DOCUMENTACIÓN APLICACIÓN NANOCHAT

Alberto Gálvez Gálvez, Jaime Sánchez Sánchez

UNIVERSIDAD DE MURCIA, 2º INGENIERÍA EN INFORMÁTICA 4/2021

ÍNDICE

Introducción

En este documento se especifica el diseño de los mensajes usados para la comunicación entre los diferentes programas que conforman la práctica entera. Además, se detalla mediante tres autómatas las diferentes fases por las que pasan cada uno de los programas.

El programa usa una comunicación mediante el protocolo *UDP* para las comunicaciones del servidor de Directorio con el servidor de Chat y con el propio cliente. Se usan mensajes binarios con un campo *opcode* que indica el tipo de mensaje transmitido.

Para la comunicación entre el servidor de Chat y el cliente *Nanochat* se usa el protocolo *TCP*, los datos se transmiten mediante mensajes de texto. Puesto que usamos un byte para la codificación del identificador de protocolo, a la suma de nuestros DNIs le aplicamos el módulo 128, obteniendo 37. Puesto que el número es impar codificamos los mensajes TCP usando *field:value*.

El uso de ambos protocolos se encuentra reflejado en los autómatas donde se encuentra unificada las partes de *UDP* con las partes de *TCP*.

Los autómatas indican el estado en el que se encuentra cada programa y las acciones permitidas en un momento específico.

Mensajes UDP

En esta sección se muestran los mensajes usados para el registro del servidor de Chat en el servidor de Directorio y la consulta de un servidor de Chat por parte de un cliente *Nanochat*.

- Mensaje que envía un servidor de Chat para registrarse en el servidor de Directorio

Opcode	*param1*	*param2*
1	*id.protocolo*	*puerto*
(1 byte)	*(1 byte)*	*(4 bytes)*

- Mensaje que envía el servidor de Directorio para confirmar el registro del servidor de Chat

Opcode
2
(1 byte)

- Mensaje que envía el cliente *Nanochat* al servidor de Directorio para consultar la dirección de servidor de Chat

Opcode	*param1*
3	*id.protocolo*
(1 byte)	*(1 byte)*

- Mensaje que envía el servidor de Directorio al cliente *Nanochat* con toda la información que el cliente había pedido

Opcode	*param1*	*param2*	*param3*	*param4*	*param5*
4	*puerto*	*ip1*	*ip2*	*ip3*	*ip4*
(1 byte)	*(4 bytes)*	*(1 byte)*	*(1 byte)*	*(1 byte)*	*(1 byte)*

- Mensaje que envía el servidor de Directorio al cliente *Nanochat* para informarle de que la consulta no ha podido efectuarse

Opcode
5
(1 byte)

Mensajes TCP y comandos

En esta sección encontramos el formato de los mensajes de texto utilizados en la comunicación entre el servidor de Chat y el cliente *Nanochat* y en que comandos se usa cada uno.

Los distintos mensajes están representados por una clase Java. Hemos intentado crear el menor número de clases posibles reutilizando formatos para distintos mensajes.

NCOneFieldMessage

Se usa en los comandos:

- *Roomlist*: obtenemos una vista con la información de las salas en el servidor, solo puede usarse en el estado *OutOfRoomState*. *Opcode = GetRoomList*
- *Info*: obtenemos una vista con la información de la sala en la que nos encontramos actualmente, solo puede usarse en el estado *InRoomState*. *Opcode = GetInfoRoom*
- *Exit*: salimos de nuestra sala actual, solo puede usarse en el estado *InRoomState*. *Opcode = Exit*
- *Create*: creamos una nueva sala de chat en el servidor, solo puede usarse en el estado *OutOfRoomState*. *Opcode = Create*

Además, este tipo de mensajes los usa el servidor para avisar al cliente de si una acción ha podido llevarse a cabo satisfactoriamente o no. Por ejemplo, para informar de si el nombre de usuario con el que nos queremos registrar ya está escogido o no.

