INSTITUTO POLITÉCTICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

Carrera:

Ingeniería en Sistemas Computacionales

Unidad de aprendizaje:

Sistemas Distribuidos

Actividad:

Practica 3 – MultiCliente/Servidor

Integrante:

Hernández Vázquez Jorge Daniel

Profesor:

Chadwick Carreto Arellano

Fecha de entrega:

10/03/2025

Índice de contenido

Índice de Figuras

Antecedente

En los sistemas distribuidos modernos, la comunicación eficiente entre múltiples clientes y servidores es fundamental para garantizar el intercambio de datos en tiempo real. La arquitectura Multicliente/Servidor permite que varios clientes se conecten simultáneamente a un único servidor, lo que la convierte en un modelo ampliamente utilizado en aplicaciones web, plataformas de mensajería, videojuegos en línea y sistemas empresariales.

Dentro de esta arquitectura, el uso de sockets es esencial para establecer y gestionar las conexiones entre clientes y servidores a través de una red. Los sockets de flujo (TCP) son particularmente importantes, ya que proporcionan una comunicación confiable, asegurando que los datos lleguen completos y en el orden correcto.

Uno de los principales desafíos en un entorno Multicliente/Servidor es la gestión de concurrencia, ya que un solo servidor debe atender múltiples solicitudes simultáneamente sin afectar el rendimiento del sistema. Para solucionar este problema, se utilizan técnicas como el multithreading (uso de hilos), el multiprocesamiento o modelos asíncronos de comunicación, dependiendo de los requisitos de la aplicación.

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 líneaLa 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

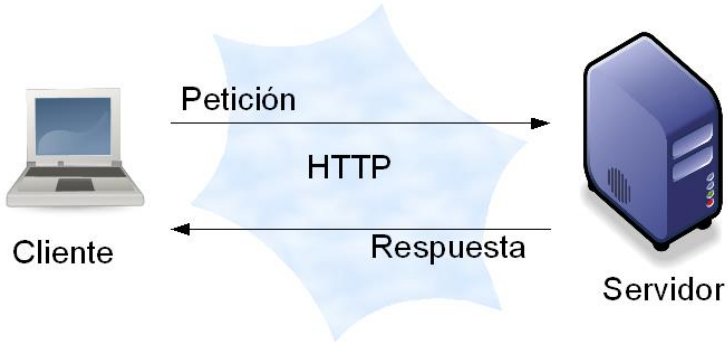

Figura 1. Diagrama del funcionamiento Modelo Cliente/Servidor.

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 cómo funciona este tipo de modelo.

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.

Modelo multicliente-servidor

El modelo **Multicliente/Servidor** es una extensión del modelo clásico Cliente/Servidor, en el cual un único servidor es capaz de gestionar múltiples conexiones simultáneas de distintos clientes. Esta arquitectura es ampliamente utilizada en aplicaciones que requieren interacción en tiempo real, como plataformas de mensajería, sistemas de bases de datos, videojuegos en línea y servicios web.

En este modelo, el servidor actúa como un punto central de procesamiento, recibiendo solicitudes de los clientes, procesándolas y enviando las respuestas correspondientes. Por su parte, los clientes son dispositivos o aplicaciones que establecen una conexión con el servidor para consumir los servicios ofrecidos. A diferencia de un sistema Cliente/Servidor simple, donde un servidor solo atiende a un cliente a la vez, en un entorno multicliente es necesario implementar mecanismos eficientes para manejar múltiples solicitudes de manera concurrente sin que el rendimiento del sistema se vea afectado.

Para lograr esta concurrencia, el servidor puede utilizar diferentes estrategias, como el uso de múltiples hilos (multithreading), la creación de procesos independientes (multiprocesamiento) o modelos asíncronos de comunicación. La elección de la estrategia depende de diversos factores, como

la cantidad de clientes esperados, la carga de trabajo de cada solicitud y los recursos disponibles en el servidor.

