PROYECTO NANOCHAT

Pablo Miralles González
Jose Antonio Lorencio Abril

Índice

Introducción

Presentamos nuestro proyecto NanoChat. En este proyecto diseñamos e implementamos un programa para chatear con otras personas a través de la red, desde equipos distintos, utilizando TCP y algo de UDP. El programa está hecho en Java.
Para el diseño de los mensajes y los protocolos hemos tenido en cuenta todas las mejoras, excepto la relativa al envío de archivos binarios a través de una sala. La mejora relativa a mostrar el historial de mensajes viene especificada en los autómatas, y en el programa, sin embargo, no la hemos añadido en el apartado de diseño porque aún no hemos abordado este problema con claridad.

Diseño de los protocolos

Formato de los mensajes de Directorio

En el directorio podemos encontrar los siguientes mensajes:

- Registro: consta de cuatro campos, que son Code (1B), IP (4B), Puerto (4B) y Protocolo (1B). Su Code es el 1.
- Consulta: formado por dos campos, Code (1B) y Protocolo (1B). Su Code es el 2.
- ACK: únicamente tiene el campo Code (1B), que es el 3.
- Info: constituido por tres campos, Code (1B), IP (4B) y Puerto (4B). Con Code el 4.
- NoEncontrado: formado por el campo Code (1B), y es el 5.

Por ejemplo, supongamos que el Directorio está en funcionamiento, y queremos registrar nuestro servidor, que funciona con el protocolo Z. Si nuestra IP es X, y vamos a escuchar por el puerto Y, enviaremos el siguiente mensaje al directorio:

1	X	Y	Z

Y nos responderá, tras registrarnos como servidor de chat:

3

Supongamos ahora, que somos un usuario en busca de un servidor que implemente el protocolo Z', en cuyo caso enviaremos el siguiente mensaje:

2	Z'

Y el servidor nos responderá en consecuencia:

Si ha encontrado tal sala, con IP X' y puerto de escucha Y':

4	X'	Y'

Si no es así:

5

A partir de este momento, si la respuesta fue afirmativa, podremos comunicarnos directamente con el servidor de chat que buscábamos.

Formato de los mensajes de Chat

Debido a la elevada cantidad de mensajes de chat, los presentaremos ordenados por grupos, atendiendo a las asociaciones con las que definimos las distintas clases de Java.

Mensajes de Control:

Constan de un único campo, el campo de operación Op (Byte). Sirven en su mayoría para informar si una petición se ha procesado correctamente o no, aunque en algunos casos se usan para solicitar alguna acción por parte del servidor. Son los siguientes:

Nombre	Nick_Ok	Nick_Dup	Room_List	Enter_Ok	Enter_No	Exit
Opcode	8	9	10	11	12	13
Nombre	Send_Ok	Send_No	Info	Rename_Ok	Rename_No	History
Opcode	15	16	17	18	19	20
Nombre	Not_An_Admin	User_Is_Admin	User_Not_Found	Room_Ok	Room_No	
Opcode	21	24	22	25	26	

Mensajes de Sala:

Están formados por dos campos, uno de ellos es la operación Op, y el otro es el campo Name (String), que indicará el nombre del ente sobre el que queremos realizar la operación especificada, el nombre con el que queremos registrarnos, o con el que registrar una sala.

Nombre	Nick	Enter	Remove	Admin	Ascenso	GTFO	CreateRoom
Opcode	1	2	3	4	5	6	7
Name	Nombre de usuario	Nombre de sala	Nombre de usuario	Nombre de usuario	Nombre de usuario	Nombre de usuario	Nombre de sala

Mensajes de Texto:

Constituidos por tres campos: el campo Op, el campo User (String), que indica el destinatario del mensaje, y el campo Text (String), que almacena el mensaje de texto enviado.

Nombre	Send_Req	Send_Res	User_Expelled
Op	29	30	23
User	Broadcast / Nombre de usuario	Broadcast / Nombre de usuario	Broadcast
Text	Texto a enviar	Texto a recibir	Recibir: “The user *user* has been expelled”

Mensajes de Argumento Variable

Estos mensajes destacan por el hecho de que uno de sus campos puede tener más de un argumento.
Constan de dos campos: el campo Op y el campo salas (Conjunto de descripciones de sala), que es el argumento variable, pues contendrá la información relativa a tantas salas como la cantidad que haya en el servidor, o solo de una, dependiendo del comando utilizado.

Nombre	Room_List_Res	Info_Res
Opcode	27	28
Salas	Información relativa a todas las salas del servidor	Información relativa a la sala en la que se encuentra el usuario que pide la información.

Mensajes de Entrada/Salida (IO):

Hay un único mensaje con este formato, que consta de los campos Op, Type (boolean) y Name (String). Sirve para informar a los usuarios de una sala de la entrada y salida de otros usuarios de la misma.