El formato de este tipo de mensajes sería algo parecido a lo siguiente:

> *operation: <opcode>\n*
>
> *\n*

NCQueryMessage

Se usa en los comandos:

- *Nick*: se usa para registrarse en el servidor de chat con el nombre especificado como parámetro, solo puede usarse una vez y en el estado *PreRegistration*. *Opcode = Nick*
- *Enter*: se usa para entrar a la sala especificada como parámetro, solo puede usarse en el estado *OutOfRoomState*. *Opcode = EnterRoom*
- *Rename*: se usa para cambiar el nombre de la sala en la que nos encontramos, solo puede usarse en el estado *InRoomState*. *Opcode = Rename*

Este tipo de mensajes se usa cuando debemos especificar un parámetro, como podría ser el nombre de una sala. Aparte de en los comandos anteriormente mencionados

este formato lo usa el servidor para mandar un mensaje a todos los integrantes de la sala informando de las salidas y entradas de otros usuarios a la sala.

El formato de este tipo de mensajes sería algo parecido a lo siguiente:

operation: <opcode>\n

Query: <param>\n

\n

NCSendRoomListMessage

Este formato es usado por el servidor para enviar la información de una o varias salas. Tanto si el usuario teclea *info* o *roomlist* el servidor usa este formato para codificar la información de la sala.

Realmente en el mensaje se codifica la información de una lista de salas, cuando tecleamos *info* solo habría una sala en la lista.

El formato de este tipo de mensajes sería algo parecido a lo siguiente:

operation: <opcode>\n

rooms: nameRoom1%userlist1%lastTimeMessage1&
nameRoom2%userlist2%lastTimeMessage2&…&
nameRoomN%userlistN%lastTimeMessageN&\n

\n

userlistN -> user1; user2; user3; …; userN;

NCSendMessage

Se usa en los comandos:

- *Send*: se usa para mandar un mensaje a todos los usuarios de la sala actual , solo puede usarse una vez y en el estado *InRoomState*. *Opcode = Send*

El formato de este tipo de mensajes sería algo parecido a lo siguiente:

operation: Send \n

send: (message) \n

sender: sender\n

\n

NCPrivateMessage

Se usa para mandar un mensaje privado a un usuario de la sala en la que estemos.

El formato de este tipo de mensajes sería algo parecido a lo siguiente:

