Documentación de nanoChat

Redes de Comunicaciones

Facultad de Informática de la Universidad de Murcia

Carlos Cañellas Tovar

Grupo 1.1

28 de junio de 2020

Índice general

1 Introducción

El presente documento ha sido realizado con el propósito de explicar el diseño, funcionalidad y funcionamiento del proyecto. Es decir, cómo se ha conseguido que funcione correctamente.

Se explicará y demostrará el desarrollo y funcionamiento de esta aplicación de acuerdo a las pautas indicadas por el profesorado.

No se va a desarrollar completamente una descripción de este programa, puesto que ya está indicado en los materiales provistos por el profesorado para poder iniciarnos en el desarrollo de esta aplicación. Sin embargo sirven como una base para explicar aspectos del proyecto.

2 Diseño de protocolos

Se ha de tener en cuenta que hay dos tipos de comunicación, en los cuales los roles y actores son diferentes. En uno de ellos, existe una comunicación entre dos tipos de clientes (el cliente de chat y el servidor de chat), y el directorio de servidores, el cual recibe peticiones de ambos.

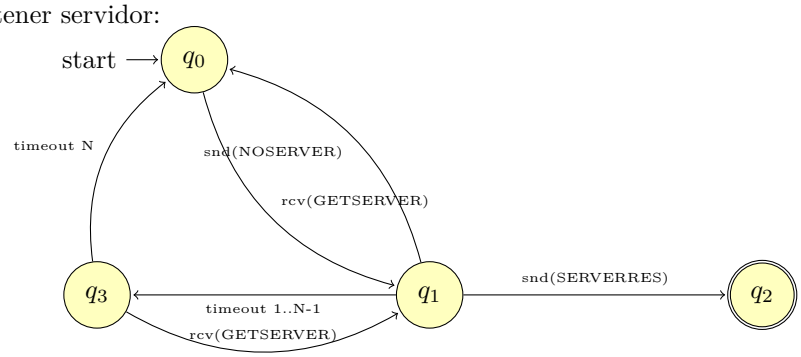
Sus protocolos de transporte ya se conocen por las indicaciones de la práctica. Sin embargo, el proceso de su funcionamiento se indica en los autómatas que se indican a continuación, en la próxima sección del actual capítulo.

2.1 Autómatas

Se listan aquí cuatro autómatas, dos autómatas para la comunicación del cliente y el servidor con el directorio, un autómata para el sentido cliente-servidor y otro para el sentido servidor-cliente.

2.1.1 Autómata de cliente y directorio

Desde el lado del directorio se aceptan conexiones entrantes. El siguiente autómata representa cómo lidia el directorio con las peticiones de clientes de obtener servidor:

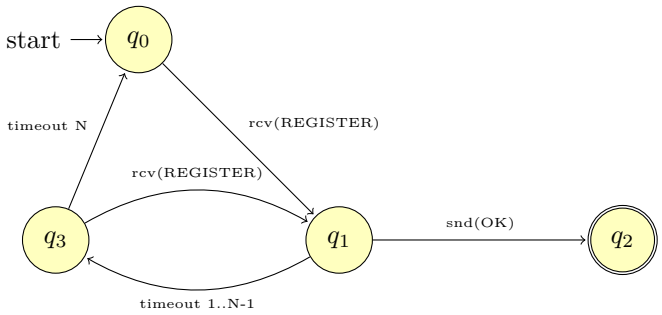


GETSERVER es una orden mediante protocolo sin establecimiento de conexión que se envía desde el cliente y ha de recibirla el directorio. Si esa orden llega (puede perderse, esto es UDP), el directorio puede enviar, en caso de existir tal servidor, su dirección IP almacenada (SERVERRES), o bien indicar que para su protocolo no existe servidor registrado (NOSERVER). Siempre contando con la eficacia de la respuesta y que sea capaz el cliente de recibir tal respuesta.

Si el número de timeouts llega a un N específico (por ejemplo, cinco), no se vuelve a reintentar la conexión.

