Imagen que contiene Icono

El contenido generado por IA puede ser incorrecto.Dibujo en blanco y negro

El contenido generado por IA puede ser incorrecto.

Instituto Politécnico Nacional

Escuela Superior de Cómputo

Ingeniería en Sistemas Computacionales

Aplicaciones para comunicaciones en red

Academia “Sistemas Distribuidos”

Plan 2020

Práctica 23:

“**Sincronización de hilos**”

2015090269

González González Armando Omar

Profesor: Ojeda Santillan Rodrigo

Contenido

[Objetivo 3](#_Toc201736495)

[Introducción 3](#_Toc201736496)

[Desarrollo 4](#_Toc201736497)

[3.1 Implementación del Servidor (Server.java) 4](#_Toc201736498)

[3.2 Manejo de Clientes con Hilos (ClientHandler) 4](#_Toc201736499)

[3.3 Implementación del Cliente (Client.java) 5](#_Toc201736500)

[Conclusión 9](#_Toc201736501)

[Pregunta 9](#_Toc201736502)

[¿Cómo podrían generar un negocio a través de lo visto en la práctica? 9](#_Toc201736503)

[Bibliografía 9](#_Toc201736504)

## Objetivo

Desarrollar una aplicación de chat multicliente en Java que demuestre la importancia de la sincronización de hilos. El objetivo es gestionar múltiples conexiones de clientes de forma concurrente, asegurando que el acceso al recurso compartido (la lista de clientes conectados) se realice de manera segura para evitar condiciones de carrera y garantizar que todos los mensajes se distribuyan correctamente a todos los participantes.

## Introducción

La sincronización multihilo es una técnica esencial en la programación concurrente que se encarga de controlar el acceso de múltiples hilos a recursos compartidos. Cuando varios hilos intentan modificar un mismo recurso, como una variable o una colección de datos, pueden surgir inconsistencias o condiciones de carrera si el acceso no es gestionado adecuadamente. La sincronización previene estos problemas al asegurar que solo un hilo pueda acceder a una sección crítica del código a la vez, bloqueando a los demás hilos hasta que el recurso esté disponible.

Los componentes clave de la sincronización incluyen:

* **Secciones críticas:** Partes del código que acceden a recursos compartidos y que requieren protección para ser ejecutadas por un solo hilo a la vez.
* **Bloqueos (Locks):** Mecanismos que un hilo utiliza para "cerrar" una sección crítica mientras la está utilizando. En Java, esto se logra comúnmente con el bloque synchronized.
* **Monitores:** Mecanismos de alto nivel que garantizan que solo un hilo pueda ejecutar una sección de código. En Java, cada objeto posee un monitor implícito que es utilizado por los bloques synchronized.

Esta práctica utiliza estos conceptos para construir un servidor de chat donde múltiples hilos (uno por cliente) deben agregar su información a una lista compartida y leer de ella para retransmitir mensajes.

## Desarrollo

Para la práctica se desarrolló una aplicación de chat cliente-servidor en Java. El servidor es capaz de aceptar múltiples conexiones y retransmitir los mensajes a todos los clientes conectados de forma sincronizada.

### **3.1 Implementación del Servidor (**Server.java**)**

El servidor se inicia y espera conexiones en un puerto específico (5555). Utiliza un ServerSocket que, dentro de un bucle infinito, espera a que un cliente se conecte mediante el método serverSocket.accept(). Por cada cliente que se conecta, se crea e inicia un nuevo hilo de la clase ClientHandler, lo que permite al servidor manejar múltiples clientes de forma concurrente sin bloquearse.

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.net.ServerSocket;

import java.net.Socket;

import java.util.ArrayList;

public class Server {

public static void main(String[] args) {

int PORT = 5555;

System.out.println("Servidor de chat iniciado...");

try (ServerSocket serverSocket = new ServerSocket(PORT)) {

while (true) {

new ClientHandler(serverSocket.accept()).start();

}

} catch (IOException e) {

e.printStackTrace();

}

}

...

