INSTITUTO POLITÉCTICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

Carrera:

Ingeniería en Sistemas Computacionales

Unidad de aprendizaje:

Sistemas Distribuidos

Actividad:

Practica 2 – Cliente/Servidor

Integrante:

Hernández Vázquez Jorge Daniel

Profesor:

Chadwick Carreto Arellano

Fecha de entrega:

03/03/2025

Índice de contenido

Índice de Figuras

Antecedente

En la actualidad, la comunicación entre dispositivos y la transferencia eficiente de datos son aspectos fundamentales en el desarrollo de sistemas distribuidos. Los **sockets** juegan un papel crucial en este contexto, ya que permiten la interconexión entre procesos a través de una red, facilitando la implementación de arquitecturas Cliente/Servidor. Su correcta utilización es esencial para garantizar una comunicación confiable y eficiente, lo que resulta clave en el desarrollo de aplicaciones que requieren intercambio de información en tiempo real, acceso remoto a servicios y procesamiento distribuido de datos.

Modelo Cliente/Servidor

El modelo Cliente/Servidor es una de las arquitecturas más utilizadas en la informática moderna, ya que permite la comunicación eficiente entre dispositivos en una red. Su concepto principal radica en la diferenciación entre dos tipos de procesos: los servidores, encargados de ofrecer y gestionar recursos o servicios, y los clientes, que consumen dichos servicios mediante solicitudes específicas. Esta separación facilita la centralización de la información y la delegación de responsabilidades, lo que mejora la administración de los sistemas distribuidos y optimiza el acceso a los datos.

Históricamente, esta arquitectura surgió con el desarrollo de las redes de computadoras y se consolidó con la popularización de Internet. En sus inicios, la computación se basaba en modelos centralizados con mainframes, donde múltiples terminales dependían de un único sistema. Sin embargo, con la aparición de computadoras personales y redes más avanzadas, se hizo viable distribuir las tareas entre clientes y servidores, permitiendo mayor flexibilidad y eficiencia. En la actualidad, esta arquitectura es la base de numerosos servicios, desde aplicaciones web hasta sistemas en la nube y videojuegos en línea.

La comunicación entre clientes y servidores se basa en el intercambio de mensajes a través de protocolos de red. Entre los más utilizados se encuentran TCP/IP, que proporciona una transmisión confiable, HTTP/HTTPS, empleado en la web, FTP, para transferencia de archivos, y SSH, que permite conexiones seguras. Estos protocolos garantizan la correcta transmisión de datos y establecen reglas claras para la comunicación entre dispositivos. En la siguiente Figura podemos observar un diagrama de como funciona este tipo de modelo.

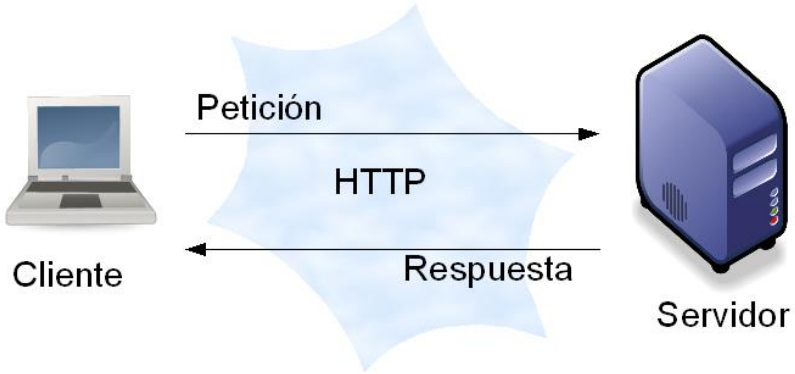


Figura 1. Diagrama del funcionamiento Modelo Cliente/Servidor.

De igual manera, uno de los elementos clave en la implementación de sistemas Cliente/Servidor son los sockets, que actúan como puntos de conexión entre procesos en una red. En términos simples, un socket es una interfaz de comunicación que permite enviar y recibir datos, ya sea en una misma máquina o en dispositivos remotos. Existen dos tipos principales: los sockets de flujo, que utilizan TCP y garantizan una comunicación confiable y ordenada, y los sockets de datagrama, basados en UDP, que son más rápidos, pero no garantizan la entrega de datos. Para esta práctica, se emplearán sockets de flujo debido a su fiabilidad y control en la transmisión de información.

A pesar de sus ventajas, el modelo Cliente/Servidor enfrenta desafíos importantes. Uno de ellos es la gestión de concurrencia, ya que un servidor puede recibir múltiples solicitudes simultáneamente. Para manejar esta carga, se emplean mecanismos como hilos, procesos concurrentes o balanceo de carga. Otro problema es la dependencia del servidor, pues si este falla sin un sistema de respaldo, los clientes quedan inoperantes. Para evitarlo, se implementan estrategias como clustering y réplicas de servidores, e incluso la caída del servidor, ya que debido a la disposición centralizada y a la dependencia en un modelo cliente-servidor, la caída del servidor conlleva la caída de todo el sistema. Si el servidor se cae, los clientes dejan de funcionar porque no pueden recibir las respuestas necesarias del servidor.

Con el avance de la tecnología, esta arquitectura ha evolucionado hacia modelos más dinámicos, como los microservicios, que dividen las aplicaciones en componentes independientes, y la computación en la nube, que permite escalabilidad global y alta disponibilidad. Estas innovaciones han optimizado el rendimiento y la confiabilidad de los sistemas basados en Cliente/Servidor.

En general, esta arquitectura sigue siendo un pilar fundamental en el desarrollo de sistemas informáticos. Su flexibilidad y eficiencia la han convertido en la base de múltiples aplicaciones y servicios.