2.1.2 Autómata de servidor y directorio

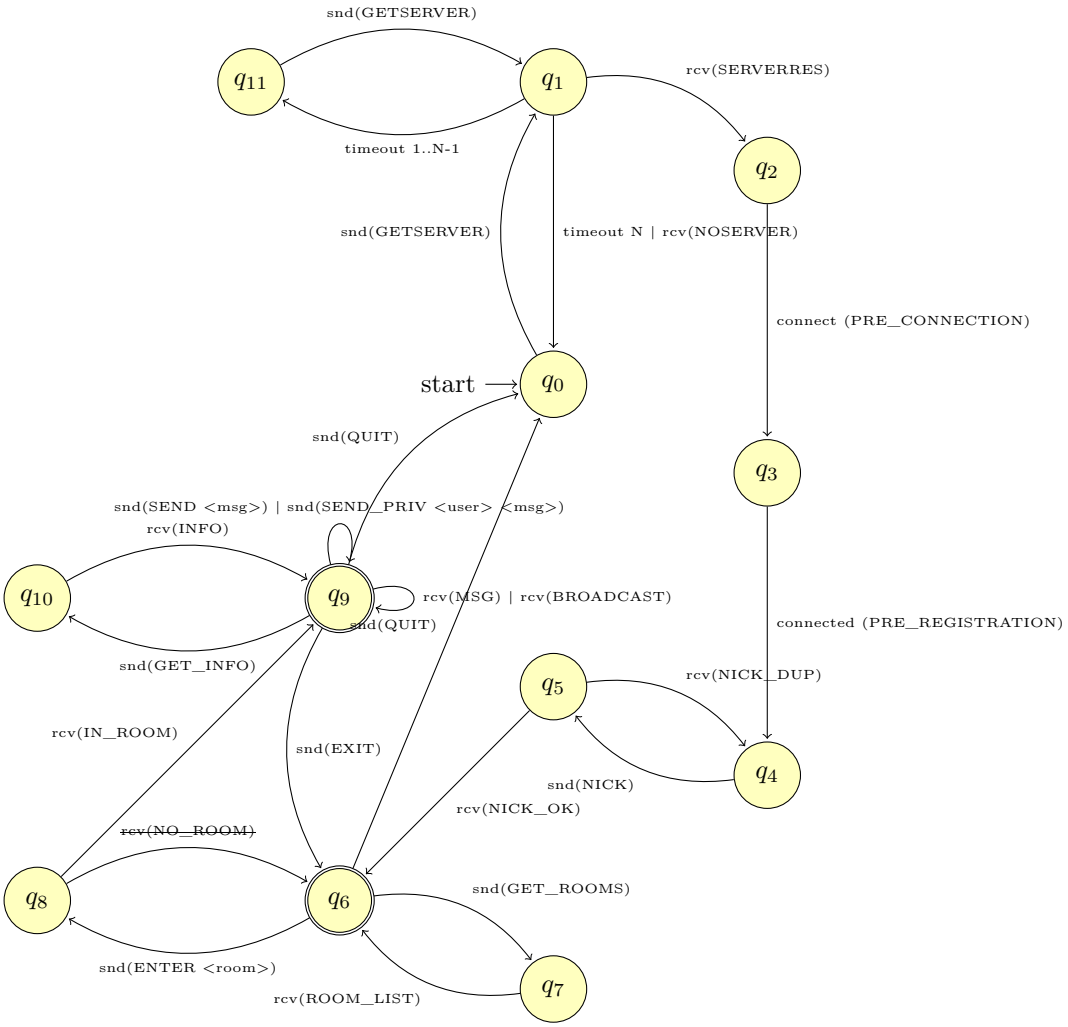
Igualmente, desde el lado del directorio, se aceptan las conexiones de nuevos servidores para ser registrados:



El servidor envía una petición de registro (REGISTER) al directorio. Este debe aceptarlo y responder que sí (OK). Si el mensaje se pierde por lo que sea, salta el timeout y se reintenta el registro hasta un total de N veces, siendo N un número natural arbitrario, por ejemplo cinco.

2.1.3 Autómata del cliente

En el siguiente autómata se muestran todos los estados posibles del cliente, desde que pide un servidor al directorio hasta que envía mensajes en chat, pasando por otros.



Este autómata es más complicado que el resto. Se inicia estando el cliente iniciado pero sin haber conectado aún con el directorio. Entonces realiza una petición a este para obtener el servidor. Si no hay ningún servidor o si el tiempo de espera de recibir una respuesta del directorio se agota, se vuelve al estado inicial. En otro caso, recibe una respuesta con el servidor al cual conectarse.

Tras recibir la respuesta se intentará conectar al servidor. Si la conexión falla, se reintentará. En otro caso se llegará a un estado de pre-registro. Para registrarse el usuario debe enviar al servidor cuál será su nick. Este le dirá al cliente si puede usarlo (NICK_OK) o si está ya en uso (NICK_DUP) y debe elegir otro.