operation: private\n

send: (message) \n

sender: sender\n

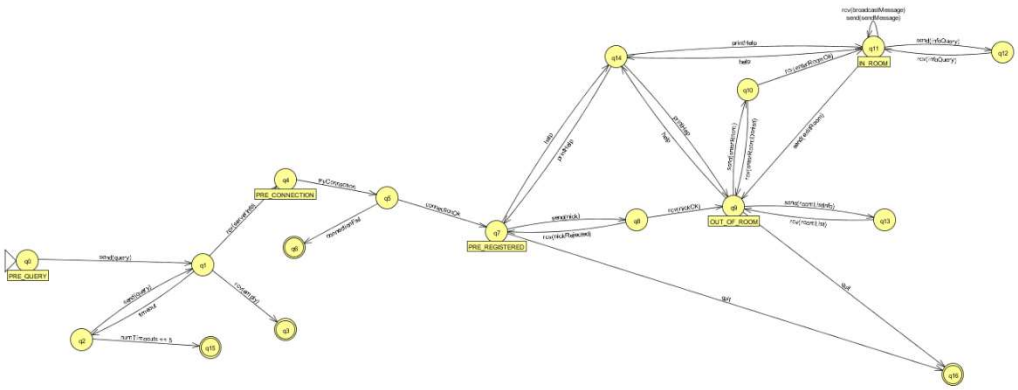
receiver: receiver\n

\n

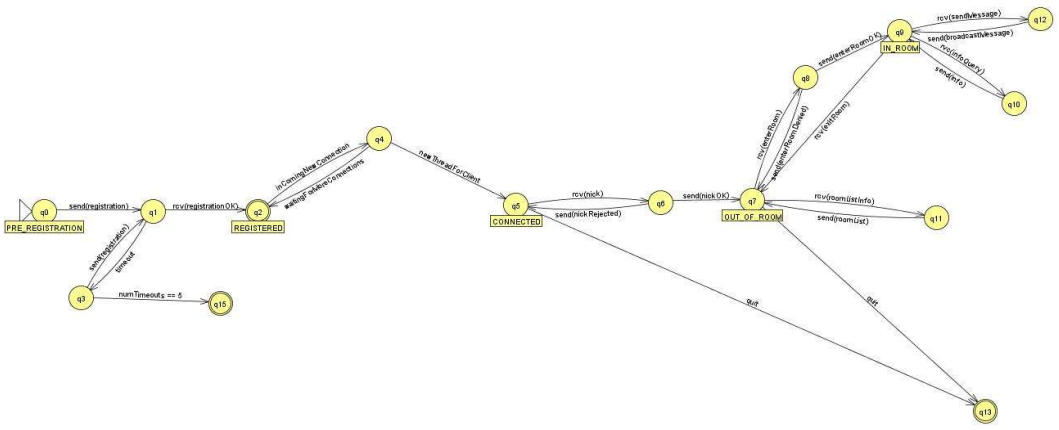
Autómatas

En esta sección se encuentran los autómatas correspondientes a cada programa. Estos autómatas marcan las fases por las que pasan los programas y las acciones permitidas en cada momento por cada uno de ellos.

Autómata del Cliente

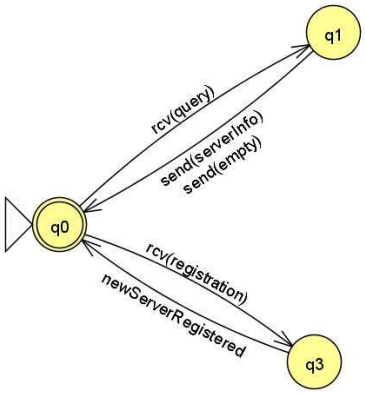


Autómata del servidor de chat



En el estado *q4* realmente lo que ocurre es que se crea un nuevo hilo, instancia de la clase *NCServeThread*, y la clase *NanoChatServer* seguiría a la espera de más conexiones.

Autómata del servidor de Directorio



Ejemplo de transmisión de mensajes

A continuación, se ejemplificarán algunas de las posibles conversaciones que podríamos encontrar entre los tres programas.

Al comienzo servidor y cliente se comunican con el directorio para registrarse y para pedir información acerca de un servidor de chat, respectivamente. El autómata del servidor comienza en el estado *PRE_REGISTRATION* y el del cliente en *PRE_QUERY*.

- Servidor se registra en el Directorio:

    El servidor debe enviar el mensaje con *OPCODE* correspondiente y el directorio lo registrará.

    Mensaje servidor:

Opcode	*param1*	*param2*
1	*37*	*6969*
(1 byte)	*(1 byte)*	*(4 bytes)*

    Respuesta Directorio:

Opcode
2
(1 byte)

-Cliente pide *IP* y *puerto* de un servidor de chat al Directorio:

El cliente debe enviar el mensaje con *OPCODE* correspondiente y el directorio le responderá con un mensaje dándole la información pedida o, por el contrario, con un mensaje informándole de que el servidor consultado no existe.

Mensaje cliente:

Opcode	*param1*
3	*37*
(1 byte)	*(1 byte)*

Respuesta Directorio:

Opcode	*param1*	*param2*	*param3*	*param4*	*param5*
4	*6969*	*127*	*0*	*0*	*1*
(1 byte)	*(4 bytes)*	*(1 byte)*	*(1 byte)*	*(1 byte)*	*(1 byte)*

A partir de aquí el cliente y el servidor podrían empezar la conexión. El autómata del cliente entra al estado *PRE_CONNECTION* y el del servidor al estado *REGISTERED*.

