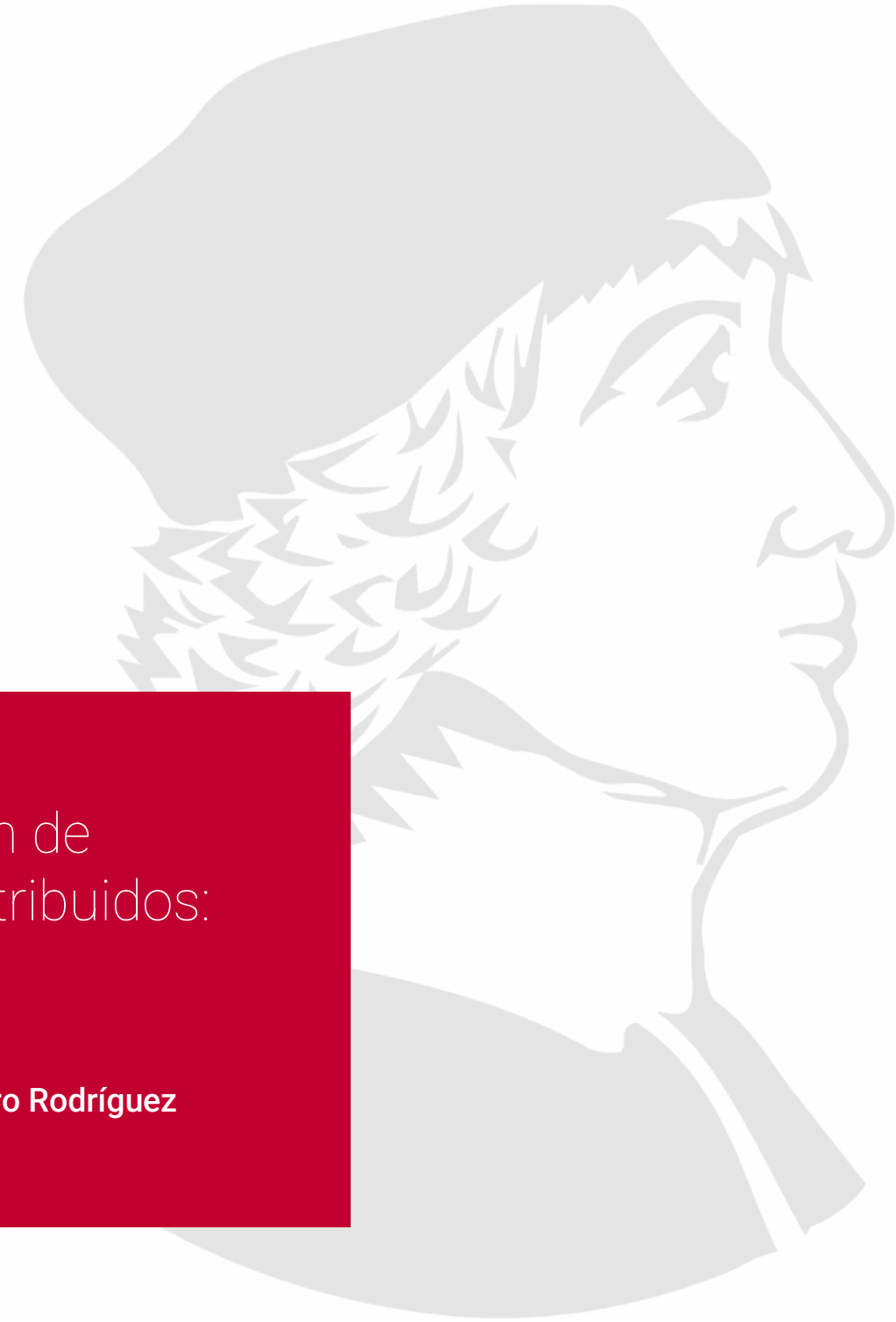


Programación de Sistemas Distribuidos: Practica 5

Chenlei Ye
Sara Marcos Cornejo
Jose Francisco Romero Rodríguez

UNIVERSIDAD
NEBRIJA

ÍNDICE

Introducción

En el lenguaje de programación Java, los sockets proporcionan una forma de establecer una comunicación bidireccional entre un cliente y un servidor a través de una red. Los sockets en Java se basan en el modelo Cliente-Servidor, donde el servidor espera conexiones entrantes y el cliente establece una conexión con el servidor.