Sockets

Los sockets, son una interfaz de comunicación que permite el intercambio de datos entre dos procesos, ya sea en la misma máquina o en dispositivos remotos conectados a través de una red. Es un mecanismo fundamental en el desarrollo de sistemas distribuidos, ya que proporciona un punto final para la comunicación, permitiendo la transmisión de información de manera eficiente y estructurada.

El concepto de socket surgió con la evolución de las redes y los sistemas Cliente/Servidor, donde se requería una forma estándar de comunicación entre aplicaciones en diferentes dispositivos. En términos simples, un socket es una combinación de una dirección IP y un número de puerto, que juntos identifican de manera única una conexión en una red.

Para establecer una comunicación entre un cliente y un servidor mediante sockets, es necesario seguir un proceso estructurado. El servidor debe crear un socket, enlazarlo a un puerto específico, escuchar conexiones entrantes y aceptar solicitudes de clientes. Por su parte, el cliente crea un socket y lo conecta al servidor, estableciendo así un canal de comunicación a través del cual se pueden intercambiar mensajes.

Los sockets se dividen en dos tipos principales según el protocolo de transporte utilizado:

- **Sockets de flujo (Stream Sockets):** Basados en el protocolo TCP (Transmission Control Protocol), proporcionan una comunicación orientada a la conexión, asegurando que los datos se entreguen en el mismo orden en que fueron enviados y sin pérdidas. Son ampliamente

utilizados en aplicaciones que requieren fiabilidad, como la transferencia de archivos, la navegación web y los servicios de correo electrónico.

- **Sockets de datagrama (Datagram Sockets):** Utilizan UDP (User Datagram Protocol) y permiten una comunicación sin conexión, lo que significa que los mensajes pueden perderse o llegar en desorden. Son empleados en aplicaciones donde la velocidad es prioritaria sobre la fiabilidad, como los servicios de streaming y los videojuegos en línea.

Desde el punto de vista de la programación, los sockets se implementan a través de API específicas del sistema operativo, como la API de Berkeley Sockets en Unix/Linux o la Winsock en Windows. Lenguajes de programación como C, Python y Java incluyen bibliotecas que facilitan la creación y gestión de sockets, proporcionando funciones para abrir conexiones, enviar y recibir datos, manejar errores y cerrar la conexión de manera segura. En la siguiente Figura podemos observar como se conecta un cliente/servidor con sockets.

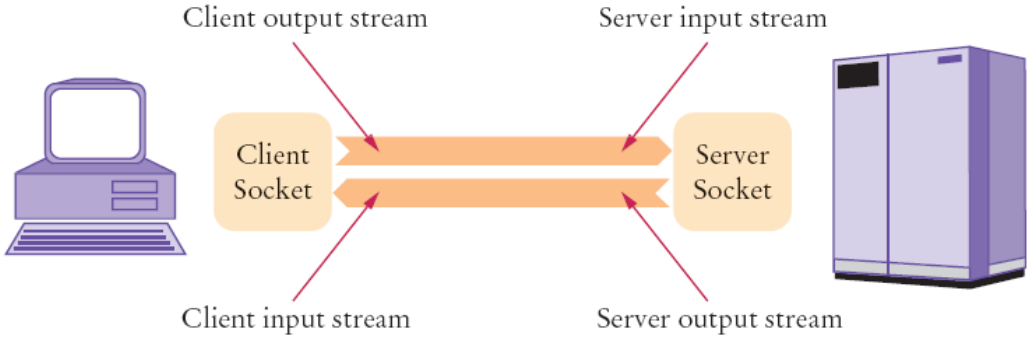


Figura 2. Cliente y Servidor conectados

A pesar de sus ventajas, el uso de sockets también presenta desafíos. Uno de los principales es la gestión de concurrencia, ya que los servidores pueden recibir múltiples solicitudes simultáneamente. Para manejar esta situación, se utilizan hilos (threads), procesos concurrentes o modelos asíncronos que permiten atender múltiples clientes de manera eficiente. Otro desafío es la seguridad, ya que las conexiones pueden ser vulnerables a ataques como intercepción de datos, inyección de paquetes y denegación de servicio (DoS). Para mitigar estos riesgos, se emplean mecanismos de cifrado como TLS/SSL, autenticación de clientes y servidores, y validación de paquetes, sin embargo, nosotros los usaremos de forma muy básica. En la siguiente Figura vemos la secuencia de conexión que se sigue con el cliente y el servidor.

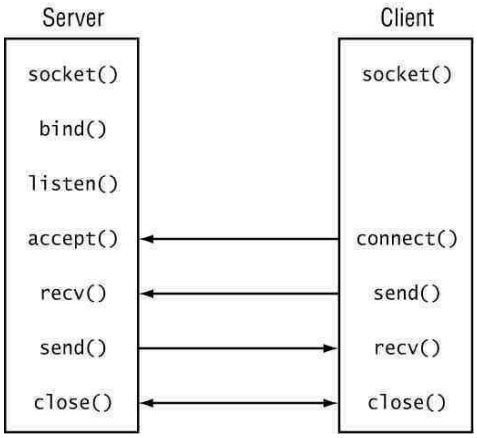


Figura 3. Secuencia de Conexión

En general, los sockets son una herramienta fundamental en la comunicación entre procesos dentro de los sistemas distribuidos. Su flexibilidad y eficiencia han permitido su adopción en una amplia variedad de aplicaciones, desde servicios web hasta sistemas en tiempo real.

Planteamiento del problema

En el contexto de los sistemas distribuidos, la implementación de arquitecturas Cliente/Servidor es fundamental para permitir la interacción entre múltiples usuarios y un servicio centralizado. Sin embargo, la gestión eficiente de múltiples clientes concurrentes, la sincronización de recursos compartidos y la comunicación en tiempo real representan desafíos importantes en este tipo de sistemas, además de que, en caso de caerse el servidor, todo el sistema cae.