Uno de los principales retos en la implementación de un servidor multicliente es la correcta administración de los recursos compartidos, ya que múltiples clientes pueden intentar acceder a la misma información simultáneamente. Para evitar inconsistencias o bloqueos, es necesario emplear técnicas de sincronización que permitan garantizar la integridad de los datos y el correcto funcionamiento del sistema.

A pesar de los desafíos, el modelo Multicliente/Servidor ofrece ventajas significativas, como la escalabilidad y la centralización de la gestión de datos, lo que lo convierte en una solución eficiente para sistemas distribuidos que requieren atender a un gran número de usuarios al mismo tiempo.

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.

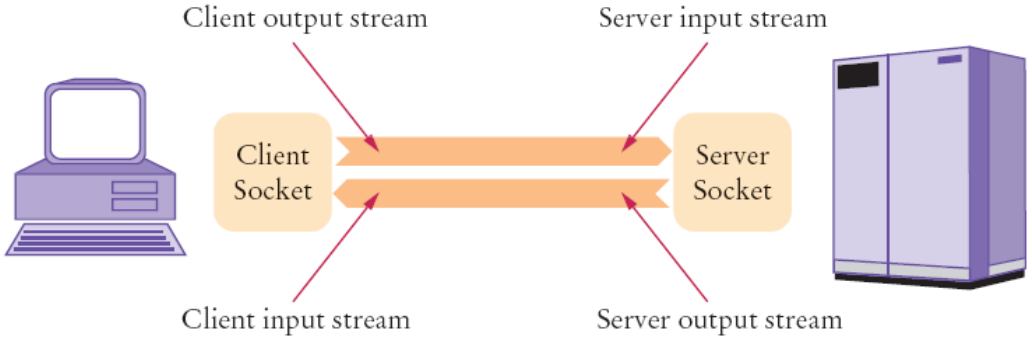

Figura 2. Cliente y Servidor conectados

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

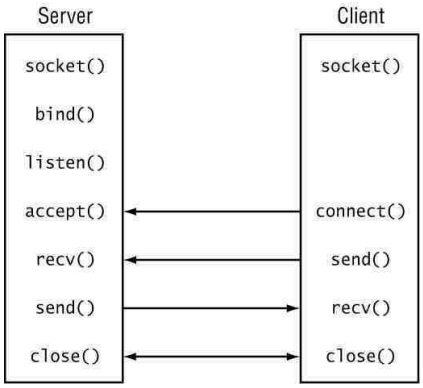

Figura 3. Secuencia de Conexión

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.

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

El modelo Cliente/Servidor tradicional, en el que un servidor atiende las solicitudes de un único cliente a la vez, presenta limitaciones cuando se requiere atender múltiples clientes simultáneamente. Esta restricción puede generar problemas de rendimiento, tiempos de espera prolongados y una experiencia de usuario deficiente, especialmente en sistemas donde las solicitudes de los clientes pueden ser concurrentes y requieren respuesta en tiempo real.

En este contexto, surge la necesidad de implementar un modelo Multicliente/Servidor, en el que un servidor sea capaz de gestionar múltiples conexiones de clientes de manera eficiente. La problemática principal radica en cómo garantizar que cada cliente reciba atención sin afectar el desempeño del sistema ni generar bloqueos en la comunicación. Para ello, se requiere un manejo adecuado de concurrencia, sincronización de recursos compartidos y estrategias que permitan distribuir la carga de trabajo del servidor.

En el caso específico de la simulación de una panadería digital, esta problemática se traduce en la gestión simultánea de múltiples clientes que desean consultar el stock de pan, realizar compras y recibir respuestas inmediatas del servidor. Si el servidor no administra correctamente la concurrencia, podrían generarse inconsistencias en el stock de pan o tiempos de espera excesivos para los clientes.

