>
</message>
```

NOTA: Este formato incluye los mensajes con dos campos, uno "operation" y otro con texto, ya sea nombre, nombre de sala, texto…

Formato: **ControlMessage**

<message>

<operation>xxxx</operation>

</message>

Formato: **ListMessage**

<message>

<operation>xxxx</operation>

<list>

<room><name>sala1</name><time> last_message_time</time><items><item> Juan</item>…</items></room>

…

</list>

</message>

Formato: ThreeParameterMessage

<message>

<operation>xxxx</operation>

<nick>xxxx</nick>

<text>xxxx</text>

</message>

Tipo y descripción de los mensajes:

Mensaje: **Nick.**

Formato: RoomMessage.

Sentido de la comunicación: Cliente de chat → Servidor de chat.

Descripción: Este mensaje lo envía el Cliente de chat al servidor de chat para registrarse con un *Nick* cuyo valor viene dado en el campo *name*.

Ejemplo:

```
<message>
    <operation>Nick</operation>
    <name>Pablo</name>
</message>
```

Mensaje: **Nick_Ok**

Formato: ControlMessage

Sentido de la comunicación: Servidor de chat → Cliente de chat.

Descripción: Mensaje de respuesta del servidor de chat al cliente para confirmarle que se ha registrado correctamente con el nick que ha solicitado.

Ejemplo:

```
<message>
    <operation>Nick_Ok</operation>
</message>
```

Mensaje: **Nick_Duplicated**

Formato: ControlMessage

Sentido de la comunicación: Servidor de chat → Cliente de chat.

Descripción: Este mensaje lo envía el Servidor de Chat al Cliente de Chat en respuesta a un mensaje de "registerNick" para indicarle que no se ha podido realizar el registro porque el nick dado ya estaba registrado por otro cliente de chat.

Ejemplo:

<message>

<operation>Nick_Duplicated</operation>

</message>

Mensaje: **Room_List_Query**

Formato: ControlMessage

Sentido de la comunicación: Cliente de chat → Servidor de chat.

Descripción: Mensaje que envía el cliente al servidor de chat para solicitar la lista de salas del servidor.

Ejemplo:

<message>

<operation>Room_List_Query</operation>

</message>

Mensaje: **Room_List**

Formato: ListMessage

Sentido de la comunicación: Servidor de chat → Cliente de chat.

Descripción: Mensaje que envía el servidor de chat al cliente, tras una petición de este último, con la lista de las salas del servidor.

Ejemplo:

<message>

<operation>Room_List</operation>

<list>

<room><name>sala1</name><time> last_message_time</time><items><item> Juan</item>…</items></room>

…

</list>

</message>

Mensaje: **Enter_Room**

Formato: RoomMessage

Sentido de la comunicación: Cliente de chat → Servidor de chat

Descripción: Mensaje que envía el cliente de chat al servidor para pedir entrar en una sala de este.

Ejemplo:

```
<message>
    <operation>Enter_Room</operation>
    <name>sala1</name>
</message>
```

Mensaje: **Enter_Room_Ok**

Formato: ControlMessage

Sentido de la comunicación: Servidor de chat → Cliente de chat.

Descripción: Mensaje que envía el servidor de chat al cliente para confirmarle su entrada en una sala de este.

Ejemplo:

```
<message>
    <operation>enterRoomOk</operation>
</message>
```

Mensaje: **Enter_Room_Fail**

Formato: ControlMessage

Sentido de la comunicación: Servidor de chat → Cliente de chat

Descripción: Mensaje que envía el servidor de chat al cliente para informarle de que no ha podido entrar a la sala solicitada.

Ejemplo:

```
<message>
    <operation>enterRoomFail</operation>
</message>
```

Mensaje: **Exit**

Formato: ControlMessage

Sentido de la comunicación: Cliente de chat → Servidor de chat

Descripción: Mensaje que envía el cliente de chat al servidor para pedir salir de la sala en la que se encuentra.

Ejemplo:

```
<message>
    <operation>Exit</operation>
</message>
```

Mensaje: **Send**

Formato: RoomMessage

Sentido de la comunicación: Cliente de chat → Servidor de chat

Descripción: Mensaje que envía el cliente de chat al servidor para enviar un texto por la sala.

Ejemplo:

```
<message>
    <operation>Send</operation>
    <name>texto…</name>
</message>
```

Mensaje: **Send_In**

Formato: ThreeParametersMessage

Sentido de la comunicación: Servidor de chat → Cliente de chat

Descripción: Mensaje que envía el servidor de chat al cliente con el texto enviado por alguien en la sala.

Ejemplo:

```
<message>
    <operation>Send_In</operation>
    <nick>Pepe</nick>
    <text>texto…</text>
</message>
```

Mensaje: **Info_Query**

Formato: ControlMessage

Sentido de la comunicación: Cliente de chat → Servidor de chat

Descripción: Mensaje que envía el cliente de chat al servidor para pedir información relacionada con la sala en la que se encuentra.

Ejemplo:

```
<message>
    <operation>Info_Query</operation>
</message>
```

Mensaje: **Info**

Formato: listMessage

Sentido de la comunicación: Servidor de chat → Cliente de chat

Descripción: Mensaje enviado por el servidor al cliente con la descripción de la sala en la que se encuentra.

Ejemplo:

```
<message>
    <operation>Info</operation>
    <list>
        <room><name>sala1</name><items> Pepe</items><time>18.04.2021-18:10</time></room>
    </list>
</message>
```

NOTA: Nótese que se reutiliza el formato listMessage para info, porque al final se está mandando la misma información, pero solo de una sala. Por eso se manda una lista con un solo campo.

NOTA: nótese que help y quit no aparecen, pues no son mensajes, tal y como se aclaró en clase de prácticas.

Mensaje: **User_In**

Formato: RoomMessage

Sentido de la comunicación: Servidor de chat → Cliente de chat

Descripción: Mensaje que envía el servidor de chat al cliente para informar de la entrada de otro usuario a la sala.

Ejemplo:

```
<message>
    <operation>User_In</operation>
    <name>Pepe</name>
</message>
```

Mensaje: **User_Out**

Formato: RoomMessage

Sentido de la comunicación: Servidor de chat → Cliente de chat

Descripción: Mensaje que envía el servidor de chat al cliente para informar de que un usuario se salió de la sala.

Ejemplo:

```
<message>
    <operation>User_Out</operation>
    <name>Pepe</name>
</message>
```

Mensaje: **Room_Rename**

Formato: RoomMessage

Sentido de la comunicación: Cliente de chat → Servidor de chat

Descripción: Mensaje que envía el cliente de chat al servidor para renombrar una sala.

Ejemplo:

```
<message>
    <operation>Room_Rename</operation>
    <name>nombre_sala</name>
</message>
```

Mensaje: **Room_Rename_Ok**

Formato: ControlMessage

Sentido de la comunicación: Servidor de chat → Cliente de chat

Descripción: Mensaje que envía el servidor de chat al cliente para confirmar que ha renombrado la sala.

Ejemplo:

```
<message>
    <operation>Room_Rename_Ok</operation>
</message>
```

Mensaje: **Room_Rename_Fail**

Formato: ControlMessage

Sentido de la comunicación: Servidor de chat → Cliente de chat

Descripción: Mensaje que envía el servidor de chat al cliente para informar de que no ha podido cambiar el nombre de la sala como deseaba.

Ejemplo:

```
<message>
    <operation>Room_Rename_Fail</operation>
</message>
```

Mensaje: **Create_Room**

Formato: ControlMessage

Sentido de la comunicación: Cliente de chat → Servidor de chat

Descripción: Mensaje que envía el cliente de chat al servidor para crear una sala con el siguiente nombre que le corresponda.

Ejemplo:

```
<message>
    <operation>Create_Room</operation>
</message>
```

Mensaje: **Send_Private**

Formato: ThreeParameterMessage

Sentido de la comunicación: Cliente de chat → Servidor de chat

Descripción: Mensaje que envía el cliente de chat al servidor para enviar un mensaje privado al nick que indique.

Ejemplo:

```
<message>
    <operation>Send_Private</operation>
    <nick>Pepe</nick>
    <text>texto…</text>
</message>
```

Mensaje: **Send_Private_In**

Formato: ThreeParameterMessage

Sentido de la comunicación: Servidor de chat → Cliente de chat

Descripción: Mensaje que envía el servidor de chat al cliente para enviar un mensaje privado a su destinatario. En este caso el Nick es el remitente del mensaje.

Ejemplo:

```
<message>
    <operation>Send_Private_In</operation>
    <nick>Pepe</nick>
    <text>texto…</text>
</message>
```

Mensaje: **Send_Private_Fail**

Formato: ControlMessage

Sentido de la comunicación: Servidor de chat → Cliente de chat

Descripción: Mensaje que envía el servidor de chat al cliente para informarle de que no se pudo enviar su mensaje privado.

Ejemplo:

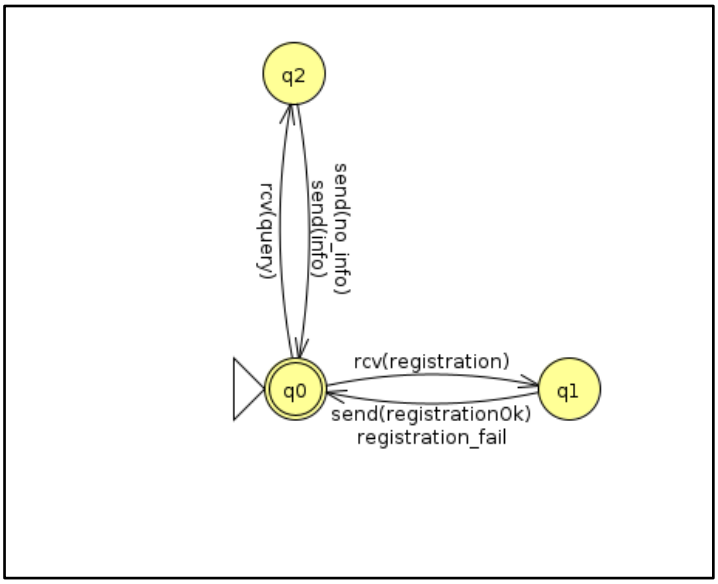
<message>

<operation>Send_Private_Fail</operation>

</message>

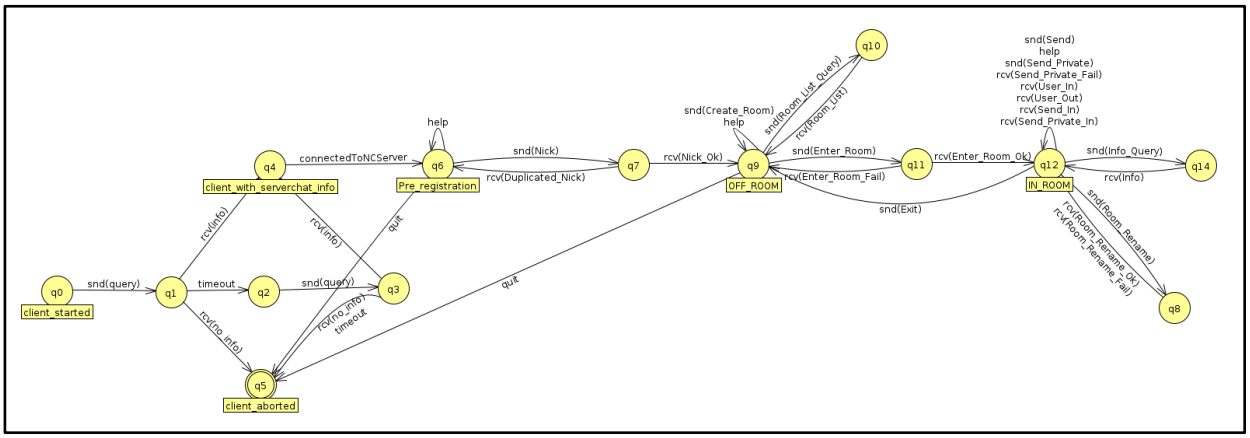
4. AUTÓMATAS DE PROTOCOLO

Autómata para el servidor de directorio:



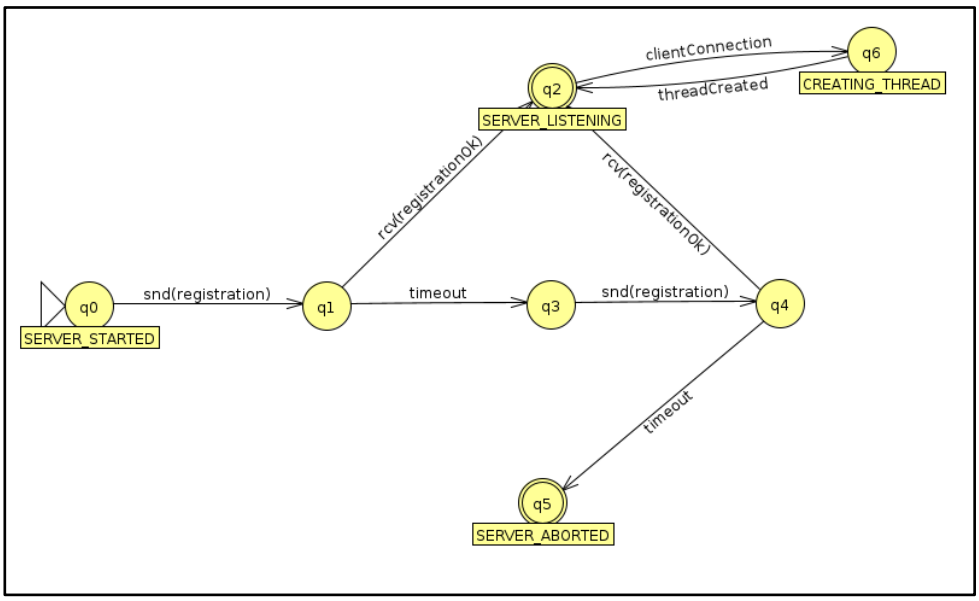
Nótese que puede fallar el registro de un servidor de chat pues, por ejemplo, puede ya haber uno registrado para su protocolo.

Autómata para el cliente de chat:

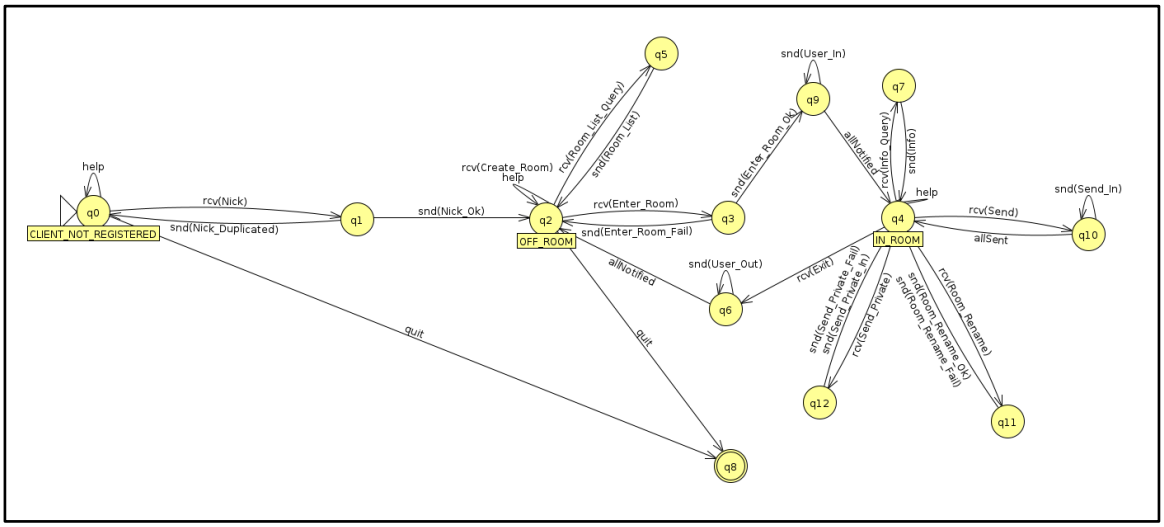


Para el autómata del servidor de chat, se ha optado por separarlo en dos autómatas: el primero se encarga de la parte de UDP (registrarse en el servidor de directorio) y dejar el servidor en el estado final, en el que deja el hilo en segundo plano para escuchar por el puerto TCP. El segundo se encarga del NCServerThread (parte TCP), es decir, la comunicación con el cliente de chat.

Autómata para el servidor de chat (parte UDP):



Autómata para el NCServerThread (parte TCP del servidor de chat):



5. EJEMPLO DE INTERCAMBIO DE MENSAJES

Servidor de chat: registration(protocol = 104; puerto = 6969)

Servidor de directorio: registrationOk

Cliente de chat: query(protocol = 104)

Servidor de directorio: queryOk(IP = 127.0.0.1; puerto = 6969)

Cliente de chat: ConnectionToChatServer.

En este estado el cliente solo puede enviar un nick para registrarse.

Cliente de chat: registerNick(nick= Pepe)

Servidor de chat: nickOk

Como el Nick ha sido aceptado, ahora puede pedir la lista de salas o intentar entrar en una sala.

Cliente de chat: roomListQuery

Servidor de chat: roomList(sala1, sala2, …)

Cliente de chat: enterRoom(sala2)

Servidor de chat: enterRoomOk

Ahora el cliente está en el estado IN_ROOM, puede pedir ayuda, enviar mensajes, pedir la información de la sala, salirse de la sala o desconectarse del autómata.

Cliente de chat: send(“hello world”)

Cliente de chat: help

Cliente de chat: exit

Cliente de chat: quit (no es un mensaje)

6. DETALLES SOBRE LOS PRINCIPALES ASPECTOS DE LA IMPLEMENTACIÓN

6.1 IMPLEMENTACIÓN DEL FORMATO DE LOS MENSAJES