Si la conexión es exitosa el cliente entra al estado *PRE_REGISTRATION* y el servidor al estado *CONNECTED*

-Cliente se registra en el servidor de chat:

Mensaje cliente:

operation: nick\n

query: jaime\n

\n

Respuesta servidor:

operation: nickOk\n

\n

El cliente ya estaría registrado en el servidor. El cliente y el servidor se encuentran en el estado *OUT_OF_ROOM*

El cliente puede pedir la lista de salas del servidor:

-Cliente pide la lista de salas:

Mensaje cliente:

operation: GetRoomList \n

\n

Respuesta servidor:

operation: SendRoomList \n

Rooms: roomA%alberto;luis%45& \n

\n

El cliente puede solicitar entrar a la sala que quiera. El servidor puede aceptar o denegar la entrada del cliente a la sala solicitada

-Cliente pide entrar a una sala:

Mensaje cliente:

operation: Enter \n

query: roomA \n

\n

Mensaje servidor:

operation: enterRoomOk\n

\n

El cliente y servidor entran al estado *IN_ROOM*.

El cliente puede pedir información de la sala o mandar un mensaje.

-Cliente manda un mensaje a la sala:

Mensaje cliente:

operation: send \n

Sender: jaime \n

Message: hola chicos solo paso a saludar\n

\n

Mensaje servidor:

Este tipo de acción por parte del cliente no requiere una respuesta del servidor

El cliente podría salir de sala y finalizar su conexión con el servidor

-Cliente sale de la sala:

Mensaje cliente:

operation: exit \n

\n

Mensaje servidor:

Este tipo de acción por parte del cliente no requiere una respuesta del servidor

-Cliente finaliza conexión con el cliente:

Mensaje servidor:

operation: quit \n

\n

Mensaje servidor:

Este tipo de acción por parte del cliente no requiere una respuesta del servidor

Detalles y decisiones de diseño

En esta sección se hablará sobre las decisiones tomadas en la implementación de los tres programas en aspectos como la implementación de los mensajes, gestión de las salas y mejoras adicionales que hayamos decidido añadir.

Mensajes UDP

En cuanto a los mensajes *UDP* tenemos los necesarios para que la comunicación del cliente y el servidor con el directorio sea satisfactoria.

Además, también buscamos que la decodificación de los mensajes fuese lo más sencilla posible. El mensaje más complejo seguramente sea el mensaje donde se codifica la dirección *IP* del chat para enviársela al cliente. La manera más sencilla que encontramos para construir este mensaje era con un campo para cada número de la dirección *IP*.

El tamaño de los campos es algo más trivial y simplemente buscamos aprovechar el espacio lo mejor posible. Por eso siempre que un número cupiera en un solo byte, intentamos usar uno; en caso contrario, usamos cuatro.

Por último, tenemos implementado un sistema que siempre que se agota el *timeout* se envía el mensaje de nuevo, si el *timeout* se agota cinco veces seguidas entonces el programa aborta.

Mensajes TCP

En la parte de *TCP* tenemos una cantidad mayor de mensajes. Se podría dividir en mensajes de petición y mensajes que transmiten una información concreta.

En cuanto a los mensajes de petición son aquellos que usa el usuario para hacer peticiones al servidor de chat. Por ejemplo, para entrar a una sala, mandar un mensaje, obtener la información de las salas y muchos más. Estos mensajes únicamente se diferencian en el código de operación. Se trata de los mensajes codificados con la clase *NCOneFieldMessage*, ese único campo que tiene es el que va variando.

Por otra parte, tenemos los mensajes que contienen información concreta. Estos mensajes son más complejos que los anteriores debido a que tiene más campos. Cada campo codifica un trozo del mensaje. Es posible que el mensaje solo necesite codificar un nombre, varios campos o varios datos codificados en el mismo campo.