Esta práctica retoma la implementación previa del modelo Cliente/Servidor, en el cual un único cliente interactuaba con el servidor de la panadería. Sin embargo, en esta ocasión, se amplía la funcionalidad para que múltiples clientes puedan conectarse al servidor simultáneamente y realizar operaciones concurrentes. Para ello, se implementa un enfoque basado en hilos, donde cada cliente es gestionado por un hilo independiente dentro del servidor. Adicionalmente, se introduce un mecanismo de producción de pan en segundo plano, asegurando que el stock se reabastezca de manera automática y que los clientes que no puedan comprar de inmediato reciban una respuesta adecuada.

El desafío principal de este modelo es garantizar que el servidor maneje múltiples solicitudes sin afectar la consistencia del stock ni generar bloqueos innecesarios. Para ello, se emplean técnicas de sincronización en Java, como el uso de métodos sincronizados y notificaciones a los clientes en espera. Con esta implementación, se busca demostrar cómo el modelo Multicliente/Servidor mejora la escalabilidad y eficiencia del sistema en comparación con la versión anterior basada en un único cliente.

Propuesta de solución

Para abordar la problemática identificada en el modelo Cliente/Servidor tradicional, se propone una solución basada en un **modelo Multicliente/Servidor con concurrencia controlada**. La idea principal es permitir que múltiples clientes se conecten simultáneamente al servidor de la panadería, consulten el stock, realicen compras y reciban respuestas en tiempo real, sin afectar la consistencia de los datos ni el desempeño del sistema.

Elementos clave de la solución

Uso de un servidor concurrente con hilos: En lugar de manejar una única conexión de cliente a la vez, el servidor creará un hilo independiente para cada cliente que se conecte. Esto se logra mediante un ThreadPoolExecutor, que administra un conjunto de hilos y distribuye la carga de trabajo de manera eficiente. De este modo, varios clientes pueden interactuar con la panadería sin generar bloqueos o tiempos de espera excesivos.

Sincronización del acceso al stock de pan: Dado que múltiples clientes pueden intentar comprar pan al mismo tiempo, es fundamental garantizar la consistencia de los datos. Para ello, se emplean métodos sincronizados en Java, lo que evita que dos o más clientes accedan y modifiquen el stock simultáneamente, evitando inconsistencias en la cantidad de pan disponible.

Producción automática de pan mediante un hilo adicional: Para evitar que los clientes se queden sin pan de manera indefinida, se incorpora un hilo de producción que simula el horneado de pan en intervalos regulares. Este hilo, denominado HornoPanadero, se ejecuta en segundo plano y añade stock automáticamente cada cierto tiempo. Además, utiliza notifyAll() para despertar a los clientes que estaban esperando por pan, permitiendo que la panadería siga operando sin interrupciones.

Interacción con los clientes mediante comunicación en red: Los clientes y el servidor se comunican a través de **sockets TCP**, lo que permite el envío de mensajes entre ambas partes. El servidor recibe las solicitudes de los clientes (ver stock, comprar pan o salir) y responde con la información correspondiente en tiempo real.

Gestión eficiente de recursos: Para evitar el consumo excesivo de memoria y CPU, la cantidad de hilos en el servidor está limitada mediante un **ThreadPool**. Esto evita que la creación descontrolada de hilos degrade el rendimiento del sistema cuando hay muchos clientes conectados simultáneamente.

Beneficios de la solución

Mayor escalabilidad: Permite atender múltiples clientes sin bloquear la ejecución del servidor.

Consistencia en los datos: El uso de sincronización garantiza que el stock de pan refleje correctamente las transacciones realizadas por los clientes.

Automatización del reabastecimiento: El hilo del horno panadero mantiene el inventario sin intervención manual.

Respuesta en tiempo real: Los clientes reciben retroalimentación inmediata sobre el estado de sus compras o la disponibilidad de pan.

Con esta solución, se mejora significativamente la eficiencia del sistema en comparación con el modelo anterior de Cliente/Servidor, haciendo que la panadería pueda operar en un entorno con múltiples usuarios de manera fluida y sin inconsistencias.

