

Chat Multi-Cliente

Paquete NIO de java

David Gómez Pérez
Desarrollo de Servicios Telemáticos

1. Servidor Multi-Cliente

1.1. Ejecución

La ejecución de esta clase se podrá hacer desde el mismo entorno de editor del código (en mi caso IntelliJ), o compilando desde consola el archivo *Server.jar* usando el comando *java -jar Server.jar*

1.2. Variables estáticas de clase

BUFFER_SIZE: Esta variable indica el tamaño que tienen los buffers, tanto el buffer común del servidor con todos los clientes, como el tamaño de buffer del cliente.

SERVER_PORT: Puerto de conexión por el que el servidor escucha peticiones y establece las comunicaciones. En esta implementación, el valor por defecto de este campo es 6543, por lo que si se quiere cambiar del puerto del servidor, se tiene que modificar directamente esta variable.

1.3. Estructura de la clase

En ésta clase solo tenemos el método a invocar (**main**).

1.3.1. Establecimiento del servidor

Creamos los distintos canales TCP usando la clase *ServerSocketChannel* del paquete NIO, creamos un socket auxiliar que identifica al host usando el puerto *SERVER_PORT* y un buffer común para transmitir los datos (*channelBuffer*).

A partir de aquí, entramos en un bucle infinito (*while(true)*). Antes de empezar a procesar peticiones, tenemos que saber a que cliente atenderemos primero, por eso usamos el metodo *selector.select()*. Luego, asignamos al iterador *keys_iterator* un iterador sobre el conjunto de claves del selector, que iremos recorriendo mientras este tenga clientes registrados.

Luego de seleccionar la clave del usuario que vamos a atender (*keys_iterator.next()*), la eliminamos, y ahora empezamos a atender peticiones.

```
//CREAMOS LOS CANALES TCP
ServerSocketChannel server = ServerSocketChannel.open();
ServerSocket server_socket = server.socket();
//CREAMOS UN SOCKET EN LA MÁQUINA HOST, USANDO EL PUERTO 6543
SocketAddress server_address = new InetSocketAddress(SERVER_PORT);
//ENLAZAMOS LOS DOS SOCKETS
server_socket.bind(server_address);
//PONEMOS AL CANAL DE SOCKET COMO NO BLOQUEANTE
server.configureBlocking(false);
//CREAMOS SELECTOR PARA ATENDER LAS PETICIONES DE LOS CLIENTES
Selector selector = Selector.open();
//REGISTRAMOS EL SELECTOR PARA QUE INICIALMENTE SOLO PUEDA ACEPTAR PETICIONES
server.register(selector, SelectionKey.OP_ACCEPT);
//CRECIÓN DEL BUFFER
ByteBuffer serverBuffer = ByteBuffer.allocate(BUFFER_SIZE);
serverBuffer.clear();
//INICIAMOS EL BUFFER Y LA LISTA DE CLAVES A NULL
Iterator<SelectionKey> keys_iterator;
System.out.println("Servidor iniciado en el puerto " + SERVER_PORT);
```

```
while (true) {
    selector.select();
    keys_iterator = selector.selectedKeys().iterator();
    //ITERAMOS SOBRE LAS CLAVES REGISTRADAS EN EL SELECTOR
    while (keys_iterator.hasNext()) {
        SelectionKey key = (SelectionKey) keys_iterator.next();
        //ELIMINAMOS ESTA CLAVE PARA NO ATENDER DOS VECES SEGUIDAS UNA PETICIÓN DE ESTE CLIENTE
        keys_iterator.remove();
    }
}
```

1.3.2. Petición de entrada

Esta es una de las tres partes más importantes de este programa. En esta primera, el servidor se encarga de aceptar a los nuevos clientes.

Inicialmente, le asignaremos al cliente que solo puedan leer (*SelectionKey.OP_READ*) los datos que le llegan por su buffer (*channelBuffer2*), le asignaremos un tamaño inicial a este buffer de *BUFFER_SIZE* bytes y configuraremos al cliente como no bloqueante (concurrente) usando la sentencia *client.configureBlocking(false)*, donde *client* es el *SocketChannel* o socket NIO asociado al nuevo cliente.

```
//PETICIÓN DE REGISTRO EN EL SELECTOR
if (key.isAcceptable()) {
    ByteBuffer channelBuffer2;
    SocketChannel client = server.accept();
    client.configureBlocking(false);
    channelBuffer2 = ByteBuffer.allocate(BUFFER_SIZE);
    System.out.println("Entra cliente <" + client.socket().getInetAddress() + "," + client.socket().getPort() + ">");
    client.register(selector, SelectionKey.OP_READ, channelBuffer2);
}
```