Entonces se llega a un estado ya final en el que el usuario está registrado

pero puede entrar a alguna sala (enviará un ENTER con la sala), consultar las salas disponibles (GETROOMS) o bien irse del servidor (QUIT).

Si pide las salas, el servidor devolverá un mensaje ROOMLIST con las salas disponibles e información sobre ellas.

Si elige irse, vuelve al estado inicial del autómata.

Lo interesante es entrar en una sala, en la que estaba previsto recibir un NO_ROOM en caso de que no existiera, pero la funcionalidad añadida de crear sala ha hecho que el usuario entre en una sala vacía en caso de no existir tal sala (se realiza una inicialización dentro del servidor). Se plasmó en el autómata como posibilidad, pero no se da, por ello el estado está tachado.

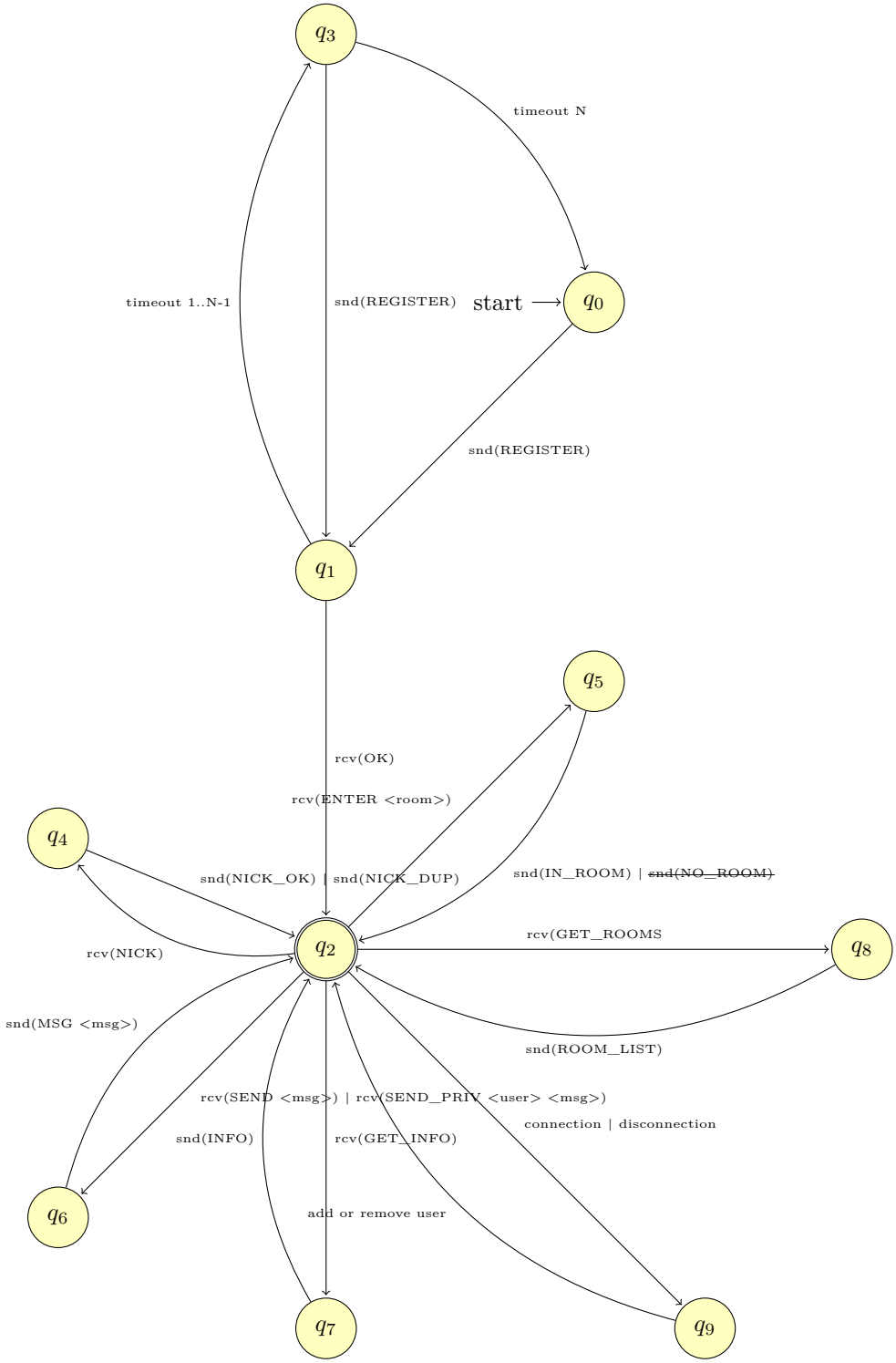
Al entrar en una sala, puedes enviar mensajes, recibir mensajes de otros clientes y del servidor (broadcast), o pedir información de la sala actual. Y también se puede salir de la sala o salir del cliente.