Cuando solo queremos codificar una sola cadena usamos la clase *NCQueryMessage* en la que irá variando el código de operación. Dependiendo del código la interpretación

del segundo mensaje puede ser desde un nombre de usuario hasta un mensaje que el servidor de chat envía a todos los usuarios informando de la entrada o salida de un nuevo usuario.

Por otra parte, si queremos enviar un mensaje privado o un mensaje a la sala de chat necesitamos más campos. El problema con estos dos mensajes es que un mensaje privado tiene más campos que un mensaje para toda la sala. Por eso usamos dos formatos para cada uno de ellos, *NCPrivateMessage* y *NCSendMessage*.

Por último, tenemos el caso particular del mensaje que contiene la información de una o varias salas, *NCSendRoomListMessage*. Este tipo de mensaje podría tener varios campos, pero decidimos codificarlo todo en uno, de esta manera nos era más fácil diferenciar entre la información de una sala y otra.

Cuantos más campos tenga un mensaje más bucles hay que realizar en la codificación y decodificación de los mismos, por el contrario, si decidimos codificar en el mismo campo varios datos entonces debemos definir nuevos delimitadores que actúan como frontera entre cada dato en el campo. Elegir uno u otro depende mucho del contenido del mensaje. El mensaje *NCSendRoomListMessage* lo codifica todo en un mismo campo, en este caso porque es más fácil diferenciar entre las salas. El mensaje *NCPrivateMessage* codifica usando dos campos, pero perfectamente podría haber usado uno.

Si el mensaje solo envía una cadena como podría ser un nombre de usuario la codificación es trivial, pues solo necesitamos un campo.

Implementación directorio

Se trata del programa más simple de los tres, solo interviene en la parte de UDP. El directorio se encarga de recibir solicitudes de registro por parte de los servidores de chat y solicitudes de información acerca de un servidor de chat por parte de los clientes. El directorio mapea identificadores de protocolo con servidores, de esta manera el cliente solo necesita saber el identificador de protocolo del servidor al que se quiere contectar.

El directorio es la clase *Directory*, que es donde se encuentra el *main*, y únicamente se queda escuchando posibles solicitudes por parte de un cliente o solicitudes de registro de servidores.

Implementación del cliente y del servidor

La implementación de estos dos programas no es demasiado compleja.

En el lado del cliente, siempre esperamos a que el cliente escriba un nuevo comando. Dependiendo del comando enviamos una petición al servidor de hacer una cosa u otra. El cliente recibe mensajes de confirmación del servidor o mensajes de otros usuarios de su misma sala. El cliente es la clase *NanoChat*, que es donde tenemos el *main*. Pero utiliza muchas otras clases de apoyo como:

- *NCShell y NCCommands*: en estas clases se encuentra todo lo referente a leer nuevos comandos y sus argumentos.
- *NCControlle*r: sería la clase donde, dependiendo del comando, realizamos una petición u otra.
- *NCConnector*: es la clase que se comunica directamente con el servidor mediante el envío de paquetes *TCP*.
- *NCMessage* y toda su jerarquía: el cliente utiliza muchos mensajes para realizar peticiones al servidor.

El servidor siempre está esperando nuevas peticiones de conexión al servidor. Cuando una petición se acepta se crea un nuevo hilo que se encarga de atender las peticiones del cliente que se haya conectado. El servidor sería la clase *NanoChatServer*, pero la clase que se encarga de gestionar las peticiones de cada usuario seria *NCServerThread*. Las clases de apoyo que utiliza serían:

- *NCServerManager*: se trata de una clase que gestiona la creación de salas y la entrada o salida de clientes del servidor.
- *NCRoomDescription*: esta clase sirve para mostrar los atributos de una sala como pueden ser los usuarios que hay en ella, tiempo del último mensaje y nombre.
- *NCRoomManager* y su jerarquía: esta es la clase que representa a las salas.
- *NCMessage* y toda su jerarquía: el servidor utiliza muchos mensajes para realizar atender las peticiones del cliente.