En cuanto a los mensajes binarios, todo lo relacionado con ellos se hace directamente en las clases DirectoryConector o DirectoryThread, no se implementan clases para cada tipo de mensajes como si se hace con los mensajes entre el cliente y el servidor. En los binarios simplemente, a la hora de enviar, se codifica adecuadamente la información en el mensaje que se va a enviar y a la hora de recibir se obtienen correspondientemente.

En el caso de la comunicación TCP, se crean clases descendientes de la clase NCMessage, una para cada tipo de mensaje. En nuestro caso, nos tocó usar lenguaje de marcas. Cada clase tendrá unos atributos u otros, dependiendo del tipo. Sin embargo, todas comparten el atributo opCode que heredan de NCMessage y el hecho de que implementan los métodos readFromString (dada una cadena la convierte en un objeto del mensaje correspondiente, va separando la información y "metiéndola" en los atributos del objeto) y toEncodedString (crea una cadena con el mensaje a partir del objeto, haciendo uso de las características del lenguaje de marcas). Como usamos lenguaje de marcas, tuvimos que hacer uso de *Pattern* y *Matcher* para implementar el método *readFromString.*

Como se observa en los apartados anteriores de la presente memoria, hemos definido 4 tipos de mensaje, cada uno con su clase descendiente de NCMessage: NCControlMessage, NCRoomMessage, NCListMessage y NCThreeParameterMessage. Los mensajes en los que se usa cada tipo ya se han comentado en apartados anteriores.

NCControlMessage es la clase más sencilla, pues no incluye ningún parámetro adicional, solo usa el opCode. Además, no es necesario que se implemente el método readFromString que sí implementan los demás, pues para estos mensajes solo se envía el opCode, nada más. NCRoomMessage incluye un "nombre" (una string) y el opCode. NCListMessage, como indica el nombre, incluye una lista, específicamente una lista de NCRoomDescription, que es la clase que implementa las descripciones de las salas del servidor, pues este tipo de mensaje solo se usa para los mensajes de *roomList* y *info*. Por último, NCThreeParameterMessage incluye, además del opCode, un nombre y un texto (ambos atributos son cadenas), pues este es el tipo de mensaje que se usará para hacer "broadcast" de un mensaje en una sala, por ejemplo.

6.2 MECANISMO DE GESTIÓN DE SALAS

La gestión de qué salas hay en el servidor en cada momento la lleva realmente la clase NCServerManager. Sin embargo, en la clase NCServerThread, podemos encontrar alguna variable relacionada con esto también, como roomManager (indica el manager de la sala actual, la actual del usuario relacionado con el thread del servidor) y currentRoom (almacena el nombre de la sala actual del usuario relacionado con el thread).

En NCServerManager, la principal estructura de datos usada para la gestión de salas es un mapa, llamado "rooms", que relaciona cada RoomManager con el nombre de la sala.