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
            }

            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();
        }
    }
}
```

```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;
```

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();
        }
    }
}
```

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

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.");
```

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 panaderia... ¡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 demuestra su capacidad para gestionar múltiples clientes de manera simultánea, garantizando una interacción eficiente y sin conflictos con el servidor de la panadería. A continuación, se detallan los resultados obtenidos al ejecutar el código tanto en el servidor como en los clientes.

En primer lugar, al iniciar el servidor en el puerto 5000, este permanece a la espera de conexiones entrantes, además de iniciar el servidor ahora no en la dirección de “localhost”, sino en la dirección ip en mi caso de mi PC para poder ejecutarlo desde distintos equipos dentro de mi red local. Y cada vez que un cliente se conecta, el servidor acepta la conexión y crea un hilo independiente para manejar la interacción con dicho cliente. Esto permite que varios clientes se conecten de forma concurrente, sin afectar el desempeño del sistema. Además, el servidor ejecuta un hilo adicional que se encarga de reabastecer el stock de panes automáticamente cada cierto tiempo.

Esto puede observarse en la siguiente figura:

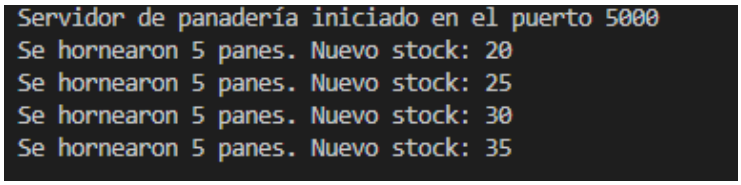

Figura 10. Servidor iniciado y reabastecimiento de pan.

Cuando un cliente ejecuta su código y la conexión se establece correctamente, el sistema muestra un mensaje de confirmación y solicita al usuario que ingrese su nombre. Una vez proporcionado, el cliente recibe un menú de opciones, permitiéndole interactuar con la panadería de la siguiente manera:

1. Ver stock de pan: Al seleccionar esta opción, el servidor envía la cantidad actual de panes disponibles, asegurando que la información proporcionada sea precisa y actualizada.
2. Comprar pan: Si el usuario elige comprar pan, se solicita la cantidad deseada.
    - Si hay suficiente stock, la compra se procesa correctamente, y el servidor actualiza la cantidad restante.
    - Si el stock es insuficiente, el cliente recibe un mensaje de espera, indicándole que debe aguardar hasta que el horno produzca más panes.
3. Salir: Al elegir esta opción, el cliente se desconecta del servidor y la conexión se cierra de manera ordenada.

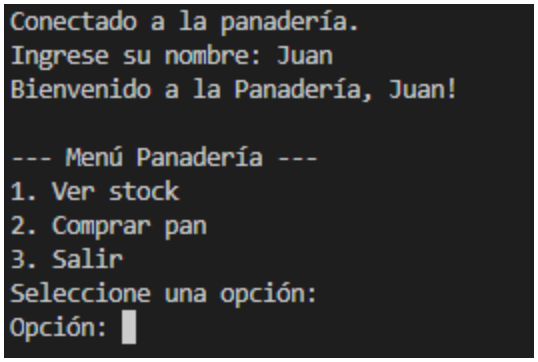

Figura 11. Conexión desde el cliente.

El servidor maneja estas interacciones mediante **sincronización**, evitando condiciones de carrera y asegurando que el stock se **actualice correctamente** sin generar inconsistencias cuando múltiples clientes intentan comprar pan al mismo tiempo. Esto puede observarse en los siguientes mensajes registrados en el servidor:

```
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:

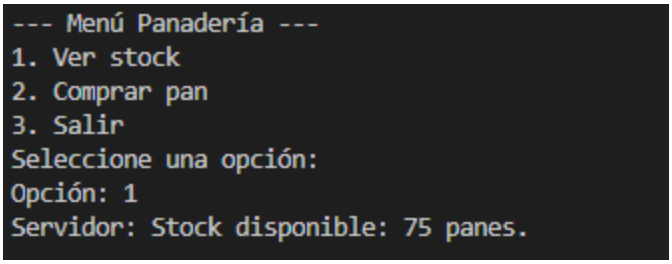

Figura 15. Ver Stock desde el cliente.