A continuación, se describen los pasos generales para utilizar sockets en Java:

1. Configuración del servidor: El servidor debe crear un objeto ServerSocket que escuche conexiones en un puerto específico. Una vez que se establece una conexión con un cliente, se crea un objeto Socket que representa la conexión entre el cliente y el servidor. El servidor puede utilizar métodos en el objeto Socket para enviar y recibir datos.

2. Configuración del cliente: El cliente debe crear un objeto Socket y proporcionar la dirección IP y el puerto del servidor al que desea conectarse. Al crear el objeto Socket, se establece una conexión con el servidor. El cliente también puede utilizar métodos en el objeto Socket para enviar y recibir datos.

3. Envío y recepción de datos: Tanto el cliente como el servidor pueden utilizar los flujos de entrada y salida asociados con el objeto Socket para enviar y recibir datos. El uso de InputStream y OutputStream permite el intercambio de información entre el cliente y el servidor. Por ejemplo, el servidor puede usar getInputStream() para recibir datos del cliente y getOutputStream() para enviar datos al cliente.

4. Cierre de la conexión: Después de que la comunicación entre el cliente y el servidor se haya completado, se debe cerrar adecuadamente la conexión para liberar recursos. Tanto el cliente como el servidor pueden llamar al método close() en el objeto Socket para cerrar la conexión.

Es importante tener en cuenta que el uso de sockets en Java implica la gestión adecuada de excepciones, ya que pueden ocurrir errores durante la conexión y la transmisión de datos. También se debe tener cuidado al diseñar el protocolo de comunicación entre el cliente y el servidor para asegurarse de que ambas partes estén sincronizadas en la interpretación y el formato de los datos intercambiados.

Java proporciona una API muy completa para trabajar con sockets, incluyendo las clases ServerSocket y Socket en el paquete java.net. Estas clases permiten una implementación flexible y escalable de la comunicación entre el cliente y el servidor en aplicaciones de red.

Práctica

Arquitectura del Sistema

La arquitectura del sistema del chat implementado con sockets en Java se basa en el modelo Cliente-Servidor. En este modelo, hay un servidor centralizado que espera conexiones entrantes de múltiples clientes y gestiona la comunicación entre ellos.

El sistema consta de dos componentes principales: el servidor y los clientes.

Servidor:
El servidor actúa como el punto central de comunicación y coordina las interacciones entre los clientes. Su función principal es escuchar conexiones entrantes de los clientes y gestionar la comunicación bidireccional entre ellos.
El servidor utiliza un objeto ServerSocket para escuchar en un puerto específico y esperar las conexiones entrantes de los clientes. Cuando se establece una conexión con un cliente, se crea un objeto Socket que representa la conexión entre el cliente y el servidor.

Para manejar las comunicaciones con cada cliente, el servidor crea un objeto ClientHandler para cada conexión entrante. El ClientHandler es un hilo separado que se encarga de recibir y enviar mensajes específicos de un cliente en particular. Esto permite que el servidor atienda a múltiples clientes simultáneamente.

Dentro del ClientHandler, se obtienen los flujos de entrada y salida asociados con el objeto Socket del cliente. Esto permite la recepción y el envío de mensajes entre el cliente y el servidor. Además, el ClientHandler también se encarga de gestionar el ingreso del nombre de usuario por parte del cliente, enviar mensajes de notificación a todos los clientes cuando se unen o abandonan el chat, y realizar la transmisión de mensajes a todos los clientes conectados.

El servidor mantiene una lista de todos los ClientHandlers activos, lo que le permite enviar mensajes a todos los clientes conectados, excepto al que envió el mensaje originalmente.