Cuando se inicia el servidor, se crea una primera sala "Room1". Nótese que tenemos una variable "INITIAL_ROOM" que almacena el número que se le asignará a la primera sala. También tenemos una variable "ROOM_PREFIX" que almacena el prefijo que se pondrá a cada sala creada, y un entero que se va aumentando conforme se van creando salas para que vaya avanzando el número de la sala. Cuando se registra una sala, ya sea al iniciarse o tras un "createroom" (se tratará esta mejora posteriormente), se inserta en el mapa el RoomManager con su nombre correspondiente (creado usando el prefijo y el entero). Claro está, se hace un bucle para ir avanzando de número hasta encontrar uno que no haya sido usado, pues podría ocurrir que algún usuario hubiese renombrado alguna sala de modo que se llamase "Room4" por ejemplo (posteriormente trataremos la mejora de renombrar salas implementada), y hay que evitar que se repitan los nombres. Una consideración de diseño que debemos comentar es que decidimos que, si alguna sala se queda vacía, no se borra, es decir, en nuestro programa no se pueden eliminar salas.

Nótese que al principio la implementación que se nos proporcionó usaba letras y un byte para ir creando las salas nuevas, pero como esto haría que hubiese un límite de salas, para evitar problemas (aunque nunca crearemos tantas salas en realidad), decidimos cambiar la letra por un entero, cuyo límite nunca alcanzaremos.

Por otro lado, cabe comentar que las salas son objetos de la clase NCRoom, que desciende de la clase abstracta NCRoomManager. NCRoom implementa, claro está, todos los métodos abstractos de la clase abstracta, además de incluir dos nuevos atributos: la fecha del último mensaje (realmente, es un long y almacena los milisegundos desde "la época" hasta cuando se envió el último mensaje) y un mapa para relacionar a cada usuario con su socket. La fecha se almacena para luego poder crear objetos del tipo NCRoomDescription cuando se pide información con el comando *info* o cuando se pide la lista de salas con el comando *roomList*.

Cabe mencionar que la fecha se actualiza cada vez que se envía un mensaje en la sala, pero si se envía un mensaje privado se decidió no actualizar la fecha, pues para eso es un mensaje privado. En cuanto a los métodos que nos encontramos, cabe destacar primero que en la clase abstracta se implementó un método getName() para obtener el nombre de la sala. Se decidió implementar aquí pues el atributo name se define en la clase abstracta. Por lo demás, son todo métodos abstractos que se implementan en NCRoom. Nos encontramos con métodos como: *registerUser* (que devuelve un booleano pero que en realidad nunca podrá devolver falso pues no se dará el caso de intentar entrar con un usuario que ya existe, pues esto ya se controla con los nicks en el propio servidor) y *removeUser*, que añaden y quitan usuarios de la sala; *setRoomName* para nombrar o renombrar la sala (nótese que el constructor de NCRoom no inicializa el nombre, hay que llamar a *setRoomName* siempre que se cree una sala); *broadcastMessage* y *sendMessage*, para mandar mensajes de sala y mensajes privados, respectivamente; *broadcastIONotification*, que según se le pase un 1 o un 2 notificará de salidas o entradas de otros usuarios; *usersInRoom* para obtener el número de usuarios dentro de la sala; y *getDescription*, que devuelve la descripción de la sala. Este último método es interesante porque devuelve un objeto del tipo NCRoomDescription, que es lo que usamos para las descripciones necesarias para *info* y *RoomList*. Para crear un objeto de este tipo es necesaria la lista de usuarios, el nombre de la sala y el tiempo del último mensaje (que está en milisegundos desde "la época"). Con esto, mediante el método *toPrintableString*, crea una cadena con la descripción de la sala.

En el siguiente apartado de esta memoria trataremos mejoras implementadas que afectan a las salas, como la creación de nuevas salas y la posibilidad de renombrarlas.

6.3 MEJORAS ADICIONALES IMPLEMENTADAS

Hemos podido implementar 4 mejoras: notificar a los demás usuarios de las entradas/salidas de otros usuarios en la sala (0.5), posibilidad de crear otras salas (0.5), enviar mensajes privados (0.5) y posibilidad de renombrar salas (1).

Para notificar a los demás usuarios de las entradas/salidas, lo primero fue definir dos mensajes más: User_In y User_Out, ambos de tipo ControlMessage. Cuando un usuario entra a una sala, en el serverThread, si se puede entrar a la sala (podría pasar que no se pudiese pues no existe la sala), antes de llamar al método "processRoomMessages()", se notifica a todos los demás usuarios de la sala. Para ello, se definió otro método en NCRoom llamado "broadcastIONotification", que avisa a todos de una entrada o salida del usuario que se

indique (el método se comporta de manera similar a broadcastMessage). Para la salida de un usuario se procedió igual, cuando un usuario decide salir de la sala, en el serverThread, antes de actualizar las variables correspondientes, se llama a "broadcastIONotification" para notificar a los demás de la salida. También podríamos haber metido la funcionalidad del método "broadcastIONotification" en los métodos "registerUser" y "leaveRoom", y nos hubiésemos ahorrado definir otro método, pero pensamos que separar en métodos distintos ayuda a tener más claro qué hace cada uno y a distinguir qué forma parte de la implementación de las mejoras y qué no. Claramente, también hubo que hacer cambios en NCController y NCConnector, pues deben poder recibir los mensajes correspondientes. Para ello, simplemente se modificaron los métodos "processIncomingMessage" y "receiveMessage", respectivamente, para que pudiesen recibir este tipo de mensajes además de los mensajes de texto normales.

Para poder crear nuevas salas, se creó un nuevo mensaje "Create_Room", que será enviado por el cliente cuando se encuentre fuera de sala y quiera crear una. Nótese que no se pasa ningún nombre, es un mensaje de tipo ControlMessage, pues se decidió que las nuevas salas creadas se crearían con el siguiente número disponible, y si se quiere poner otro nombre, es responsabilidad del usuario meterse en la sala y renombrarla. También hay que tener en cuenta que se consideró que no hacía falta un mensaje informando de que no se pudo crear la sala, pues si el siguiente nombre de sala ya existe, se hará un bucle hasta encontrar algún número disponible para el nombre de la sala. Como ya se comentó, se decidió cambiar las letras por números para evitar problemas de que se acaben las salas o los nombres posibles. Por parte del servidor, cuando el serverThread recibe un mensaje de creación de sala, simplemente registra en el serverManager una nueva sala, ya que este método del manager ya se encarga de ponerle a la sala un nombre adecuado.