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.

Para comprobar la capacidad concurrente del sistema, se realizaron conexiones desde múltiples clientes de manera simultánea. Como se puede observar en las figuras anteriores, se establecieron dos conexiones distintas, permitiendo que ambos clientes consultaran el stock y realizaran compras sin interferencias. Sin embargo, ahora de igual forma lo hicimos desde dos equipos distintos, al estar dentro de la misma red, únicamente teniendo el código del cliente desde otro equipo podemos conectarnos al servidor, además de añadirse una opción para observar la dirección ip desde donde el cliente se está conectando, como se ve en la siguiente figura:

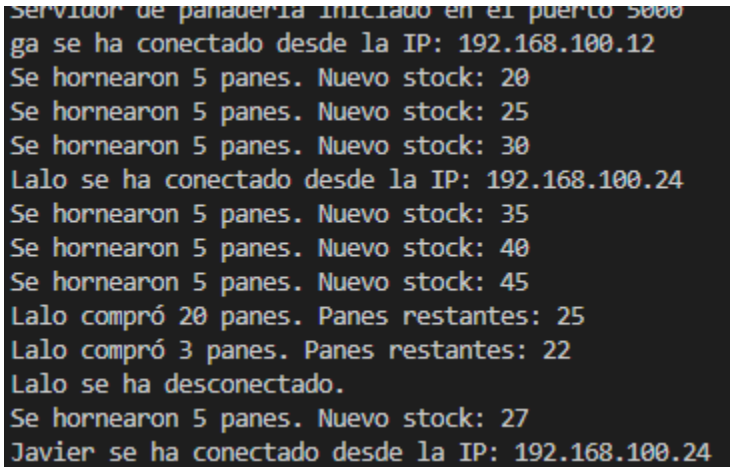

Figura 17. Conexión con múltiples clientes.

Y para obtener la dirección IP del cliente en el servidor, utilizamos el objeto Socket que representa la conexión entre el servidor y el cliente. Este objeto tiene un método llamado getInetAddress(), que devuelve un objeto InetAddress que contiene la dirección IP del cliente. Luego, usamos el método getHostAddress() para obtener esa dirección IP en formato de cadena (por ejemplo, "192.168.1.5").

En el código, al aceptar una conexión de un cliente, primero leemos su nombre a través de un BufferedReader, luego obtenemos la dirección IP con socket.getInetAddress().getHostAddress(). Después, imprimimos tanto el nombre del cliente como la dirección IP en la consola del servidor para que el administrador pueda ver desde qué IP se está conectando el cliente.

El cambio que realizamos fue agregar estas líneas dentro del hilo que maneja la conexión de cada cliente, en el método run() de la clase HiloCliente. Esto nos permite mostrar en la consola un mensaje como "ga se ha conectado desde la IP: 192.168.100.12".

```
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();
        String ipCliente = socket.getInetAddress().getHostAddress(); // Obtener la IP del cliente
        System.out.println(nombreCliente + " se ha conectado desde la IP: " + ipCliente); // Mostrar IP
        out.println("Bienvenido a la Panadería, " + nombreCliente + "!");
```

Figura 18. Método para obtener la dirección ip del cliente.

De igual forma, si algún cliente intenta hacer compras simultáneamente, gracias a synchronized, el servidor garantiza que solo un cliente pueda acceder y modificar el stock de pan en un momento dado. Esto evita que dos clientes puedan comprar más pan del que realmente está disponible. Si varios clientes intentan realizar una compra al mismo tiempo, el método comprarPan() se ejecutará de manera exclusiva para cada hilo, es decir, el segundo cliente tendrá que esperar hasta que el primero termine su compra y libere el acceso al método.