En este practica, se desarrolla una simulación de una panadería, siguiente claramente el ejemplo desarrollado en la práctica anterior, solo que ahora en vez de utilizar únicamente hilos, ahora la panadería se opera bajo un modelo Cliente/Servidor utilizando sockets en Java. El servidor actúa como la panadería, administrando el stock de pan y procesando las solicitudes de compra de los clientes, mientras que los clientes pueden consultar el inventario o realizar compras de manera concurrente.

Uno de los principales problemas identificados en este escenario es la gestión de la concurrencia, ya que múltiples clientes pueden intentar comprar pan al mismo tiempo, lo que requiere una adecuada sincronización para evitar inconsistencias en el stock. Además, la panadería debe contar con un mecanismo para reabastecer su inventario periódicamente, lo que introduce la necesidad de un proceso adicional que gestione la producción de pan.

Por otro lado, se debe garantizar una comunicación eficiente entre el servidor y los clientes, asegurando que las respuestas sean claras y oportunas para evitar bloqueos o tiempos de espera prolongados. La implementación de hilos en el servidor es clave para permitir la atención simultánea de múltiples clientes sin afectar el rendimiento del sistema.

Este practica busca resolver estos desafíos mediante el uso de Java y sockets, implementando una arquitectura Cliente/Servidor con soporte para múltiples conexiones concurrentes y una gestión eficiente de los recursos compartidos.

Propuesta de solución

Para abordar los problemas identificados en la simulación de la panadería, se propone una solución basada en una arquitectura Cliente/Servidor utilizando **Java y sockets**. La práctica estará realizada de manera que el servidor administre eficientemente la concurrencia de múltiples clientes y la sincronización de los recursos compartidos.

El servidor de la panadería será responsable de manejar las solicitudes de los clientes de manera concurrente, para lo cual se implementará un **pool de hilos**, permitiendo que múltiples clientes puedan conectarse simultáneamente sin afectar el rendimiento del sistema. Cada cliente que se conecte será atendido en un hilo independiente, lo que garantiza que las operaciones se realicen sin bloqueos innecesarios.

Para la gestión del stock de pan, se utilizará una estructura de datos sincronizada que permita controlar la disponibilidad del producto en tiempo real. Esto evitará inconsistencias en los datos cuando varios clientes intenten comprar pan al mismo tiempo. Además, se implementará un **hilo adicional**

encargado de reabastecer periódicamente el inventario de pan, simulando la producción de pan en la panadería.

En el lado del cliente, mediante la consola se le permitirá interactuar con el servidor mediante opciones como consultar el stock, comprar pan y salir de la panadería. La comunicación entre cliente y servidor se realizará mediante el protocolo **TCP**, asegurando una transmisión confiable de datos y evitando la pérdida de información durante las interacciones.

Con esta solución, se busca proporcionar un practica robusta donde se ejemplifique de manera correcta el uso de hilos y el modelo cliente/servidor, que permita la interacción fluida entre los clientes y el servidor, asegurando la sincronización de los recursos compartidos y optimizando la comunicación en entornos concurrentes.

Materiales y métodos empleados

Para llevar a cabo la práctica y desarrollar la simulación de la panadería con el modelo cliente/servidor, se utilizaron las siguientes herramientas y metodologías:

Materiales (Herramientas utilizadas)

1. **Lenguaje de programación: Java**
    - Se eligió Java debido a su robusto manejo de hilos y sus herramientas integradas para la concurrencia, como la palabra clave synchronized.
2. **Entorno de desarrollo: Visual Studio Code (VS Code)**
3. **JDK (Java Development Kit):**
    - Se usó el JDK para compilar y ejecutar el programa.
4. **Sistema Operativo: Windows**
    - La práctica se desarrolló y ejecutó en un sistema operativo Windows.

Métodos Empleados

1. **Programación con Sockets TCP**

Se utilizó la API de sockets de Java para establecer la comunicación entre el cliente y el servidor mediante el protocolo TCP, que garantiza una transmisión confiable de datos.

- El servidor emplea un ServerSocket para escuchar conexiones entrantes en un puerto específico (5000).
- Los clientes utilizan Socket para conectarse al servidor e intercambiar mensajes mediante flujos de entrada (InputStreamReader) y salida (PrintWriter).

2. **Manejo de Concurrencia con Hilos**

Dado que múltiples clientes pueden conectarse simultáneamente, el servidor implementa un pool de hilos mediante la clase ThreadPoolExecutor. Esto permite la atención concurrente de múltiples clientes sin afectar el rendimiento del sistema.

Además, cada cliente se gestiona dentro de un hilo independiente (HiloCliente), lo que permite que las interacciones ocurran de manera paralela sin interferencias.

3. **Sincronización de Recursos Compartidos**

El stock de pan es un recurso compartido entre todos los clientes, por lo que se implementaron métodos sincronizados (synchronized) dentro de la clase Panaderia para evitar condiciones de carrera y garantizar la consistencia de los datos.

- El método comprarPan(int cantidad, String cliente) verifica la disponibilidad de stock y lo actualiza de manera segura.
- El método añadirStock(int cantidad) es utilizado por un hilo adicional encargado de hornear pan cada cierto tiempo.

4. **Implementación de un Productor (Horno) y Consumidores (Clientes)**

Siguiendo el modelo Productor-Consumidor, se creó un hilo adicional (HornoPanadero) que periódicamente reabastece el stock de pan en la panadería. Este hilo opera en segundo plano y notifica a los clientes en caso de que deban esperar por más pan.

5. **Manejo de Entrada y Salida de Datos**

- En el servidor, se utiliza BufferedReader y PrintWriter para recibir y enviar mensajes a los clientes de forma eficiente.
- En el cliente, se emplea Scanner para capturar las entradas del usuario y enviarlas al servidor.
- Se implementó un menú interactivo en el cliente, permitiendo al usuario consultar el stock, comprar pan o salir del sistema.