Nombre	IO
Opcode	14
Type	True (In) / False (Out)
Name	Nombre de usuario

Veamos un ejemplo:
Supongamos que entramos al servidor y ya hemos registrado un nick. Ahora queremos ver las posibles salas a las que podemos entrar. La secuencia de mensajes sería la siguiente:

1. Teclearíamos el comando roomlist.
2. Se enviaría el siguiente mensaje desde el proceso cliente al servidor:

10

3. El servidor lo recibiría, y le respondería con:

27	*Salas creadas en el servidor*

4. El proceso cliente recibiría este mensaje, e imprimiría por pantalla la información relativa a cada sala.

Si nos decidimos a entrar en una sala, las acciones serían:

1. Introduciríamos el comando enter *Sala*.
2. El proceso cliente enviaría al servidor el siguiente mensaje:

2	*Sala*

3. El servidor lo recibiría, comprobaría que esa sala, efectivamente, existe, nos haría entrar a la sala y nos notificaría de ello, enviándonos:

11

4. Además, enviaría a los demás usuarios de la sala, para informarles de mi entrada, el siguiente mensaje:

14	True	*Mi_Nombre*

Autómatas

Los autómatas se encuentran como archivos .jpg (siendo estos *automata_cliente.jpg*, *automata_directorio.jpg* y *automata_servidor.jpg*) en la carpeta principal del proyecto, pues al insertarlos como imágenes en este documento la resolución disminuía demasiado, imposibilitando una lectura cómoda de los mismos.

Detalles sobre los principales aspectos de la implementación

Implementación del formato de los mensajes

Nuestra implementación debía ser utilizando codificación field-value, por ello, partimos de la clase abstracta *NCMessage* del paquete *MessageFV*.
Esta clase tiene como único atributo el opcode (byte), que indicará la operación a realizar.
Definimos como constantes:

- Los opcodes de todas las operaciones que codificaremos.
- Los caracteres especiales, que usaremos como delimitador, separador y fin de línea.
- La cadena 'operation', que pertenece a la nomenclatura de codificación FV.
- Las distintas operaciones válidas, relacionadas con sus respectivos opcodes, mediante la utilización de dos arrays.
- Una cadena que permite diferenciar los mensajes enviados a un solo usuario, de los enviados a todos los usuarios de una sala, llamada *BROADCAST_NAME.*

A partir de este punto, termina la estructura general de todos los mensajes, y debemos ahora organizarlos en subclases, atendiendo a la estructura específica que comparten los distintos mensajes. Como ya adelantamos en el apartado anterior, la agrupación queda como sigue:

NCControlMessage: representan los Mensajes de Control. No añaden nada nuevo a la estructura de NCMessage, solo implementamos los métodos *toEncodedString*, que transcribe la información relativa a un mensaje de este tipo al formato FV, y *readFromString*, que básicamente es la operación inversa. Se entiende que estos métodos se implementan para todos los tipos de mensajes, variando, claro está la cantidad de campos codificados o decodificados, por este motivo no vamos a repetir en cada clase que implementamos estos métodos.

NCRoomMessage: representan los Mensajes de Sala. Añaden el campo *name*, así como la constante necesaria para su codificación como mensaje, 'name'.

NCTextMessage: son la implementación de los Mensajes de Texto, y tienen dos nuevos campos, *text* y *user*, y las relativas constantes de codificación 'text' y 'user'. Se implementa el método *toPrintableString*, que devuelve el mensaje en el formato en el que queremos que se muestre a los usuarios destinatarios (o sea, 'name: text').

NCVarMessage: implementan los Mensajes de Argumento Variable. Contiene el campo *salas*, que es un arrayList de NCRoomDescription, para permitir dar la información de una sola sala o de todas las salas del servidor. El campo necesario para la codificación es la cadena constante 'info'.

NCIOMessage: representan los Mensajes de Entrada/Salida e incorporan los campos *type* (boolean) y *name* (String) y las constantes ‘type’ y ‘name’ relativas a la codificación.

Mecanismo de gestión de salas

La gestión de salas es llevada a cabo por las clases del paquete *server*, concretamente por la clase NCServerManager, que tiene como atributos:

- nextRoom: indica la siguiente habitación que se creará.
- users: conjunto de los usuarios que acceden al servidor.
- rooms: hashmap que asocia el nombre de cada habitación registrada en el servidor con un objeto de la clase NCRoomManager, encargada de la gestión de una sala.
- Además, se definen las constantes INITIAL_ROOM y ROOM_PREFIX.

En la inicialización de un objeto de esta clase se crea una primera sala a la que se puede acceder desde el momento de inicialización.
Esta clase permite, además, añadir nuevas habitaciones, obtener una lista con las habitaciones ya existentes, añadir usuarios al servidor, borrarlos y permite a un usuario entrar a una habitación del servidor, así como abandonarla.