El uso de synchronized asegura que el acceso al stock de pan sea secuencial y seguro, lo que significa que no se producirán errores de concurrencia, como la compra de más pan del que hay en existencia, incluso si múltiples clientes están interactuando con el servidor al mismo tiempo. Y lo podemos ver en la siguiente Figura, aunque no se ve a detalle, pero realice compras en un equipo y en otro al mismo tiempo.

```
Javier se ha conectado desde la IP: 192.168.100.24
Se hornearon 5 panes. Nuevo stock: 32
Javier compró 21 panes. Panes restantes: 11
Se hornearon 5 panes. Nuevo stock: 16
Se hornearon 5 panes. Nuevo stock: 21
ga compró 6 panes. Panes restantes: 15
ga compró 5 panes. Panes restantes: 10
```

Figura 19. Compras de pan con múltiples clientes.

Con esto pudimos observar finalmente que, al utilizar synchronized, se logra un control adecuado sobre las operaciones concurrentes en el servidor. Esto garantiza que el sistema pueda manejar

múltiples clientes que intentan realizar compras al mismo tiempo, sin comprometer la consistencia del stock de pan. De este modo, evitamos condiciones de carrera y aseguramos que los clientes solo puedan comprar la cantidad de pan disponible, lo que mejora la fiabilidad y la experiencia del usuario al interactuar con el servidor.

Conclusión

En esta práctica, pude profundizar aún más en el modelo cliente-servidor, pero con la particularidad de manejar múltiples clientes simultáneamente. A pesar de que el código base es muy similar al de la práctica anterior, con la estructura básica de servidor y cliente implementada, se introdujeron mejoras clave, como la gestión de conexiones concurrentes, la muestra de la dirección IP desde donde se conecta cada cliente y el uso de un pool de hilos para manejar varias solicitudes al mismo tiempo. Estas modificaciones me permitieron mejorar la capacidad del servidor para manejar de manera eficiente la comunicación con varios clientes sin que se produjeran bloqueos o errores debido a la concurrencia.

Al incorporar múltiples clientes, pude observar cómo el servidor interactúa con varios usuarios al mismo tiempo, lo cual es fundamental en sistemas distribuidos reales. Además, la sincronización de recursos, como el stock de pan en la simulación de la panadería, sigue siendo esencial para evitar conflictos, asegurando que cada cliente reciba un trato justo y que no se presenten condiciones de carrera.

Aunque la base del código no cambió significativamente con respecto a la práctica anterior, la adaptación al manejo de múltiples clientes me permitió entender de manera más clara los desafíos y ventajas de los sistemas distribuidos. La utilización de un pool de hilos para manejar varias conexiones simultáneas fue un paso importante para mejorar la escalabilidad y la eficiencia del servidor. Este enfoque refleja cómo los sistemas evolucionan para ser capaces de servir a muchos usuarios de forma concurrente, lo cual es fundamental para aplicaciones reales de alto tráfico, y algo que hemos visto en clase también, el hecho de estar viendo como fueron evolucionando los distintos modelos y como se les fueron encontrando fallas.

En general, esta práctica no solo reforzó mis conocimientos sobre la implementación de procesos e hilos, sino que también me permitió entender cómo estos conceptos se aplican de manera efectiva en un entorno multicliente. Al trabajar con varios clientes conectados simultáneamente, pude visualizar mejor cómo se gestionan los recursos compartidos y cómo se asegura la correcta sincronización en un sistema distribuido. Esto me ha motivado a seguir explorando cómo mejorar la eficiencia en la comunicación y gestión de recursos en sistemas más complejos y escalables.

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.
- **J. Geeks**, "Programación de servidores multiclientes en Java," Geeks for Geeks, 2023. [En línea]. Available: https://www.geeksforgeeks.org/multithreaded-server-in-java. [Último acceso: 2 Marzo 2025].

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 = "192.168.100.12"; // 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();
            String ipCliente = socket.getInetAddress().getHostAddress(); //
Obtener la IP del cliente
            System.out.println(nombreCliente + " se ha conectado desde la
IP: " + ipCliente); // Mostrar IP
            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();
            }
        }
```

```
    }
}


// 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();
        }
    }
}
```