Con estos materiales y métodos, se logró realizar la práctica, permitiendo visualizar de manera práctica el modelo cliente/servidor.

Desarrollo de la solución

La solución se basa en la implementación de un modelo Cliente/Servidor utilizando **sockets en Java**, donde la panadería funciona como el servidor y los clientes como múltiples conexiones simultáneas. Se emplearon mecanismos de **concurrencia y sincronización** para evitar problemas en el acceso al stock de panes.

A continuación, se describe cómo se llegó a la solución:

1. Clase Panaderia (Recurso Compartido y Servidor)

La panadería es el recurso compartido entre los clientes. Se ha implementado la clase Panaderia, que tiene un atributo stockPan para representar la cantidad de panes disponibles. El constructor inicializa el stock de panes. Y posteriormente, se utiliza el metodo comprarPan, el cual asegura que los hilos (clientes) no accedan al stock de panes al mismo tiempo. El uso de synchronized garantiza que solo un hilo pueda modificar el stock a la vez, evitando condiciones de carrera. En la siguiente Figura podemos observar el bloque de codigo de lo anterior.

```java
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;

// Clase que representa la panadería en el servidor
class Panaderia {
    private int stockPan;

    public Panaderia(int stockInicial) {
        this.stockPan = stockInicial;
    }

    public synchronized String comprarPan(int cantidad, String cliente) {
        if (stockPan < cantidad) {
            System.out.println(cliente + " no tiene suficiente pan. Esperando...");
            // Notificar de inmediato al cliente que debe esperar mientras se hornean más panes.
            return "No hay suficiente pan. Por favor, espere mientras horneamos más.";
        }

        // Si hay suficiente pan, lo compramos
        stockPan -= cantidad;
        System.out.println(cliente + " compró " + cantidad + " panes. Panes restantes: " + stockPan);
        return "Compra exitosa. Panes restantes: " + stockPan;
    }



    public synchronized int getStockPan() {
        return stockPan;
    }

    public synchronized void añadirStock(int cantidad) {
        stockPan += cantidad;
        System.out.println("Se hornearon " + cantidad + " panes. Nuevo stock: " + stockPan);
        notifyAll(); // Despierta a los clientes en espera
    }
}
```

Figura 4. Clase Panaderia y método comprarPan en el Servidor.

2. Clase HornoPanadero (Hilo para la producción de pan)

El hilo HornoPanadero simula la producción de panes cada ciertos intervalos de tiempo (cada 10 segundos en este caso), para garantizar que haya más pan disponible cuando el stock se agote. Esto lo podemos ver en la siguiente Figura:

```java
// Hilo que maneja la producción de pan cada cierto tiempo
class HornoPanadero extends Thread {
    private final Panaderia panaderia;

    public HornoPanadero(Panaderia panaderia) {
        this.panaderia = panaderia;
    }

    @Override
    public void run() {
        while (true) {
            try {
                Thread.sleep(millis:10000); // Cada 10 segundos se hornean más panes
                panaderia.añadirStock(cantidad:5);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Figura 5. Clase HornoPanadero en el Servidor.

3. Clase HiloCliente (Representa a los clientes)

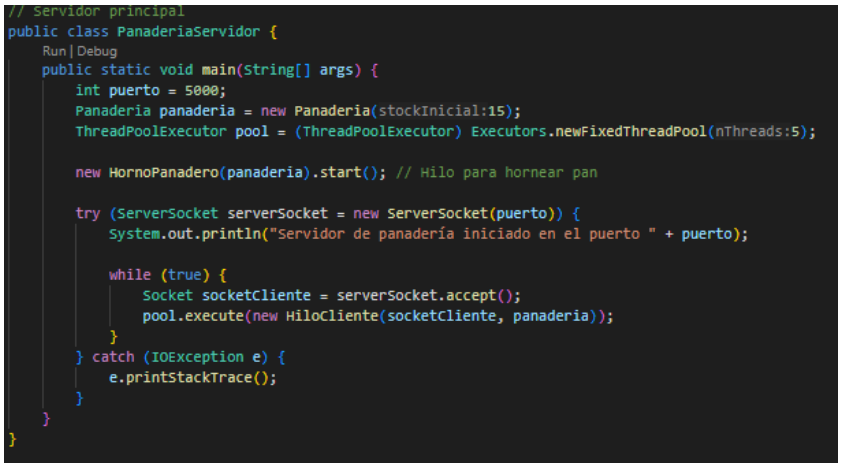
Cada cliente es representado por un hilo. La clase HiloCliente maneja las interacciones con el cliente, permitiéndole hacer compras de pan. Este hilo lee las entradas del cliente, muestra las opciones y comunica las respuestas correspondientes. En la siguiente Figura podemos ver el bloque de codigo:

```java
// Hilo para manejar cada cliente
class HiloCliente extends Thread {
    private final Socket socket;
    private final Panaderia panaderia;

    public HiloCliente(Socket socket, Panaderia panaderia) {
        this.socket = socket;
        this.panaderia = panaderia;
    }

    @Override
    public void run() {
        try (
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(), autoFlush:true)
        ) {
            // Leer el nombre del cliente y dar la bienvenida
            String nombreCliente = in.readLine();
            System.out.println(nombreCliente + " se ha conectado.");
            out.println("Bienvenido a la Panadería, " + nombreCliente + "!");

            while (true) {
                // Enviar menú sin que se repita innecesariamente
                out.println(x:"\n--- Menú Panadería ---");
                out.println(x:"1. Ver stock");
                out.println(x:"2. Comprar pan");
                out.println(x:"3. Salir");
                out.println(x:"Seleccione una opción:");