Por otro lado, la encargada de la gestión de cada sala es la clase *NCRoomManager*, que, al ser abstracta, hemos completado en la clase *NCRoomManagerU*. Esta clase tiene los siguientes atributos:

- roomName: atributo heredado de la clase padre. Es un String que indicará el nombre de la sala.
- mapa: hashmap que asocia cada usuario de sala con su Socket para poder enviar y recibir mensajes. Los usuarios se representan mediante la clase *Member*, clase auxiliar que consta del nombre de usuario (String) y un valor booleano que indica si el usuario es Admin (true) o no (false).
- timeLastMessage: long que permite guardar el instante en el que por la sala se envió un mensaje de Broadcast.

Se implementan los métodos abstractos de la clase padre, que son registrar un usuario, enviar un mensaje de Broadcast, eliminar un usuario, cambiar el nombre de la sala, obtener la descripción de la sala (devuelve un objeto NCRoomDescription, formado por el nombre de la sala, el conjunto de usuarios de la sala y el momento de envío del último mensaje de Broadcast) y contabilizar el número de usuarios activos en la sala.
Además, hemos añadido el método *privateMessage*, para enviar un mensaje a un solo destinatario, y no a todos los miembros de la sala.

Mejoras adicionales implementadas

- Posibilidad de crear nuevas salas en el servidor: hemos añadido el comando *createroom* en la clase NCController, que debe utilizarse en el estado *CONNECTED* del autómata del cliente. También hemos incorporado este comando en la clase NCShell, de modo que el intérprete pueda procesarlo correctamente.
  Para usar el comando debemos escribir: *createroom 'NombreSala'*

  En NCConnector hemos creado el método createRoom, que envía al servidor un mensaje indicando la intención del cliente de crear una nueva sala. Si recibimos un RoomOk significa que la sala se ha creado correctamente, si recibimos un RoomNo quiere decir que ya había una sala con ese nombre. Además, tenemos en cuenta que pueden llegarnos otro tipo de mensajes en este momento, como un mensaje destinado a nosotros, o la notificación de que otro usuario ha entrado a la sala,... Esto siempre lo hacemos, por lo que no volveremos a mencionarlo.

  En NCServerThread, al switch que actúa en consonancia con los mensajes que recibe, hemos añadido la opción de que reciba un mensaje createRoom. Si esto sucede, extraerá el nombre de la habitación, e intentará añadirlo al hashmap de salas de NCServerManager, si esta operación se efectúa correctamente, devolverá true, y en otro caso false.

- Enviar un mensaje privado a un usuario de la sala: de forma similar al anterior, añadimos el comando a NCController (debe utilizarse, esta vez, en el estado ROOMED del autómata del cliente), así como modificado lo necesario para que el intérprete pueda procesarlo.
  Para utilizarlo correctamente escribiremos: *privatesend* 'Destinatario' 'Mensaje'

  En NCConnector, utilizamos el mismo método que para enviar un mensaje de Broadcast. Así, envíamos al servidor una petición de envío de mensaje, pasándole el mensaje y el destinatario (el destinatario, en el caso de un mensaje normal, es la cadena reservada para Broadcast). Si nos devuelve SendOk, el mensaje se habrá enviado correctamente, y así se lo notificamos al usuario, si devuelve SendNo, significará que el destinatario no está en la sala.

  En NCServerThread usamos el mismo método que para enviar un mensaje de Broadcast, diferenciándose por el parámetro destinatario pasado como argumento. Así, al ver que el destinatario es distinto de la cadena reservada para broadcast, buscamos el socket asociado al usuario con ese nombre, y, en caso de encontrarlo, le enviamos el mensaje, indicándole el remitente. Nótese que no hay ninguna diferencia entre recibir un mensaje privado o recibir un mensaje de Broadcast.

- Administradores (incompleta): para poder diferenciar un usuario admin de uno que no lo es nos aprovechamos del hecho de que un usuario solo puede pertenecer a una sala en un momento dado. Por ello creamos la clase auxiliar Member, que consta del nombre de usuario y de un valor booleano, indicador de si el usuario es administrador (true) o no (false).

Un usuario será administrador de una sala si es el primero que entra a esta.

- Nota: las demás mejoras no están implementadas a fecha de entrega de esta memoria. Se completará esta en las semanas posteriores a dicha entrega.

Conclusión

Se trata sin duda del mejor proyecto que hemos realizado hasta la fecha, en términos de asemejarse a la idea que teníamos de desarrollo de software cuando entramos en la carrera. Seguramente sea por el desarrollar distintas partes que luego interactúan entre sí, lo cual resulta bastante novedoso para nosotros.

Obviamente también ha sido bastante aclaratorio en cuanto al uso de redes a nivel de software, cosa que usaremos mucho en el futuro, pues el desarrollo web es seguramente el sector más atractivo para muchos de nosotros.

Por decir algo negativo sobre el proyecto, es cierto que podríamos haber tenido algo más de autonomía, aunque suponemos que estamos aún muy verdes para poder hacer las cosas de forma independiente.

En resumen, consideramos el desarrollo de esta actividad como una experiencia muy positiva, y esperamos que este tipo de prácticas se repitan en lo que nos queda de carrera.