Con todo, quizá sea este el autómata más completo, ilustrando todos los estados del cliente.

Un inciso: después de varios timeout el programa deja de intentar conectarse.

2.1.4 Autómata del servidor

Este autómata es más centralizado que el anterior. Se tiene el estado inicial q_0 y los estados de transición q_1 y q_3. A q_3 se llega tras un timeout y tras varios timeouts se vuelve a q_0. Si el registro se realiza con éxito el servidor llega a un estado final q_2 en el cual recibe peticiones, conexiones y mensajes de clientes y los procesa en cada uno de sus estados *satélite*. En total tenemos seis estados *satélite* para cada petición: establecimiento de *nick*, envío de mensaje (público o privado), obtención de información de salas, obtención de información de la sala actual, o una conexión o desconexión de un cliente, para añadirlo o eliminarlo en la lista de usuarios con su respectivo *socket*.


q_3
timeout N
timeout 1..N-1
snd(REGISTER)
start
q_0
snd(REGISTER)
q_1
q_5
rcv(OK)
rcv(ENTER <room>)
q_4
snd(NICK_OK) | snd(NICK_DUP)
snd(IN_ROOM) | ~~snd(NO_ROOM)~~
rcv(GET_ROOMS
q_8
rcv(NICK)
q_2
snd(MSG <msg>)
snd(ROOM_LIST)
rcv(SEND <msg>) | rcv(SEND_PRIV <user> <msg>)
connection | disconnection
snd(INFO)
rcv(GET_INFO)
q_6
add or remove user
q_7
q_9

2.2 Mensajes

En esta sección se explicarán los diferentes tipos de mensaje que el directorio utiliza, y el protocolo implementado para la comunicación cliente-servidor.

2.2.1 Comunicación con el directorio

La comunicación es UDP. Los mensajes tienen un formato compacto para una mayor eficiencia. Los tipos de mensaje que se van a enviar están guardados en un enumerado de Java, lo cual se verá en la implementación. De momento vamos a describir qué mensajes se usan.

opcode (1 byte)	protocolo (4 bytes)
Valor byte de GETSERVER	78375777

Cuadro 1: Mensaje del cliente para obtener el servidor a partir del protocolo.

opcode (1 byte)
Valor byte de NOSERVER

Cuadro 2: Mensaje del directorio al cliente si no hay un servidor compatible.

opcode (1 byte)	ip_addr (4 bytes)	puerto (tamaño en bytes de 1 int)
Valor byte de SERVERRES	IP	PUERTO

Cuadro 3: Mensaje del directorio para proveer al cliente del *socket* del servidor.

opcode (1 byte)	ip_addr (4 bytes)	puerto (tamaño en bytes de 1 int)
Valor byte de REGISTER	IP	PUERTO

Cuadro 4: Mensaje del servidor para registrar su *socket* en el directorio.

opcode (1 byte)
Valor byte de OK

Cuadro 5: Este mensaje sirve para decir al servidor que fue registrado con éxito.

2.2.2 Comunicación entre cliente y servidor

El protocolo cliente-servidor es TCP (confiable) y casi toda la comunicación que realizarán los programas de cliente y servidor, obviando el registro y peticiones con el directorio, serán mediante tuberías basadas en TCP, por lo que el propio framework estándar de Java tiene ya realizada la implementación de cómo comunicar los elementos (*sockets*, *streams* de datos por cada *socket*...).

Se usan mensajes en formato clave-valor. Son los siguientes:

- OP_NICK: Mensaje del cliente. Acepta un campo mensaje que será el nick que se usará.
- OP_NICK_OK: Mensaje del servidor en el que notifica que se acepta el nick y ya está registrado el usuario.
- OP_NICK_DUP: Mensaje del servidor en el que notifica al cliente que ya se está usando ese nick por parte de otro usuario y por tanto debe elegir otro.
- OP_GET_ROOMS: Mensaje del cliente para pedir al servidor información de todas las salas en uso, sus usuarios y la fecha del último mensaje.
- OP_ROOM_LIST: Mensaje del servidor en respuesta a OP_GET_ROOMS. Trae la información consigo.
- OP_ENTER: Mensaje del cliente junto al nombre de la sala en la que se desea entrar.
- OP_IN_ROOM: Mensaje del servidor. Indica que se ha entrado con éxito a la sala especificada en OP_ENTER.
- ~~OP_NO_ROOM~~: Mensaje del servidor que indica que no se puede acceder a la sala. Actualmente no se utiliza en este proyecto pero se mantiene por si se expande el programa para incluir privilegios y vetos a usuarios.
- OP_SEND: Mensaje del cliente acompañado del mensaje a enviar globalmente a la sala.
- OP_MSG: Mensaje del servidor que contiene un mensaje proveniente de un cliente concreto. Puede ser privado (OP_SEND_PRIV) o público (OP_SEND).
- OP_EXIT: Mensaje del cliente para poder salir de la sala.
- OP_GONE: Mensaje del servidor que indica que se ha ido un usuario de la sala.
- OP_INFO: Mensaje del servidor con la información de la sala actual.
- OP_GET_INFO: Mensaje del cliente para pedir al servidor información de la sala actual.
- OP_BROADCAST: Mensaje del servidor en difusión.
- OP_SEND_PRIV: Mensaje del cliente para enviar un mensaje privado a un cliente concreto.

3 Implementación en el código

El código es bastante intuitivo, se mantienen los comentarios que componían los *TODOs*, y se ha utilizado al máximo los recursos ya existentes, para poder simplificar todo lo posible.

Quizás hay que reconocer el posible abuso del *casting* para los mensajes, pero ha sido la manera más fácil de realizar algunas operaciones, aunque se vea un poco *spaghetti*, dentro del orden mantenido y realizado.

Por tanto, aquí solo se destacarán algunos detalles de la implementación que puedan parecer extraños o curiosos.

3.1 Enumerado serializado

Lo primero que cabe destacar, es que se ha introducido un enumerado para los opcodes de los mensajes UDP.

El código está en `./src/es/um/redes/nanoChat/directory/DirectoryOpCodes.java` y se procede a explicar algún que otro detalle:

```
// Omito los import
// Serializable implica que puedo transferir
// cada valor del enumerado en bytes
public enum DirectoryOpCodes implements Serializable {
    // Aquí especifico cada
    OK(1),
    NOSERVER(2),
    REGISTER(3),
    GETSERVER(4),
    SERVERRES(5);

    private final int value; // Valor de cada valor del enum
    private static final Map map = new HashMap<>();
    // Mapeo para relacionar valores

    DirectoryOpCodes(int value) { // Constructor para cada valor
        this.value = value;
    }

    static { // Al crear el enumerado se hace su mapeo
        for (DirectoryOpCodes opcode : DirectoryOpCodes.values()) {
            map.put(opcode.value, opcode);
        }
    }

    public byte getByteValue() {
    // Obtenemos el número del código como Byte
        return (byte) value;
    }
}
```

Listing 1: Parte del código

Aquí se ve la verdadera potencia de un enumerado en Java. Se añaden valores al enumerado, y se extraen como bytes (magia del Serializable).

Se podría también implementar para los códigos de operación de mensajes, pero sería algo más complejo al existir ya código usado para esas variables.

3.2 Tipos de mensajes creados

Se han realizado varios tipos de mensajes, a saber:

- NCBroadcastMessage: Mensaje del servidor que avisa de entradas o salidas.
- NCImmediateMessage: Mensaje en el que solo se necesite un opcode.
- NCRoomInfoMessage: Mensaje por el que el servidor devuelve información de una sala.
- NCRoomListMessage: Mensaje que devuelve varios `NCRoomInfoMessage` para la lista de salas.
- NCRoomMessage: Mensaje con un solo parámetro. Sirve para entrar en salas, cambios de nick...
- NCRoomSndRcvMessage: Mensaje para... mensajes. Especifica emisor, receptor o receptores, privacidad y mensaje en sí.

El formato de los mensajes es el siguiente:

- NCBroadcastMessage:
  `operation:<Op2Str>\n`
  `<user>:<joined|left>\n\n`
- NCImmediateMessage:
  `operation:<Op2Str>\n\n`
- NCRoomInfoMessage:
  `operation:<Op2Str>\n`
  `Room Name:<room>\n`
  `Members (<i>): <user`$_1$`>[,<user`$_{2..i}$`>]\n`
  `Last message: <time>\n\n`
  Se imprimen todos los usuarios separados por comas.
- NCRoomListMessage:
  `operation:<Op2Str>\n`
  `room name: <room`$_i$`>\n`
  `members: <user`$_1$`>[,<user`$_{2..j}$`>]\n`
  `last message: <time`$_i$`>\n`
  `\n`
  Se repiten las líneas de nombre de habitación, lista de usuarios y último mensaje por cada sala existente.