                String opcionStr = in.readLine();
                if (opcionStr == null) break; // Cliente cerró la conexión

                int opcion;
                try {
                    opcion = Integer.parseInt(opcionStr);
                } catch (NumberFormatException e) {
                    out.println(x:"Opción no válida. Intente de nuevo.");
                    continue;
                }

                if (opcion == 1) {
                    out.println("Stock disponible: " + panaderia.getStockPan() + " panes.");
                } else if (opcion == 2) {
                    out.println(x:"Ingrese la cantidad de pan que desea comprar:");
                    String cantidadStr = in.readLine();
                    int cantidad;
                    try {
                        cantidad = Integer.parseInt(cantidadStr);
                    } catch (NumberFormatException e) {
                        out.println(x:"Cantidad no válida. Intente de nuevo.");
                        continue;
                    }
                    String respuesta = panaderia.comprarPan(cantidad, nombreCliente);
                    out.println(respuesta);
                } else if (opcion == 3) {
                    out.println(x:"Gracias por visitar la panadería. ¡Hasta luego!");
                    System.out.println(nombreCliente + " se ha desconectado.");
                    break;
                } else {
                    out.println(x:"Opción no válida. Intente de nuevo.");
                }
            }
        } catch (IOException e) {
            System.err.println("Error en la conexión con el cliente: " + e.getMessage());
        } finally {
            try {
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Figura 6. Clase HiloCliente en el Servidor.

4. Servidor Principal (PanaderiaServidor)

La clase principal PanaderiaServidor maneja la inicialización del servidor y acepta múltiples clientes, asignándoles un hilo cada vez que se conecta. A continuación, en la siguiente Figura podemos observar la clase principal del servidor:



```java
// Servidor principal
public class PanaderiaServidor {
    Run | Debug
    public static void main(String[] args) {
        int puerto = 5000;
        Panaderia panaderia = new Panaderia(stockInicial:15);
        ThreadPoolExecutor pool = (ThreadPoolExecutor) Executors.newFixedThreadPool(nThreads:5);

        new HornoPanadero(panaderia).start(); // Hilo para hornear pan

        try (ServerSocket serverSocket = new ServerSocket(puerto)) {
            System.out.println("Servidor de panadería iniciado en el puerto " + puerto);

            while (true) {
                Socket socketCliente = serverSocket.accept();
                pool.execute(new HiloCliente(socketCliente, panaderia));
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

El cliente debe conectarse al servidor y enviar solicitudes, como ver el stock de pan, comprar pan o salir de la panadería. A continuación, se detalla el proceso de la implementación del cliente:

5. Conexión al Servidor

Primero, el cliente se conecta al servidor usando un Socket a través de la dirección y puerto proporcionados. La conexión se establece a través de un flujo de entrada y salida para leer y escribir mensajes. Lo anterior lo podemos ver en la siguiente Figura:

```java
import java.io.*;
import java.net.Socket;
import java.util.Scanner;

public class PanaderiaCliente {
    Run | Debug
    public static void main(String[] args) {
        String servidor = "localhost"; // Dirección del servidor
        int puerto = 5000; // Puerto del servidor

        try (Socket socket = new Socket(servidor, puerto);
             BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
             PrintWriter out = new PrintWriter(socket.getOutputStream(), autoFlush:true);
             Scanner scanner = new Scanner(System.in)) {

            System.out.println(x:"Conectado a la panadería.");
```

Figura 7. Clase PanaderiaCliente, donde se conecta al servidor.

5. Enviar el nombre del Cliente, recibir mensajes y mostrar menú de opciones

Después de que la conexión se establece, el cliente debe enviar su nombre al servidor, lo cual se hará a través del flujo de salida (out). Luego, el servidor enviará un mensaje de bienvenida, que el cliente recibirá y mostrará en su consola.

- El cliente solicita su nombre, lo ingresa desde la consola y lo envía al servidor.
- El servidor responderá con un mensaje de bienvenida, que el cliente leerá e imprimirá en su consola.

Una vez hecho este proceso, después de recibir el mensaje de bienvenida, el cliente se encuentra con un menú interactivo que le permite elegir qué acción realizar. El menú incluye las siguientes opciones:

1. Ver el stock de pan disponible.
2. Comprar pan.
3. Salir de la panadería.

El cliente entra en un ciclo donde recibirá continuamente el menú del servidor y podrá tomar decisiones hasta que decida salir.

Este bloque de código lo podemos en la siguiente Figura:

```
// Ingresar nombre del cliente
System.out.print(s:"Ingrese su nombre: ");
String nombre = scanner.nextLine();
out.println(nombre); // Enviar el nombre al servidor
System.out.println(in.readLine()); // Mensaje de bienvenida del servidor

while (true) {
    // Leer y mostrar el menú hasta que se reciba "Seleccione una opción:"
    String linea;
    while (!(linea = in.readLine()).contains(s:"Seleccione una opción")) {
        System.out.println(linea);
    }
    System.out.println(linea); // Imprime "Seleccione una opción:"

    // Leer opción del usuario
    System.out.print(s:"Opción: ");
    String opcionStr = scanner.nextLine();
    out.println(opcionStr); // Enviar opción al servidor

    if (opcionStr.equals(anObject:"1") || opcionStr.equals(anObject:"3")) {
        // Si es ver stock o salir, recibir respuesta del servidor
        System.out.println("Servidor: " + in.readLine());
        if (opcionStr.equals(anObject:"3")) break; // Salir del loop si elige salir
    } else if (opcionStr.equals(anObject:"2")) {
        // Comprar pan
        System.out.println(in.readLine()); // Esperar mensaje del servidor "Ingrese la cantidad..."
        String cantidad = scanner.nextLine();
        out.println(cantidad); // Enviar cantidad al servidor
        String respuesta = in.readLine(); // Recibir respuesta de compra
        System.out.println("Servidor: " + respuesta);
    } else {
        System.out.println(x:"Opción no válida.");
    }
}