}

### **3.2 Manejo de Clientes con Hilos (**ClientHandler**)**

ClientHandler es una clase interna que extiende de Thread y maneja la comunicación con un solo cliente en un hilo independiente. Su método run() es donde ocurre la lógica principal:

1. Se establecen los flujos de entrada (BufferedReader) y salida (PrintWriter) usando el socket del cliente.
2. Se añade el PrintWriter del cliente a una lista compartida clientWriters. Esta operación se realiza dentro de un bloque synchronized (clientWriters) para evitar que varios hilos modifiquen la lista al mismo tiempo, previniendo así una condición de carrera.
3. El hilo entra en un bucle para leer los mensajes entrantes del cliente. Cada mensaje recibido se retransmite a todos los clientes de la lista. Esta retransmisión también ocurre dentro de un bloque synchronized para garantizar una lectura consistente de la lista.

private static class ClientHandler extends Thread {

private Socket socket;

private PrintWriter out;

private BufferedReader in;

private ArrayList<PrintWriter> clientWriters;

public ClientHandler(Socket socket) {

this.socket = socket;

this.clientWriters = new ArrayList<>();

}

@Override

public void run() {

try {

in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

out = new PrintWriter(socket.getOutputStream(), true);

synchronized (clientWriters) {

clientWriters.add(out);

}

String message;

while ((message = in.readLine()) != null) {

System.out.println("Mensaje recibido: " + message);

synchronized (clientWriters) {

for (PrintWriter writer : clientWriters) {

writer.println(message);

}

}

}

} catch (Exception e) {

e.printStackTrace();

}

}

}

### **3.3 Implementación del Cliente (**Client.java**)**

El cliente se conecta al servidor especificando su dirección IP y puerto. La comunicación se maneja con dos hilos para permitir operaciones simultáneas:

1. **Hilo principal:** Lee la entrada del usuario desde la consola (scanner.nextLine()) y la envía directamente al servidor a través del flujo de salida. Este proceso se ejecuta en un bucle infinito para que el usuario pueda enviar mensajes en cualquier momento.
2. **Hilo secundario (receiveMessages):** Se dedica exclusivamente a leer los mensajes que llegan del servidor a través del flujo de entrada (in.readLine()) y los muestra en la consola del cliente. Esto permite que el cliente reciba mensajes de otros usuarios de forma asíncrona, sin interrumpir su capacidad para escribir un nuevo mensaje.

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.net.Socket;

import java.util.Scanner;

public class Client {

private static final String SERVER\_ADDRESS = "127.0.0.1";

private static final int SERVER\_PORT = 5555;

public static void main(String[] args) {

try (

Socket socket = new Socket(SERVER\_ADDRESS, SERVER\_PORT);

PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

Scanner scanner = new Scanner(System.in)

) {

// Hilo para recibir mensajes del servidor

Thread receiveMessages = new Thread(() -> {

try {

String message;

while ((message = in.readLine()) != null) {

System.out.println("Mensaje recibido: " + message);

}

} catch (IOException e) {

e.printStackTrace();

}

});

receiveMessages.start();

// Enviar mensajes al servidor

System.out.println("Conectado al chat. Escribe tu mensaje:");

while (true) {

String messageToSend = scanner.nextLine();

out.println(messageToSend);

}

} catch (Exception e) {

e.printStackTrace();

}

}

}

A screenshot of a computer

AI-generated content may be incorrect.  
Figura 1. Server ejecutando (se muestran los mensajes del Cliente 1 y el Cliente 2).

A screenshot of a computer screen

AI-generated content may be incorrect.  
Figura 2. Cliente 1 conectado.

A screenshot of a computer

AI-generated content may be incorrect.  
Figura 2. Cliente 2 conectado.

## Conclusión

La práctica permitió implementar con éxito un sistema de chat funcional, demostrando la necesidad y efectividad de la sincronización en aplicaciones multihilo. El uso de bloques synchronized fue crucial para proteger la lista de clientes (clientWriters), un recurso compartido, evitando condiciones de carrera que podrían haber ocurrido al añadir nuevos clientes o al retransmitir mensajes de forma concurrente.

Se concluye que sin una sincronización adecuada, una aplicación concurrente es propensa a errores impredecibles y difíciles de depurar. La correcta identificación de las secciones críticas y la aplicación de mecanismos de bloqueo como synchronized son fundamentales para garantizar la integridad de los datos y el comportamiento correcto del programa.

## Pregunta

### ¿Cómo podrían generar un negocio a través de lo visto en la práctica?

El modelo de servidor de chat puede escalarse para crear una plataforma de comunicación empresarial segura (similar a Slack o Microsoft Teams). El negocio podría ofrecer este software como un servicio (SaaS) con diferentes planes de suscripción. Las características premium podrían incluir historiales de chat persistentes, integraciones con otras herramientas (bots, calendarios), roles y permisos de usuario, y canales privados, generando ingresos recurrentes de empresas que buscan una solución de comunicación interna.

## Bibliografía

[1] J. Bloch, Effective Java, 3rd Edition. Boston: Addison-Wesley Professional, 2018.

[2] B. Goetz et al., Java Concurrency in Practice. Upper Saddle River, NJ: Addison-Wesley, 2006.

[3] H. Schildt, Java: The Complete Reference, Twelfth Edition. New York: McGraw-Hill Education, 2021.