- NCRoomMessage:
  `operation:<Op2Str>\n`
  `msg: <msg>\n\n`
  Si, por ejemplo, la operación es **nick**, el contenido de *msg* será evaluado como el nick a usar.

3.2.1 Mensajes de chat (públicos y privados)

Tanto los mensajes públicos como los privados se han implementado en `NCRoomSndRcvMessage` y cuenta con los siguientes campos:

- Usuario emisor
- Mensaje
- Booleano que indica si es privado
- Usuario receptor, en caso de ser privado

¿Posible mejora? Evaluar el receptor (*nullable?*) en lugar de evaluar si hay un *flag* activo y coger el receptor en consecuencia.

3.3 Implementación de mensajes diferenciados

Para diferenciar mensajes normales de mensajes privados o mensajes del servidor en sala de chat, he decidido aprovechar la potencia de la terminal y usar códigos ANSI de color.

```
1 public static final String ANSI_BLUE = "\u001B[34m"; // Privados
2 public static final String ANSI_GREEN = "\u001B[32m"; // Servidor
3 public static final String ANSI_RESET = "\u001B[0m"; // Reset
```

Listing 2: Colores

Al principio del *string* que imprime los mensajes privados o mensajes de servidor se inserta el *string* para su respectivo color. Al final se inserta el *string* para devolver el color original de la shell.

4 Otros

4.1 Demostración de la comunicación entre dos equipos

Se incluyen algunas imágenes en la carpeta donde está localizado el documento. Se han realizado las pruebas en dos máquinas virtuales de Fedora 32 con JRE 8 de Oracle instalado en ellas (se usa Fedora en lugar de Ubuntu porque se puede instalar Java con un RPM de forma sencilla).

En ellas se puede ver el correcto envío de mensajes, capturándose en Wireshark ya que está rastreando la red virtual (estas máquinas virtuales se encuentran en KVM/QEMU, en VirtualBox no funciona este método).

Todos los OK_*.png son capturas del código 1, que corresponde al OK de UDP, viendo las tres caras (Directorio, cliente y servidor).

Se realiza una prueba de establecer el nick como *ElonMusk* en el cliente, y se captura en NickElon.png la operación enviada. Y en NickIsOkay.png se observa que efectivamente se acepta el nick.

Y en Miaus.png se ve cómo se envía un mensaje al servidor dentro de una sala concreta.

Con este par de ejemplos se pretende demostrar el correcto funcionamiento de este programa correspondiéndose con TCP y UDP.

4.2 Tests

Para poder realizar correctamente algunas funcionalidades de paso de mensajes he realizado algunos tests con tal de comprobar que se envían y reciben correctamente. Están en el directorio /tests aunque solo existe la clase MessageTests. Por supuesto, es recurrente el uso de nombres de celebridades de Silicon Valley a modo de ejemplo.

Obviando ese detalle *algo freak*, es muy útil realizar tests. Te ayudan a descubrir si el paso de mensajes está bien implementado, si la creación del mensaje y la lectura del mensaje codificado en una cadena de caracteres también lo está.

5 Conclusiones

Este proyecto es bastante útil para empezar a programar en serio Java, descubrir la manera *dura* de desarrollar un protocolo de mensajería básico, tratar conexiones TCP y UDP desde la capa de aplicación, y comprobar que realmente esto funciona y es *transparente* al usuario gracias a los mecanismos de sistema operativo y a la pila TCP/IP.

Seguramente funcione incluso entre equipos localizados en distintas redes y en distintos sistemas. Es decir, es un programa portable, con crear los .jar (ojo, siempre que usemos Oracle u OpenJDK, Amazon Coretto da problemas, que es el que trae IntelliJ por defecto. No se ha probado con GraalVM u otras máquinas virtuales) se consigue un programa que corra sobre cualquier equipo con JRE instalado. Hablando de problemas: resulta que la implementación de Coretto de las tuberías TCP da problemas con otras implementaciones de Java.

Además, se deja la puerta abierta a más modificaciones, para hacer que se parezca más a un IRC, con operadores, vetos, *kickbans*... De hecho al poder correlacionar IRC con esta práctica se ha podido llevar muy bien el flujo de la mensajería.