Implementaciones adicionales

Hemos añadido un total de cuatro nuevas funciones al servidor.

La primera de ellas es que cuando un cliente se sale de una sala o entra se envía un mensaje avisando a todos los usuarios de la sala de que un cliente ha salido o entrado, algo similar a lo que ocurre en la aplicación *Whatsapp* cuando alguien sale o entra a un grupo. Para la implementación de esta función no ha sido necesario añadir ningún mensaje nuevo ni ningún comando.

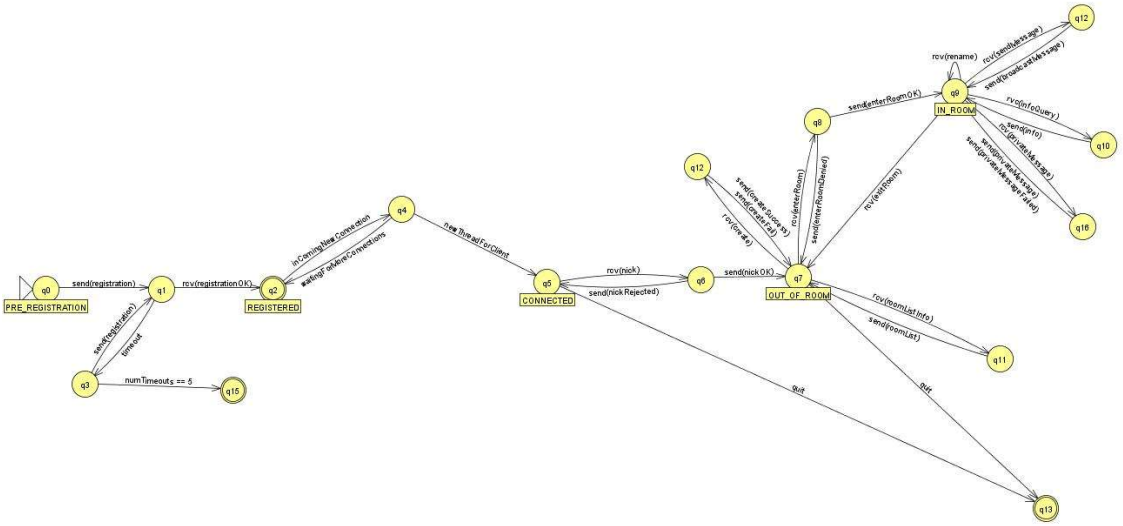
La segunda funcionalidad es crear nuevas salas. Para esto si se ha necesitado un nuevo comando (*create*). Las salas tienen un nombre por defecto que sería la palabra *Room* seguida de una letra del abecedario empezando por la letra *'A'*. Solo pueden crearse hasta veintiséis salas.

La tercera funcionalidad sería renombrar una sala. Para esto se ha necesitado un nuevo comando (*rename*). Cuando diseñamos este mensaje teníamos dos posibilidades, que el nombre se cambiase desde fuera de la sala o desde dentro. Optamos por la segunda porque de esta manera nos ahorramos codificar un nuevo campo y el tener que lidiar con un segundo argumento.