System.out.println(x:"Saliendo de la panadería... ¡Hasta luego!");
```

Figura 8. Acciones que puede realizar el cliente.

- **Ciclo de Menú:** El ciclo while(true) asegura que el cliente vea el menú de opciones repetidamente hasta que decida salir (opción 3).
- **Leer y Mostrar Menú:** Cada vez que el cliente recibe una respuesta del servidor que contiene el mensaje "Seleccione una opción", el cliente lo muestra en la consola.
- **Elegir Opción:** El cliente ingresa una opción desde la consola, que se envía al servidor usando el flujo de salida (out.println(opcionStr)).
    - ✓ **Opción 1 (Ver Stock):** Si el cliente selecciona "1", el servidor responderá con la cantidad de panes disponibles, que el cliente imprimirá.
    - ✓ **Opción 2 (Comprar Pan):** Si el cliente elige "2", se le pedirá la cantidad de pan que desea comprar. El cliente ingresa la cantidad y la envía al servidor. Luego, el cliente recibe la respuesta de si la compra fue exitosa o si debe esperar a que se horneen más panes.
    - ✓ **Opción 3 (Salir):** Si el cliente selecciona "3", el cliente se desconectará del servidor y finalizará el programa.

6. Manejo de errores

Para finalizar, es importante que el cliente maneje adecuadamente los errores, especialmente en caso de que la conexión con el servidor se interrumpa. Si ocurre algún error de I/O, el cliente debe mostrar un mensaje adecuado y finalizar la conexión de manera limpia. Lo anterior lo podemos observar en el siguiente bloque de código de la Figura 99.

```
        } catch (IOException e) {
            System.err.println("Error de conexión: " + e.getMessage());
        }
    }
}
```

Figura 9. Acciones que puede realizar el cliente.

Al final, al ejecutar el código del servidor, este quedará esperando las conexiones de los clientes. Los clientes podrán conectarse al servidor y hacer compras, ver el stock o salir, sin interferir con las acciones de otros clientes gracias al uso de hilos en el servidor.

Resultados

Al ejecutar la solución propuesta, el sistema permite que múltiples clientes interactúen simultáneamente con el servidor de la panadería de forma eficiente y sin conflictos. A continuación, se detallan los resultados obtenidos al ejecutar el código en el servidor y los clientes.

En este caso al ejecutar el servidor en el puerto 5000, el servidor se encuentra en espera de conexiones de clientes. Cada vez que un cliente se conecta, el servidor acepta la conexión y crea un hilo independiente para manejar la interacción con ese cliente, permitiendo que varios clientes se conecten de manera concurrente, además de generar un nuevo stock de panes cada cierto tiempo. Esto lo podemos ver en la siguiente Figura:

```
Servidor de panadería iniciado en el puerto 5000
Se hornearon 5 panes. Nuevo stock: 20
Se hornearon 5 panes. Nuevo stock: 25
Se hornearon 5 panes. Nuevo stock: 30
Se hornearon 5 panes. Nuevo stock: 35
```

Figura 10. Respuesta del servidor al iniciar.

Ahora al momento de ejecutar el código del cliente, en caso de realizarse la conexión de manera correcta se nos proporciona un mensaje de conexión exitosa, y se nos solicita un nombre, una vez ingresado este nombre, se despliega el menú de opciones.

```
Conectado a la panadería.
Ingrese su nombre: Juan
Bienvenido a la Panadería, Juan!

--- Menú Panadería ---
1. Ver stock
2. Comprar pan
3. Salir
Seleccione una opción:
Opción:
```

Figura 11. Conexión desde el cliente.

Cuando un cliente compra pan, el servidor actualiza el stock de pan de manera sincronizada para evitar condiciones de carrera. Si no hay suficiente stock, el servidor notifica al cliente. Esto garantiza

que el número de panes no sea modificado incorrectamente por múltiples clientes concurrentes, y en el servidor podemos ver los siguientes mensajes:

```
Juan se ha conectado.
Se hornearon 5 panes. Nuevo stock: 140
Se hornearon 5 panes. Nuevo stock: 145
Se hornearon 5 panes. Nuevo stock: 150
Se hornearon 5 panes. Nuevo stock: 155
Se hornearon 5 panes. Nuevo stock: 160
Se hornearon 5 panes. Nuevo stock: 165
Se hornearon 5 panes. Nuevo stock: 170
Se hornearon 5 panes. Nuevo stock: 175
Se hornearon 5 panes. Nuevo stock: 180
Se hornearon 5 panes. Nuevo stock: 185
Se hornearon 5 panes. Nuevo stock: 190
Se hornearon 5 panes. Nuevo stock: 195
Luis se ha conectado.
```

Figura 12. Mensajes en el servidor de conexión y Horneado de nuevos panes.

Los clientes pueden elegir entre tres opciones en el menú: ver el stock de pan, comprar pan o salir. Al seleccionar la opción de compra, el cliente es solicitado a ingresar la cantidad de pan que desea comprar. Si la cantidad solicitada es mayor al stock disponible, el cliente recibe un mensaje indicando que debe esperar hasta que se horneen más panes.

```
--- Menú Panadería ---
1. Ver stock
2. Comprar pan
3. Salir
Seleccione una opción:
Opción: 2
Ingrese la cantidad de pan que desea comprar:
30
Servidor: No hay suficiente pan. Por favor, espere mientras horneamos más.
```

Figura 13. Menú de opciones y mensaje cuando se compra más stock del disponible.

O en caso de si realizarse la compra correctamente, se indica lo siguiente:

```
--- Menú Panadería ---
1. Ver stock
2. Comprar pan
3. Salir
Seleccione una opción:
Opción: 2
Ingrese la cantidad de pan que desea comprar:
4
Servidor: Compra exitosa. Panes restantes: 31
```