Para renombrar salas, se tuvo que definir un nuevo mensaje "Room_Rename", del tipo RoomMessage. En el cliente, simplemente hubo que crear un método en NCController y otro en NCConnector para poder mandar al servidor el mensaje de solicitud de renombrar la sala. En el servidor, se intenta renombrar la sala llamando a un método definido en NCServerManager que intenta cambiar el nombre, pero comprobando que no exista ya ninguna sala con ese nombre. Así, en NCServerThreasd, si se ha podido renombrar, se informa con el correspondiente nuevo mensaje "Room_Rename_Ok" al cliente y, si no se ha podid, con el nuevo mensaje "Room_Rename_Fail". Por lo tanto, en el cliente, en NCConnector, se lee el mensaje enviado por el servidor y devuelve un booleano al NCController, según se haya podido renombrar o no. Y en NCController, según el caso, se

imprimirá un mensaje u otro. Un detalle crucial de la implementación de esta mejora es que el NCServerThread conectado con el cliente que decide renombrar la sala (recordar que hay un NCServerThread por cliente conectado al servidor) debe actualizar correctamente su variable "currentRoom". Sin embargo, todos los demás clientes no lo hacen, lo que puede llevar a inconsistencias a la hora de ejecutar el programa (puede ocurrir que salga de la sala y que siga apareciendo como usuario dentro de la sala o cosas así). Para solucionar esto, en NCServerThread, en el bucle del método "processRoomMessages()" se actualiza en cada iteración el nombre de la sala actual llamando al método getName() del roomManager de la sala, por si algún usuario la hubiese cambiado.

Por último, para enviar mensajes privados, se definieron nuevos mensajes "Send_Private", "Send_Private_In", ambos de tipo ThreeParameterMessage, y "Send_Private_Fail", de tipo ControlMessage (pues podría ocurrir que se quiera mandar un mensaje a un usuario que no esté en la sala). Por lo tanto, se creó el método "sendPrivateMessage" tanto en NCController como en NCConnector, que se encargan de construir y mandar el mensaje correspondiente al servidor. Por su parte, en NCServerThread, si recibe un mensaje privado, llama a un nuevo método definido en NCRoom, "sendMessage", que envía el mensaje al usuario que se indique. En este método, si el usuario está dentro de la sala se le envía un mensaje "Send_Private_In", pero si no está, se envía un mensaje "Send_Private_Fail" al remitente, informándole de que no pudo enviarse el mensaje que deseaba. Por lo tanto, hay que modificar de nuevo los métodos "processIncomingMessage" y "receiveMessage" de NCController y NCConnector para que