Clientes:
Los clientes son las entidades que se conectan al servidor y participan en la comunicación del chat. Cada cliente tiene su propia instancia del programa de cliente que se ejecuta de forma independiente.
Cuando un cliente se inicia, se establece una conexión con el servidor utilizando un objeto Socket y proporcionando la dirección IP y el puerto del servidor al que se desea conectar. Una vez establecida la conexión, se crean flujos de entrada y salida para recibir y enviar datos al servidor.

El cliente utiliza un hilo separado para recibir mensajes asincrónicamente del servidor. Este hilo está constantemente leyendo el flujo de entrada y muestra los mensajes recibidos en la consola del cliente.

El cliente también proporciona una interfaz para que el usuario ingrese mensajes y los envíe al servidor a través del flujo de salida. Estos mensajes se transmiten al servidor, que a su vez los distribuye a todos los clientes conectados.

Además, el cliente permite al usuario ingresar un nombre de usuario que se envía al servidor para su identificación en el chat. El servidor utiliza este nombre de usuario para mostrar quién envía cada mensaje.

En resumen, la arquitectura del sistema del chat implementado con sockets en Java sigue el modelo Cliente-Servidor. El servidor actúa como el punto central de comunicación y coordina las interacciones entre múltiples clientes. Los clientes se conectan al servidor, envían y reciben mensajes, y participan en la comunicación bidireccional del chat.

La **gestión de errores y excepciones** es una parte crítica en el desarrollo de aplicaciones que utilizan sockets en Java. Durante la comunicación entre el cliente y el servidor, pueden ocurrir diversos errores y situaciones excepcionales que deben ser manejados adecuadamente para garantizar un funcionamiento robusto y confiable del sistema. A continuación, se explican los principales aspectos de la gestión de errores y excepciones en el contexto de un chat implementado con sockets:

1- **Excepciones de conexión:**
   Al intentar establecer una conexión, pueden producirse excepciones si el servidor no está disponible o si hay problemas de red. Por ejemplo, el servidor puede no estar en ejecución, el puerto puede estar ocupado o puede haber errores en la configuración de red.

   Para manejar estas excepciones, se utiliza un bloque try-catch alrededor del código que establece la conexión. Si se produce una excepción, se captura y se muestra un mensaje de error adecuado al usuario.

2- **Excepciones de lectura y escritura:**
   Durante la transmisión de datos entre el cliente y el servidor, pueden ocurrir excepciones relacionadas con la lectura y escritura en los flujos de entrada y salida.

   Por ejemplo, puede ocurrir una excepción si se pierde la conexión repentinamente, si hay interrupciones en la red o si se supera el tiempo de espera para la recepción de datos.

   Para manejar estas excepciones, se utilizan bloques try-catch alrededor del código de lectura y escritura en los flujos. En caso de excepción, se puede mostrar un mensaje de error, cerrar la conexión con el cliente afectado y realizar cualquier otra acción apropiada, como eliminar al cliente de la lista de clientes conectados.

3- **Excepciones de cierre de conexión:**
   Al finalizar la comunicación entre el cliente y el servidor, se deben cerrar adecuadamente las conexiones y liberar los recursos utilizados. Sin embargo, pueden ocurrir excepciones al cerrar los sockets y los flujos asociados.

   Para manejar estas excepciones, se utilizan bloques try-catch alrededor del código de cierre de conexión. Se capturan las excepciones y se realiza una gestión adecuada, como mostrar mensajes de error y liberar los recursos.

Es fundamental asegurarse de que el sistema continúe funcionando correctamente a pesar de las excepciones. Esto implica tomar las medidas adecuadas para recuperarse de los errores y mantener la integridad y la continuidad de la comunicación entre el cliente y el servidor.