```
luis se ha conectado.
Se hornearon 5 panes. Nuevo stock: 20
Se hornearon 5 panes. Nuevo stock: 25
Marco se ha conectado.
Se hornearon 5 panes. Nuevo stock: 30
Se hornearon 5 panes. Nuevo stock: 35
Marco compró 4 panes. Panes restantes: 31
Se hornearon 5 panes. Nuevo stock: 36
luis compró 31 panes. Panes restantes: 5
Se hornearon 5 panes. Nuevo stock: 10
```

Figura 14. Compra de pan y mensajes recibidos en el Servidor al comprar pan.

Ahora, para ver el stock disponible, al seleccionar la opción 1, se nos proporciona lo siguiente:

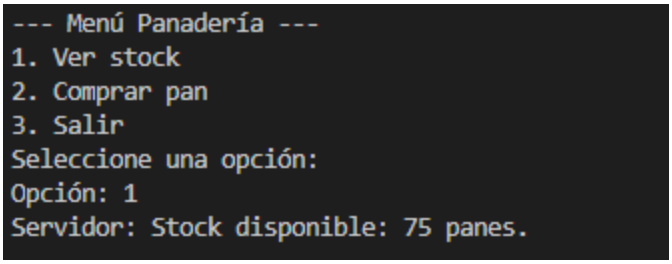


Figura 15. Ver Stock desde el cliente.

Finalmente, en caso de que seleccionemos la opción 3, nos desconectamos del servidor y se cierra la conexión con este.

Figura 16. Salir del servidor desde el cliente.

Cabe mencionar que podemos conectarnos desde varios clientes, y en este caso como se pueden observar en las Figuras anteriores, realice la conexión con dos clientes distintos, así que en general la implementación del modelo Cliente-Servidor utilizando sockets en Java ha sido exitosa. El sistema permite manejar múltiples clientes de manera concurrente sin conflictos, asegurando que el stock de pan se maneje de manera correcta y que los clientes reciban una respuesta adecuada a sus solicitudes.

Conclusión

A lo largo de la práctica, pude reforzar mis conocimientos sobre el modelo cliente-servidor, que había aprendido previamente, pero con esta práctica recordé un poco más de algunas cosas que había olvidado. A través de la simulación de una panadería con procesos e hilos, pude comprender mejor cómo estos componentes trabajan en conjunto para manejar solicitudes concurrentes, gestionando de manera eficiente el acceso a recursos compartidos. Además, entendí la importancia de la sincronización para evitar conflictos y garantizar que los clientes puedan interactuar de forma segura con el servidor.

La implementación de hilos fue un desafío que me permitió profundizar en conceptos clave de concurrencia y gestión de procesos, esenciales en la programación de sistemas distribuidos. La interacción entre cliente y servidor, que se puede comparar con las interacciones reales entre usuarios y servicios en sistemas distribuidos, fue una excelente oportunidad para aplicar teorías de redes y comunicación que son fundamentales en el ámbito de la informática.

En comparación con la práctica pasada sobre procesos e hilos, en la que también implementé la simulación de una panadería, esta nueva versión con cliente-servidor me permitió comprender cómo evolucionan los sistemas al agregar comunicación entre múltiples componentes. Mientras que en la primera práctica el enfoque estaba en la gestión interna de los hilos y la sincronización de recursos dentro de un solo programa, en esta segunda práctica logré ver cómo esas mismas ideas se extienden a un sistema distribuido, donde varios clientes pueden interactuar con un servidor central de manera

concurrente. Esto me ayudó a visualizar mejor la transición de aplicaciones locales a sistemas más complejos y escalables.

En general, esta práctica no solo me ayudó a reforzar mis conocimientos previos sobre hilos y procesos, sino que también me permitió entender mejor el papel que desempeña el modelo cliente-servidor en la arquitectura de sistemas distribuidos. Esto me motiva a seguir profundizando en cómo optimizar la comunicación entre componentes de un sistema y garantizar un manejo adecuado de la concurrencia, además de seguir viendo como avanzan las cosas, ver la evolución de los sistemas.

Bibliografía

- Code & Coke, "Sockets," 2024. [En línea]. Available: https://psp.codeandcoke.com/apuntes:sockets. [Último acceso: 2 Marzo 2025].
- IBM, "Socket programming," IBM Knowledge Center, 2024. [En línea]. Available: https://www.ibm.com/docs/es/i/7.5?topic=communications-socket-programming. [Último acceso: 2 Marzo 2025].
- Daemon4, "Arquitectura Cliente-Servidor," 2024. [En línea]. Available: https://www.daemon4.com/empresa/noticias/arquitectura-cliente-servidor/. [Último acceso: 2 Marzo 2025].
- M. J. LaMar, *Thread Synchronization in Java*, 2021. [En línea]. Available: https://www.example.com/thread-synchronization-java. [Último acceso: 21 Febrero 2025].
- J. Geeks, "Introducción a la programación con sockets en Java," Geeks for Geeks, 2023. [En línea]. Available: https://www.geeksforgeeks.org/socket-programming-in-java. [Último acceso: 2 Marzo 2025].
- A. Silberschatz, P. Galvin y G. Gagne, *Operating System Concepts*, 10ª ed., Wiley, 2018.

Anexos

Código PanaderiaCliente.java

```
import java.io.*;
import java.net.Socket;
import java.util.Scanner;

public class PanaderiaCliente {
    public static void main(String[] args) {
        String servidor = "localhost"; // Dirección del servidor
        int puerto = 5000; // Puerto del servidor

        try (Socket socket = new Socket(servidor, puerto);
             BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
             PrintWriter out = new PrintWriter(socket.getOutputStream(),
true);
             Scanner scanner = new Scanner(System.in)) {

            System.out.println("Conectado a la panadería.");

            // Ingresar nombre del cliente
            System.out.print("Ingrese su nombre: ");
            String nombre = scanner.nextLine();
            out.println(nombre); // Enviar el nombre al servidor
            System.out.println(in.readLine()); // Mensaje de bienvenida del
servidor

            while (true) {
                // Leer y mostrar el menú hasta que se reciba "Seleccione
una opción:"
                String linea;
                while (!(linea = in.readLine()).contains("Seleccione una
opción")) {
                    System.out.println(linea);
                }
                System.out.println(linea); // Imprime "Seleccione una
opción:"