1.3.3. Petición de lectura

Después de ser aceptados, esta petición es la primera en ejecutarse por todos los clientes, y es la que se encarga de leer los datos del buffer común. Antes de empezar a leer, comprobamos si el cliente asociado a la clave *key* está o no activo. Esto lo determinaremos si a la hora de leer del buffer, hemos leído un -1, luego cerramos el cliente, y en cualquier otro caso (mayor que -1), atendemos la petición de lectura correspondiente.

Una vez sabemos que el cliente está activo, procedemos a enviarle el mensaje que el servidor tiene en su buffer (*server_buffer*). Como en este caso, lo que queremos es enviar a todos los clientes este mensaje, entonces volvemos a crear un nuevo iterador sobre las claves del selector *selector_iterator*, el cuál iremos recorriendo y enviando a través de sus buffers el mensaje recibido por otro cliente (el mensaje se recibe durante la **petición de escritura**). Si el buffer del cliente sobre el que vamos a enviar datos existe (no es nulo), entonces a este cliente le pondremos las opciones tanto de escritura como de lectura (*SelectionKey.OP_WRITE* y *SelectionKey.OP_READ*).

```
//PETICIÓN DE LECTURA
} else if (key.isReadable()) {
    SocketChannel client = (SocketChannel) key.channel();
    //SE HA LEIDO EOF (EL BUFFER SE HA CERRADO), LUEGO EL CLIENTE SE HA SALIDO
    if (client.read(serverBuffer) == -1) {
        System.out.println("Sale cliente <" + client.socket().getInetAddress() + "," + client.socket().getPort() + ">");
        client.close();
    }

    //ENVIAMOS MENSAJE AL RESTO DE CLIENTES REGISTRADOS
} else {
    //PREPARAMOS BUFFER Y EL ITERADOR DE LOS USUARIOS A ENVIAR EL MENSAJE
    serverBuffer.flip();
    Iterator<SelectionKey> selector_iterator = selector.keys().iterator();
    System.out.println("> Enviando mensaje a " + (selector.keys().size() - 1) + " usuarios ...");
    SelectionKey selector_key = null;
    while (selector_iterator.hasNext()) {
        selector_key = selector_iterator.next();
        ByteBuffer buf = (ByteBuffer) selector_key.attachment();
        //SI EL BUFFER ESTÁ ABIERTO, ENVIAMOS, EN OTRO CASO, NO HACEMOS NADA
        if (buf != null) {
            buf.put(serverBuffer);
            //AL CLIENTE LE INTERESA ESCRIBIR Y LEER
            selector_key.interestOps(SelectionKey.OP_WRITE | SelectionKey.OP_READ);
            serverBuffer.rewind();
        }
    }
    //LIMPIAMOS BUFFER
    serverBuffer.clear();
}
}
```

1.3.4. Petición de escritura

En esta última condición del bucle, el servidor se encarga de escribir en su buffer común los datos que le envía un cliente para así luego enviárselos al resto de usuarios.

Para ello, tomamos el *SocketChannel* (al que llamaremos *client*) y el *ByteBuffer* (al que llamaremos *buffer*) del cliente *key* sobre el que estamos iterando. Primero reposicionamos el *buffer* usando los métodos *flip()* y *rewind()*, para luego escribir usando el método *write(buffer)* de la clase *SocketChannel*. En caso de que aún queden datos por leer, (*buffer.hasRemaining()*), hacemos sitio usando *buffer.compact()*; y en cualquier otro caso, limpiamos el buffer (*buffer.clear()*) y asignamos a este cliente la opción de solo lectura (*SelectionKey.OP_READ*).

```
//PETICIÓN DE ESCRITURA
} else if (key.isWritable()) {
    SocketChannel client = (SocketChannel) key.channel();
    ByteBuffer buffer = (ByteBuffer) key.attachment();
    buffer.flip();
    buffer.rewind();
    client.write(buffer);
    if (buffer.hasRemaining()) {
        buffer.compact();
    } else {
        buffer.clear();
        key.interestOps(SelectionKey.OP_READ);
    }
}
```