Código servidor:

```java
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.List;

public class ChatServer {
    private static List<ClientHandler> clients = new ArrayList<>();

    Run | Debug
    public static void main(String[] args) {
        try {
            try (ServerSocket serverSocket = new ServerSocket(port:12345)) {
                System.out.println(x:"Servidor iniciado. Esperando conexiones...");

                while (true) {
                    Socket socket = serverSocket.accept();
                    System.out.println("Nuevo cliente conectado: " + socket.getInetAddress().getHostAddress());

                    ClientHandler clientHandler = new ClientHandler(socket);
                    clients.add(clientHandler);
                    clientHandler.start();
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    static class ClientHandler extends Thread {
        private Socket socket;
        private InputStream input;
        private OutputStream output;
        private String username;
        private boolean connected = true;

        public ClientHandler(Socket socket) {
            this.socket = socket;
        }

        @Override
        public void run() {
            try {
                input = socket.getInputStream();
                output = socket.getOutputStream();

                // Solicitar nombre de usuario solo una vez
                requestUsername();

                byte[] buffer = new byte[1024];
                int bytesRead;
                while ((bytesRead = input.read(buffer)) != -1) {
                    String message = new String(buffer, offset:0, bytesRead);
                    System.out.println(username + " dice: " + message);

                    // Enviar mensaje a todos los clientes conectados
                    broadcastMessage(username + " dice: " + message);

                    // Verificar si el cliente desea abandonar el chat
                    if (message.trim().equalsIgnoreCase(anotherString:"/salir")) {
                        connected = false;
                        break;
                    }
                }

                // Cliente desconectado
                System.out.println(username + " se ha desconectado");

                // Eliminar el cliente de la lista de clientes
                clients.remove(this);

                // Notificar a los demás clientes sobre la desconexión
                broadcastMessage(username + " se ha desconectado");

                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        private void requestUsername() throws IOException {

            byte[] buffer = new byte[1024];
            int bytesRead = input.read(buffer);
            username = new String(buffer, offset:0, bytesRead).trim();

            System.out.println("Nuevo usuario: " + username);
            broadcastMessage(username + " se ha unido al chat");
        }

        private void broadcastMessage(String message) throws IOException {
            for (ClientHandler client : clients) {
                if (client != this) {
                    client.output.write(message.getBytes());
                }
            }
        }
    }
}
```

Explicación Servidor:

Al ejecutar el programa del servidor, se crea un objeto ServerSocket que escucha en el puerto 12345 para esperar conexiones entrantes de clientes.

Dentro de un bucle infinito, el servidor acepta nuevas conexiones utilizando el método accept() del objeto ServerSocket. Una vez que se establece una conexión con un cliente, se crea un objeto Socket y se muestra un mensaje indicando la dirección IP del cliente que se ha conectado.

Para cada cliente conectado, se crea un objeto ClientHandler, que es una clase interna que extiende Thread. El objeto ClientHandler se encarga de manejar la comunicación con un cliente específico.

Dentro del método run() del ClientHandler, se obtienen los flujos de entrada y salida del Socket. Luego, se solicita al cliente que ingrese un nombre de usuario, y se envía un mensaje de notificación a todos los clientes conectados para informar que un nuevo usuario se ha unido al chat.

A continuación, se inicia un bucle en el que se lee continuamente el flujo de entrada para recibir mensajes del cliente. Cada mensaje recibido se muestra en la consola del servidor y se envía a todos los clientes conectados utilizando el método broadcastMessage().

Si el cliente envía el comando "/salir", se cierra la conexión con ese cliente, se elimina el ClientHandler correspondiente y se envía un mensaje de notificación a todos los clientes restantes.

El código del servidor también implementa un método broadcastMessage() que envía un mensaje a todos los clientes conectados, excepto al que lo ha enviado.