                // Leer opción del usuario
                System.out.print("Opción: ");
                String opcionStr = scanner.nextLine();
                out.println(opcionStr); // Enviar opción al servidor

                if (opcionStr.equals("1") || opcionStr.equals("3")) {
```

```
                    // Si es ver stock o salir, recibir respuesta del
servidor
                    System.out.println("Servidor: " + in.readLine());
                    if (opcionStr.equals("3")) break; // Salir del loop si
elige salir
                } else if (opcionStr.equals("2")) {
                    // Comprar pan
                    System.out.println(in.readLine()); // Esperar mensaje
del servidor "Ingrese la cantidad..."
                    String cantidad = scanner.nextLine();
                    out.println(cantidad); // Enviar cantidad al servidor
                    String respuesta = in.readLine(); // Recibir respuesta
de compra
                    System.out.println("Servidor: " + respuesta);
                } else {
                    System.out.println("Opcion no válida.");
                }
            }

            System.out.println("Saliendo de la panaderia... ¡Hasta luego!");

        } catch (IOException e) {
            System.err.println("Error de conexion: " + e.getMessage());
        }
    }
}
```

Código PanaderiaServidor.java

```
import java.io.*;

import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;

// Clase que representa la panadería en el servidor
class Panaderia {
    private int stockPan;

    public Panaderia(int stockInicial) {
        this.stockPan = stockInicial;
    }

    public synchronized String comprarPan(int cantidad, String cliente) {
        if (stockPan < cantidad) {
```

```
            System.out.println(cliente + " no tiene suficiente pan.
Esperando...");
            // Notificar de inmediato al cliente que debe esperar mientras
se hornean más panes.
            return "No hay suficiente pan. Por favor, espere mientras
horneamos más.";
        }

        // Si hay suficiente pan, lo compramos
        stockPan -= cantidad;
        System.out.println(cliente + " compró " + cantidad + " panes. Panes
restantes: " + stockPan);
        return "Compra exitosa. Panes restantes: " + stockPan;
    }



    public synchronized int getStockPan() {
        return stockPan;
    }

    public synchronized void añadirStock(int cantidad) {
        stockPan += cantidad;
        System.out.println("Se hornearon " + cantidad + " panes. Nuevo
stock: " + stockPan);
        notifyAll(); // Despierta a los clientes en espera
    }
}

// Hilo que maneja la producción de pan cada cierto tiempo
class HornoPanadero extends Thread {
    private final Panaderia panaderia;

    public HornoPanadero(Panaderia panaderia) {
        this.panaderia = panaderia;
    }

    @Override
    public void run() {
        while (true) {
            try {
                Thread.sleep(10000); // Cada 10 segundos se hornean más
panes
                panaderia.añadirStock(5);
            } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }
        }
    }
}

// Hilo para manejar cada cliente
class HiloCliente extends Thread {
    private final Socket socket;
    private final Panaderia panaderia;

    public HiloCliente(Socket socket, Panaderia panaderia) {
        this.socket = socket;
        this.panaderia = panaderia;
    }

    @Override
    public void run() {
        try (
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(),
true)
        ) {
            // Leer el nombre del cliente y dar la bienvenida
            String nombreCliente = in.readLine();
            System.out.println(nombreCliente + " se ha conectado.");
            out.println("Bienvenido a la Panadería, " + nombreCliente +
"!");

            while (true) {
                // Enviar menú sin que se repita innecesariamente
                out.println("\n--- Menú Panadería ---");
                out.println("1. Ver stock");
                out.println("2. Comprar pan");
                out.println("3. Salir");
                out.println("Seleccione una opción:");

                String opcionStr = in.readLine();
                if (opcionStr == null) break; // Cliente cerró la conexión

                int opcion;
                try {
                    opcion = Integer.parseInt(opcionStr);
                } catch (NumberFormatException e) {
```

```
                    out.println("Opción no válida. Intente de nuevo.");
                    continue;
                }

                if (opcion == 1) {
                    out.println("Stock disponible: " +
panaderia.getStockPan() + " panes.");
                } else if (opcion == 2) {
                    out.println("Ingrese la cantidad de pan que desea
comprar:");
                    String cantidadStr = in.readLine();
                    int cantidad;
                    try {
                        cantidad = Integer.parseInt(cantidadStr);
                    } catch (NumberFormatException e) {
                        out.println("Cantidad no válida. Intente de
nuevo.");
                        continue;
                    }
                    String respuesta = panaderia.comprarPan(cantidad,
nombreCliente);
                    out.println(respuesta);
                } else if (opcion == 3) {
                    out.println("Gracias por visitar la panadería. ¡Hasta
luego!");
                    System.out.println(nombreCliente + " se ha
desconectado.");
                    break;
                } else {
                    out.println("Opción no válida. Intente de nuevo.");
                }
            }
        } catch (IOException e) {
            System.err.println("Error en la conexión con el cliente: " +
e.getMessage());
        } finally {
            try {
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```java
// Servidor principal
public class PanaderiaServidor {
    public static void main(String[] args) {
        int puerto = 5000;
        Panaderia panaderia = new Panaderia(15);
        ThreadPoolExecutor pool = (ThreadPoolExecutor)
Executors.newFixedThreadPool(5);

        new HornoPanadero(panaderia).start(); // Hilo para hornear pan

        try (ServerSocket serverSocket = new ServerSocket(puerto)) {
            System.out.println("Servidor de panadería iniciado en el puerto
" + puerto);

            while (true) {
                Socket socketCliente = serverSocket.accept();
                pool.execute(new HiloCliente(socketCliente, panaderia));
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```