1.4. Excepciones

En caso de que se lance alguna excepción a la hora de tratar algún cliente, lo eliminaremos del chat usando *key.cancel()* y *key.channel().close()*. El primer método se utiliza para cancelar la clave dentro del selector del servidor, mientras que el segundo método simplemente cierra el socket de conexión de este cliente con el servidor.

```
//EN CASO DE UNA EXCEPCIÓN, ELIMINAMOS ESTA CLAVE DEL SELECTOR
} catch (Exception e) {
    SocketChannel client = (SocketChannel) key.channel();
    System.out.println("Sale cliente <" + client.socket().getInetAddress() + "," + client.socket().getPort() + ">");
    key.cancel();
    try {
        key.channel().close();
    } catch (Exception ce) {
        System.out.println(ce.getLocalizedMessage());
    }
}
```

2. Cliente

2.1. Ejecución

Para la ejecución del cliente, se proporciona una interfaz gráfica llamada *Client.jar*. Esta GUI ha sido generada utilizando el *GUI Form* de IntelliJ, por lo que no estoy seguro de que se pueda ejecutar la clase *ClientGui.java* funcione en otros editores de código.

2.2. Manual de usuario

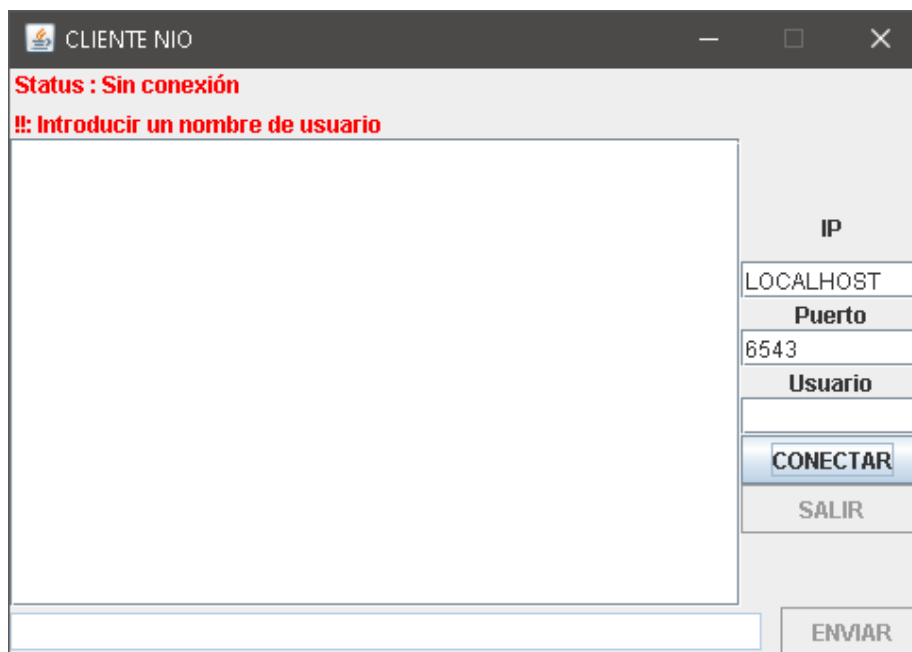
Al iniciar la aplicación del usuario, se mostrará la siguiente ventana:

The screenshot shows a Java Swing window titled "CLIENTE NIO". At the top, a status bar displays "Status : Sin conexión" in red text. The main content area is a large, empty rectangular text field. To the right of this field, there is a vertical stack of controls: an "IP" label above a text box containing "LOCALHOST", a "Puerto" label above a text box containing "8543", and an "Usuario" label above an empty text box. Below these input fields are three buttons: "CONECTAR" (highlighted in blue), "SALIR", and "ENVIAR".