Código cliente:

```java
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.util.Scanner;

public class ChatClient {
    Run | Debug
    public static void main(String[] args) {
        try {
            try (Socket socket = new Socket(host:"localhost", port:12345)) {
                System.out.println(x:"Conectado al servidor");

                InputStream input = socket.getInputStream();
                OutputStream output = socket.getOutputStream();

                try (Scanner scanner = new Scanner(System.in)) {
                    Thread receiveThread = new Thread(() -> {
                        try {
                            byte[] buffer = new byte[1024];
                            int bytesRead;
                            while ((bytesRead = input.read(buffer)) != -1) {
                                String message = new String(buffer, offset:0, bytesRead);
                                System.out.println(message);
                            }
                        } catch (IOException e) {
                            System.out.println(x:"El servidor se ha desconectado");
                            System.exit(status:0);
                        }
                    });
                    receiveThread.start();

                    System.out.print(s:"Ingresa tu nombre de usuario: ");
                    String username = scanner.nextLine();

                    // Enviar el nombre de usuario al servidor
                    output.write(username.getBytes());

                    System.out.print(s:"Escribe tus mensajes: \n");

                    while (true) {
                        String message = scanner.nextLine();

                        // Enviar mensaje al servidor
                        output.write(message.getBytes());

                        // Verificar si el cliente desea abandonar el chat
                        if (message.equalsIgnoreCase(anotherString:"/salir")) {
                            System.out.println(x:"Abandonaste el chat");
                            System.exit(status:0);
                        }
                    }
                }
            }
        } catch (IOException e) {
            System.out.println(x:"No se pudo conectar al servidor");
        }
    }
}
```

Explicación Cliente:

El código que has proporcionado implementa un cliente de chat que se conecta a un servidor en la dirección "localhost" (es decir, la misma máquina) y en el puerto 12345.

El cliente establece una conexión con el servidor utilizando un objeto Socket. Una vez establecida la conexión, crea flujos de entrada y salida (InputStream y OutputStream) para recibir y enviar datos al servidor.

El cliente utiliza un Scanner para leer la entrada del usuario desde la consola. Luego, inicia un hilo (Thread) para recibir mensajes del servidor de forma asincrónica. Cada vez que se recibe un mensaje del servidor, se muestra en la consola del cliente.

El cliente solicita al usuario que ingrese un nombre de usuario y lo envía al servidor utilizando el flujo de salida.

A continuación, se inicia un bucle infinito en el que el cliente lee los mensajes ingresados por el usuario desde la consola. Cada mensaje se envía al servidor utilizando el flujo de salida.
El cliente también verifica si el usuario ha ingresado el comando "/salir" para abandonar el chat. En ese caso, se muestra un mensaje y el programa cliente se cierra.

Si ocurre alguna excepción durante la conexión con el servidor, se maneja y se muestra un mensaje de error.

Compilación:
En primer lugar se compila tanto el servidor como el cliente con el siguiente comando:

javac ChatServer.java ChatClient.java

Ejecución:

La ejecución en la terminal del servidor y los clientes implica seguir los pasos necesarios para iniciar y interactuar con las aplicaciones.

Ejecución del servidor:

- En una nueva terminal, se compila y se ejecuta el código del servidor utilizando el compilador de Java y el comando java seguido del nombre de la clase principal. (java ChatServer)
- Una vez que el servidor se inicia, muestra un mensaje indicando que está iniciado y esperando conexiones entrantes.
- A medida que los clientes se conectan, se mostrarán mensajes en la terminal indicando que se ha establecido una nueva conexión y se mostrará la dirección IP del cliente.
- El servidor estará en espera de mensajes de los clientes y los mostrará en la terminal a medida que los reciba.
- Si un cliente se desconecta, se mostrará un mensaje en la terminal indicando que el cliente ha abandonado el chat.

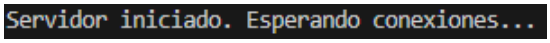
Ejecución de los clientes:

- En terminales separadas, se compila y se ejecuta el código de los clientes de manera similar a como se hizo con el servidor. (java ChatClient)
- Cada cliente se conectará al servidor utilizando la dirección IP y el puerto adecuados.
- Una vez que el cliente se conecta, se le solicitará al usuario que ingrese un nombre de usuario en la terminal. El cliente enviará este nombre de usuario al servidor.
- Después de ingresar el nombre de usuario, el cliente podrá comenzar a escribir mensajes en la terminal.