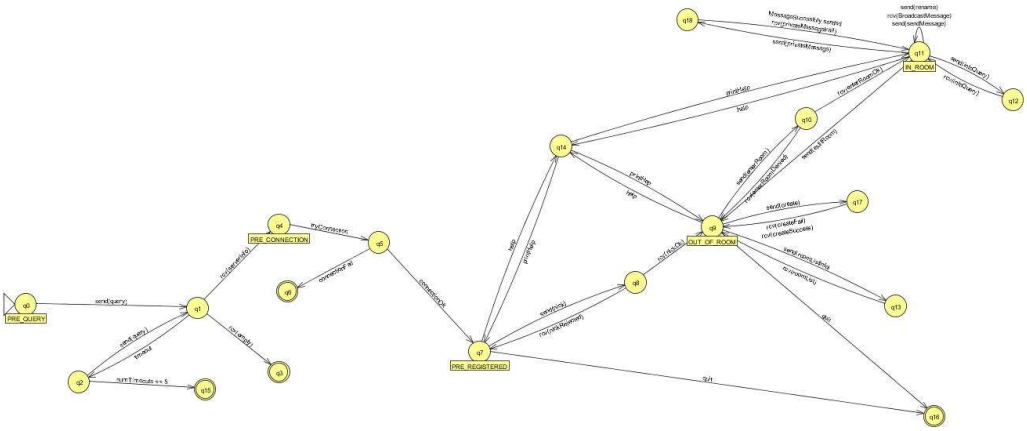
La última funcionalidad implementada es mandar mensajes privados. Para esto se ha necesitado un nuevo comando (*private*) y un nuevo tipo de mensaje (*NCPrivateMessage*), este mensaje tiene un campo para indicar quién envía el mensaje, a quién se le quiere enviar el mensaje y el mensaje en sí. Tanto en este tipo de mensaje como en el *send,* es importante que se codifique quién envía el mensaje para que después los demás clientes puedan imprimir quién ha sido el emisor. El cliente receptor de un mensaje privado no nota nada distinto entre un tipo de mensaje y otro, pero el resto de usuarios no visualizan el mensaje privado. El tiempo de último mensaje se actualiza al enviar un mensaje privado.

Los autómatas finales quedarían de la siguiente forma:

Servidor:



Cliente:



En cuanto a estos dos nuevos autómatas cabe puntualizar ciertas cosas. Por un lado, el comando *create* necesita mensajes de confirmación, de esta manera el cliente es informado de si la sala ha podido crearse o no.

El comando *rename* no necesita confirmación, así que el servidor no necesita enviarle nada al cliente.

Finalmente, el comando *private*. Cuando un cliente envía un mensaje privado pueden ocurrir dos cosas. Si el cliente intenta enviarle un mensaje a un usuario que se encuentra en la sala entonces el servidor reenviará el mensaje al receptor sin ningún problema. Si el cliente intenta enviarse un mensaje a sí mismo o a un usuario que no está en la sala, el mensaje no se envía y el cliente es informado de la incidencia.

Conclusiones

En general, las sensaciones con la práctica son muy positivas. Este tipo de proyectos son la mejor forma de afianzar los conocimientos adquiridos en teoría. Por ejemplo, ahora sabemos qué implicaciones reales tiene que *UDP* no sea un canal confiable y que *TCP* si lo sea.

Además, nunca antes habíamos programado nada haciendo uso de la red. Por esto último, las primeras partes del desarrollo de la práctica fueron de gran dificultad. Lo que más nos costó fue, mucha diferencia, el entender la estructura del proyecto, la cual, al principio, abruma bastante por la cantidad de clases y conexiones entre ellas. Sin embargo, tras entender bien la estructura, fuimos cogiendo más y más soltura y acabamos entendiendo el código lo suficiente como para implementar mejoras adicionales sin demasiada complicación.

Cabe destacar también que, la necesidad de encapsular las distintas funcionalidades en distintas clases de una manera tan clara y este enfoque tan descentralizado, nos ha hecho mejorar nuestras habilidades como programadores. Además, el uso del lenguaje *Java* en este proyecto, nos ha hecho afianzar conocimiento de la *Programación Orientada a Objetos*, que quizás tras cursar la asignatura *POO*, no se practicaron lo suficiente quizás.

En resumen, ha sido de gran agrado la realización del proyecto. Sin embargo, quizás, se podría haber sido mayor la satisfacción, si se subiese algún tipo de documentación o tutorial adicional sobre las clases de este, ya que, el comprender como funciona cada clase es de gran complicación.