2.2.1. Conexión con el servidor

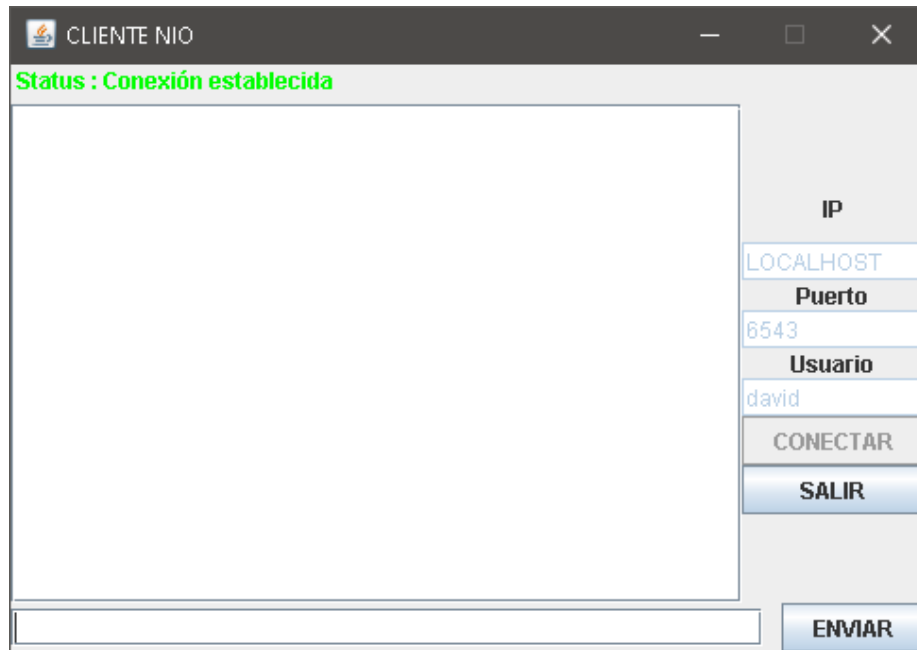
Inicialmente, los campos de IP y Puerto estarán predefinidos, pero estos se podrán cambiar antes de iniciar la conexión. También tenemos un campo usuario (que se encuentra vacío) y un botón conectar, que se pulsará cuando estemos listos para entrar en el chat. El resto de componentes se encuentran deshabilitados hasta que la conexión se inicie.

En caso de que pulsemos *CONECTAR* y no hallamos puesto ningún nombre de usuario, se indicará un error justo debajo de la etiqueta *Status*. Además, en caso de que utilicemos un puerto incorrecto o una IP que no sea válida, también se nos mostrará un mensaje de error.



The screenshot shows a window titled "CLIENTE NIO" with a standard Windows title bar. The main area has a light gray background. At the top left, it says "Status : Sin conexión" in red. Below that, in red, is "!!: Introducir un nombre de usuario". To the right of this text is a large white text area. Below the text area is a horizontal input field. To the right of the text area are several controls: a label "IP" above a text box containing "LOCALHOST", a label "Puerto" above a text box containing "6543", a label "Usuario" above an empty text box, a blue "CONECTAR" button, a gray "SALIR" button, and a gray "ENVIAR" button at the bottom right.

Una vez introducidos los parámetros correctamente, debemos de asegurarnos de que el servidor está en marcha, ya que sino nos saltará una excepción y nos dirá que tanto la IP como el puerto no son válidos. Una vez conectado al servidor, la ventana quedará de la siguiente manera:



Como vemos, la etiqueta *Status* ha cambiado, y ahora, los botones *SALIR* y *ENVIAR* se han habilitado, al igual que el área de texto. Además, los campos mencionados anteriormente y el botón *CONECTAR* se han deshabilitado para evitar cambiar estos parámetros durante la ejecución del cliente.

2.2.2. Enviar mensaje

Para enviar un mensaje, solamente tenemos que escribir en el área de texto inferior lo que queramos mandar, y luego pulsar el botón *ENVIAR*. Este mensaje llegará a todos los usuarios conectados al servidor.

2.2.3. Terminar conexión

Para finalizar la conexión con el servidor, tenemos dos opciones. La primera es pulsando el botón *SALIR*, donde nos volverá a aparecer la ventana inicial (el área de texto del chat no se limpia). La segunda opción es terminando el programa (cerrar ventana), lo cuál causará una excepción en el servidor, pero que está controlada para no parar la ejecución.

2.3. Métodos de clase

public ClientGui() Este es el constructor de la clase cliente. En él, se establecen las configuraciones iniciales de los distintos componentes de la interfaz gráfica, como por ejemplo, los valores iniciales de los campos *IP* y *Puerto* o de la etiqueta *Status*. También habilitamos y deshabilitamos los componentes

correspondientes que ya hemos mencionado anteriormente (botones, áreas de texto, ...).