puedan recibir mensajes de los tipos "Send_Private_Fail" y "Send_Private_In". Será en NCController donde, según el caso, se imprima por pantalla en consecuencia. Nótese que se implementó un mensaje para cuando no se puede enviar el mensaje privado, pero no un mensaje informando de que se envió correctamente. Decidimos hacerlo así pues no creemos que tenga mucho sentido que el cliente sea informado de que su mensaje se envió satisfactoriamente cada vez que envíe un mensaje privado, pero si vimos coherente informarle cuando no sea posible (normalmente cuando el usuario al que quiere enviarle el mensaje no está en la sala).

Claramente, para todas estas mejoras, en muchos casos, hubo que modificar también las clases NCCommands y NCShell para añadir y dar soporte a las nuevas funcionalidades que se añadieron. Además, en algunos casos también hubo que crear nuevas variables en las clases, como por ejemplo en NCController, que hubo que añadir una variable para almacenar el texto del mensaje privado.

7. CONCLUSIONES

Como conclusión, estamos muy satisfechos con el resultado final de la práctica, pues finalmente hemos sido capaces de implementar de forma satisfactoria las características básicas, e incluso hemos sido capaces de implementar ciertas mejoras: notificar las entradas/salidas de las salas, posibilidad de renombrar salas, envío de mensajes privados y posibilidad de crear nuevas salas. Además, el hecho de que nuestro documento de diseño entregado estaba correcto también supuso que fuese más sencillo terminar esta práctica de forma correcta. No obstante, nos hubiese gustado poder implementar todas las mejoras, pero debido a la falta de tiempo no fuimos capaces.

Claro está, nos hemos encontrado también con ciertas dificultades a lo largo del desarrollo del proyecto, pero con la ayuda del profesor de prácticas, Eduardo, finalmente fuimos capaces de solventar el problema. Por ejemplo, una pequeña dificultad encontrada fue al almacenar la hora del último mensaje de la sala, ya que no sabíamos cómo usar correctamente el tipo Date(). También tuvimos muchas situaciones en las que el programa no se ejecutaba de forma correcta y no conseguíamos encontrar el problema, y al final simplemente era que no habíamos puesto "break" al final de alguna sentencia de un switch. Sin embargo, creemos que el mayor tiempo intentando encontrar y solucionar un problema lo gastamos al implementar la mejora de renombrar salas, pues en un principio no se actualizaba el nombre de la sala actual de todos los usuarios, solo se actualizaba el NCServerThread del que renombraba, y esto provocaba comportamientos del programa erróneos. Al final, pudimos darnos cuenta de dónde se encontraba el error e implementamos una solución que puede no ser idónea, pero creemos que funciona correctamente.

Así, pensamos que ésta ha sido una práctica amena y que nos ha ayudado a comprender mejor los conceptos vistos en la teoría de la asignatura, además de que hemos podido ver cómo funcionan todos estos conceptos de una manera más práctica.