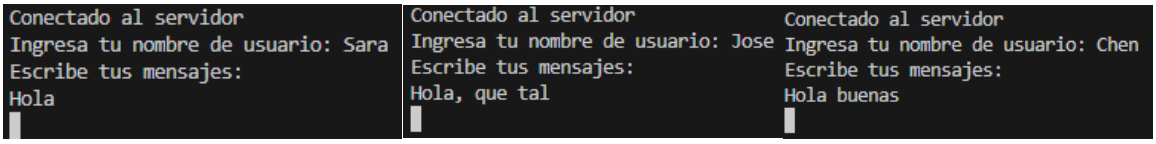
Cada mensaje que el cliente escriba será enviado al servidor y se mostrará en la terminal del servidor junto con el nombre de usuario del cliente que lo envió.
El cliente también recibirá mensajes del servidor y los mostrará en su propia terminal.
Si el cliente desea abandonar el chat, puede ingresar el comando "/salir" en la terminal. En ese caso, se mostrará un mensaje indicando que el cliente ha abandonado el chat y la aplicación cliente se cerrará.

Terminal

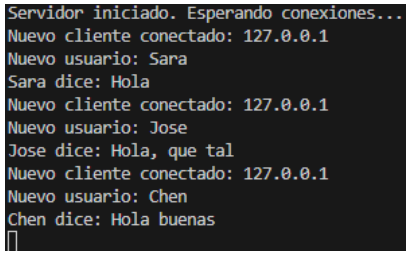
Al iniciar el servidor espera a que un usuario se conecte.



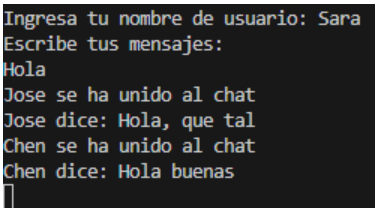
Se inician 3 clientes, Sara, Jose y Chen que escriben su respectivo mensaje.



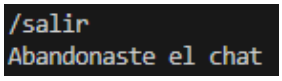
El servidor tiene la capacidad de leer todos los mensajes que se mandan de los clientes.



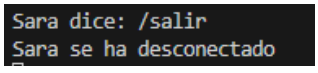
Y cada cliente lee al resto de clientes.



Y desde cualquier cliente puede salir del chat. (introduciendo el mensaje “/salir”)



Y por último al servidor le llega la información de que Sara se ha desconectado.



Conclusión

En esta práctica, hemos explorado la implementación de un chat utilizando sockets en Java. Los sockets proporcionan una forma eficiente y flexible de establecer comunicaciones entre un cliente y un servidor a través de una red.

En primer lugar, hemos desarrollado un servidor de chat utilizando la clase ServerSocket de Java. Esta clase nos permite crear un socket de servidor que acepta conexiones entrantes de clientes. Para cada cliente conectado, hemos creado un ClientHandler, que es un hilo separado encargado de manejar la comunicación con ese cliente específico.

Dentro del ClientHandler, hemos utilizado flujos de entrada y salida (InputStream y OutputStream) para recibir y enviar mensajes al cliente. Cada mensaje recibido se muestra en la consola del servidor y se envía a todos los clientes conectados utilizando el método broadcastMessage(). Además, hemos implementado la funcionalidad de solicitar al cliente un nombre de usuario y notificar a todos los clientes cuando un nuevo usuario se une al chat o cuando un usuario se desconecta.

Esta implementación de chat con sockets en Java nos ha permitido lograr una comunicación bidireccional en tiempo real entre el servidor y los clientes. Los sockets en Java ofrecen una amplia gama de posibilidades para desarrollar aplicaciones de red, y su flexibilidad y facilidad de uso los convierten en una opción popular para implementar sistemas de chat y otras aplicaciones en tiempo real.

Bibliografía

https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/net/ServerSocket.html
https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/net/Socket.html
https://www.tutorialspoint.com/java/java_networking.htm
https://www.baeldung.com/a-guide-to-java-sockets
https://github.com/topics/java-socket