Además, configuramos los eventos de los tres botones (2.3.1, 2.3.2 y 2.3.3).

public void receiveMess() Este método se dedica a extraer los mensajes del buffer del cliente, leerlos y mostrarlos en el área de texto correspondiente. Para ello reservamos *BUFFER_SIZE* bytes de memoria, limpiamos el buffer usando *buffer.clear()* y leemos *socket.read(buffer)*. Una vez leído, guardamos en la variable *mensaje* el texto recibido y lo mostramos en *textPanel* junto al resto de mensajes.

```
public void receiveMess() {
    while (true) {
        try {
            buffer = ByteBuffer.allocate(BUFFER_SIZE);
            buffer.clear();
            socket.read(buffer);
            String mensaje = new String(buffer.array()).trim();
            this.textPanel.append(mensaje + System.lineSeparator());
        } catch (Exception ie) {
            //connected = false;
        }
    }
}
```

public static void main (String [] args) En este método se crea la ventana principal, se inicia el cliente y se invoca al método anterior para estar continuamente leyendo del buffer.

```
public static void main(String[] args) {
    ClientGui cg = new ClientGui();
    JFrame window = new JFrame( title: "CLIENTE NIO");
    window.setBounds( x: 100, y: 100, width: 500, height: 350);
    window.setContentPane(cg.mainPanel);
    window.setResizable(false);
    window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    window.setVisible(true);
    cg.receiveMess();
}
```

2.3.1. Configuración del botón *CONECTAR*

Si hacemos click izquierdo *MouseEvent.BUTTON1* sobre el botón, leemos los campos de inicio (dirección IP, puerto y nombre de usuario). Comprobamos que el nombre de usuario no esté vacío y no sea **null** (en este caso, mostramos mensaje de error). En el caso de que todo sea correcto, abrimos un nuevo socket, actualizamos la etiqueta *Status*, habilitamos el campo de texto y los botones de *ENVIAR* y *SALIR* y, por último, deshabilitamos los campos *IP*, *Puerto* y *Usuario* así como el botón conectar. Además, la etiqueta de errores le ponemos un texto vacío para actualizar los errores.

En caso de que ocurra alguna excepción durante la ejecución de este evento, mostramos el mensaje *!!: Introducir dirección IP y puerto de conexión válidos* en la etiqueta de error, ya que los únicos errores que podamos cometer en caso de que esta se produzca es que el puerto o la dirección IP no sean correctas.

```
connect_button.addMouseListener(new MouseAdapter() {
    @Override
    public void mouseClicked(MouseEvent e) {
        if (e.getButton() == MouseEvent.BUTTON1) {
            try {
                server_port = Integer.parseInt(port_field.getText());
                server_IP = ip_field.getText();
                username = user_field.getText();
                if (username == null || username.equals("")) {
                    not_label.setForeground(Color.RED);
                    not_label.setText(" !!: Introducir un nombre de usuario");
                } else {
                    socket = SocketChannel.open(new InetSocketAddress(server_IP, server_port));
                    //buffer = ByteBuffer.allocate(BUFFER_SIZE);
                    connected = true;
                    status_label.setForeground(Color.GREEN);
                    status_label.setText(" Status : Conexión establecida");
                    send_button.setEnabled(true);
                    text_enter.setEnabled(true);
                    connect_button.setEnabled(false);
                    exit_button.setEnabled(true);
                    ip_field.setEnabled(false);
                    port_field.setEnabled(false);
                    user_field.setEnabled(false);
                    not_label.setText("");
                }
            } catch (Exception ex) {
                not_label.setForeground(Color.RED);
                not_label.setText(" !!: Introducir dirección IP y puerto de conexión válidos");
            }
        }
    }
});
```

2.3.2. Configuración del botón *SALIR*

Este evento es similar al anterior. Para que se active, tenemos que hacer click izquierdo sobre el botón. Lo primero que hacemos, es cerrar el socket asociado al cliente y asignar **null** al buffer de este, para que cuando el servidor comprueba si este cliente está o no activo, lo elimine de su selector.

Además, deshabilitamos el botón de *ENVIAR* y el área de texto, así como el botón *SALIR*. Luego, volvemos a habilitar los campos *IP*, *Usuario* y *Puerto* y el botón de *CONECTAR*, para poder volver a iniciar sesión. Por último, actualizamos la etiqueta *Status*.

```
exit_button.addMouseListener(new MouseAdapter() {
    @Override
    public void mouseClicked(MouseEvent e) {
        if (e.getButton() == MouseEvent.BUTTON1) {
            //cerramos cliente
            try {
                socket.close();
                buffer = null;
                send_button.setEnabled(false);
                text_enter.setEnabled(false);
                connect_button.setEnabled(true);
                exit_button.setEnabled(false);
                ip_field.setEnabled(true);
                port_field.setEnabled(true);
                user_field.setEnabled(true);
                not_label.setText("");
                status_label.setForeground(Color.RED);
                status_label.setText("Status : Sin conexión");
            } catch (Exception ex) {
            }
        }
    }
});
```

2.3.3. Configuración del botón *ENVIAR*

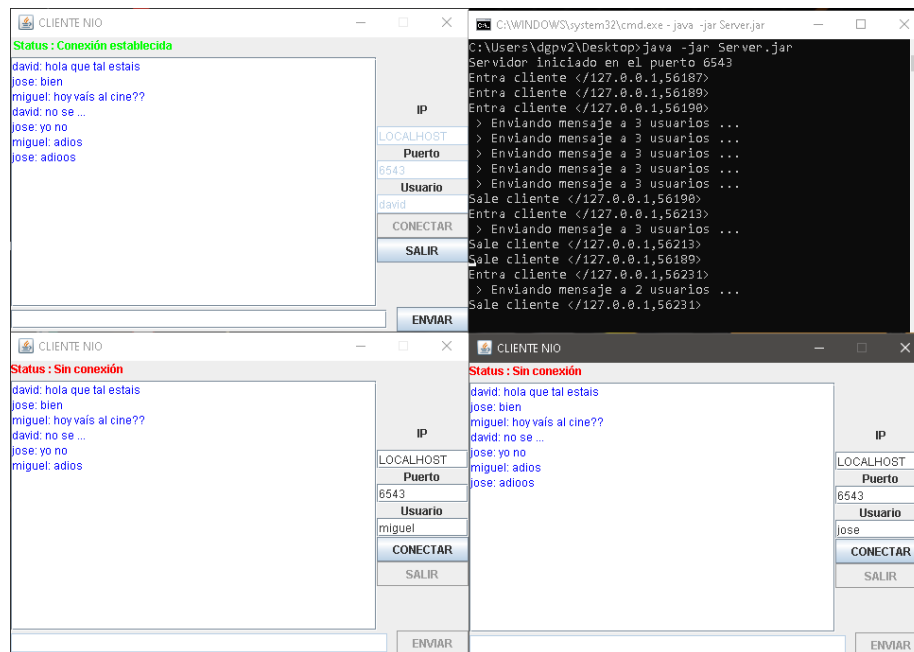
Para poder acceder a este evento, primero tendremos que estar conectados al servidor. Una vez dentro, este botón simplemente guarda en una variable el contenido de la caja de texto (mensaje a enviar). A este mensaje, se le antepone el nombre de usuario (para visualizar el origen del mensaje en el panel de texto).

Una vez tenemos el mensaje junto al nombre del usuario (*msg*), simplemente actualizamos el buffer del cliente con el contenido a enviar usando el método *ByteBuffer.wrap(msg.getBytes())*, luego escribimos en el socket usando *socket.write(buffer)* y reseteamos la caja de texto.

```
send_button.addMouseListener(new MouseAdapter() {
    @Override
    public void mouseClicked(MouseEvent e) {
        if (e.getButton() == MouseEvent.BUTTON1) {
            //enviamos mensaje
            String msg = username + ": " + text_enter.getText();
            try {
                buffer = ByteBuffer.wrap(msg.getBytes());
                //buffer.clear();
                socket.write(buffer);
                text_enter.setText("");
            } catch (IOException ioException) {
                ioException.printStackTrace();
            }
        }
    }
});
```

3. Ejemplo de ejecución

Ejemplo en la ejecución de tres clientes junto a la salida por consola del servidor:



4. Crítica

4.1. Defectos

1. No se ha utilizado ningún patrón a la hora de diseñar el cliente (ej. Modelo-Vista-Controlador), por lo que si queremos editar cualquier cosa será un poco tedioso.
2. El puerto de conexión al servidor es siempre fijo.
3. Las excepciones capturadas en el cliente son muy generales (todas se capturan usando *Exception*).
4. De vez en cuando, los botones no siempre responden a la primera (pero sí funcionan).

4.2. Puntos a favor

1. Cuando desconectemos (ya sea forzando la salida o no), el servidor se encargará automáticamente de eliminar a este cliente (no se interrumpe su ejecución).

- 2.** La interfaz del usuario es bastante simple de utilizar.
- 3.** Durante la ejecución del servidor, se puede ver las acciones que este realiza, como la aceptación de clientes o el envío de mensajes a estos.