

Pontificia Universidad Javeriana

Informe: Paralelismo



Gabriel Jaramillo Cuberos
Roberth Santiago Méndez
Mariana Osorio
Juan Esteban Vera

Asignatura: Introducción a los Sistemas Distribuidos

Profesor: John Jairo Corredor Franco

Septiembre de 2025

Parte 1: Socket Mailbox

1. Resumen

Estos programas en Java demuestran la comunicación por sockets, utilizando los protocolos TCP y UDP. El código TCP es un cliente (sockettcpcli.java) que establece una conexión confiable con un servidor en el puerto 6001. Envía mensajes secuencialmente, asegurando su correcta entrega. Por otra parte, el código UDP incluye un cliente (socketudpcli.java) y un servidor (socketudpser.java). Este protocolo es sin conexión, lo que permite una comunicación más rápida al enviar datagramas de forma independiente. El cliente envía paquetes al puerto 6000, y el servidor los recibe y procesa, sin la sobrecarga de una conexión persistente. Ambos conjuntos de programas terminan al recibir el mensaje "fin".

1.1 Planteamiento del problema

El problema principal que estos códigos resuelven es la implementación de comunicación básica entre un cliente y un servidor utilizando dos protocolos de red fundamentales: TCP y UDP.

Para el protocolo TCP

- Entrada: Un flujo continuo de mensajes de texto enviados desde el cliente (sockettcpcli.java).
- Objetivo: Establecer una conexión confiable entre un cliente y un servidor, permitiendo un envío de mensajes secuencial y seguro.
- Requisitos:
 - El cliente debe conectarse a un servidor específico en el puerto 6001.
 - La comunicación debe ser bidireccional, aunque la implementación solo muestra el flujo del cliente al servidor.
 - El cliente debe leer la entrada del usuario de forma continua y enviar cada línea como un mensaje.
 - La comunicación debe finalizar cuando el cliente envíe un mensaje que comience con la palabra "fin".

Para el protocolo UDP

- Entrada: Datagramas de texto enviados desde el cliente (socketudpcli.java).
- Objetivo: Demostrar una comunicación sin conexión, donde los paquetes de datos (datagramas) se envían de forma independiente entre el cliente y el servidor.
- Requisitos:
 - El cliente debe enviar datagramas al puerto 6000 del servidor.
 - El servidor debe escuchar de forma pasiva en el puerto 6000 para recibir estos datagramas.

Cada mensaje se envía en un datagrama individual, sin necesidad de una conexión previa.

El servidor debe procesar y mostrar cada datagrama recibido. La comunicación de recepción del servidor debe finalizar cuando reciba un datagrama que contenga el mensaje "fin".

1.2 Propuesta de solución

Para el problema se plantean 2 soluciones, una a partir del protocolo TCP y la otra a partir del protocolo UDP.

Protocolo TCP (Orientado a la conexión):

La implementación TCP se basa en una conexión punto a punto, lo que garantiza un flujo de datos confiable, secuencial y con detección de errores.

- Cliente TCP: sockettcpcli.java

Este programa actúa como un cliente que establece una conexión con un servidor en el puerto 6001 para enviar datos. Su ejecución se da en los siguientes pasos:

1. Validación de argumentos: El programa requiere un argumento de línea de comandos que especifique la dirección IP o el nombre de host del servidor. Si no se proporciona, el programa termina con un mensaje de error.
2. Resolución de dirección: Utiliza `InetAddress.getByName()` para resolver la dirección del servidor, lo que convierte el nombre del host en una dirección IP.
3. Establecimiento de la conexión: Se crea un objeto `Socket` instanciando `new Socket(address, 6001)`. Esta llamada inicia el proceso de conexión, lo que resulta en un enlace virtual y bidireccional entre el cliente y el servidor.
4. Flujo de datos: Se obtiene un `DataOutputStream` del socket (`socket.getOutputStream()`) para enviar datos. El programa entra en un bucle `do-while` que lee líneas de texto de la entrada estándar (`System.in`) y las envía al servidor utilizando `out.writeUTF()`.
5. Finalización: El bucle se mantiene activo hasta que el mensaje enviado comienza con la cadena "fin", momento en el que el cliente termina su ejecución. Las excepciones de entrada/salida (`IOException`) son manejadas para garantizar una terminación controlada en caso de fallos de red o del socket.

- Servidor TCP: sockettcpser.java

Este programa se comporta como un servidor que se enlaza al puerto 6001 y espera conexiones de clientes TCP. Una vez que acepta una conexión, lee los mensajes enviados por el cliente y los muestra en la consola. Este es el complemento directo del cliente `sockettcpcli.java`.

Flujo de ejecución:

1. Creación y vinculación del socket: El servidor crea un `ServerSocket` en el puerto 6001 (`new ServerSocket(6001)`). Este objeto se encarga de escuchar y aceptar las peticiones de conexión entrantes de los clientes.
2. Espera de conexión: La llamada a `socket = serverSocket.accept()` es una operación de bloqueo. El programa se detiene y espera indefinidamente hasta que un cliente intente conectarse. Cuando un cliente se conecta, `accept()` devuelve un nuevo objeto `Socket`, que se utiliza para la comunicación con ese cliente específico.
3. Flujo de datos: Se crea un `DataInputStream` a partir del flujo de entrada del socket (`socket.getInputStream()`). Este flujo permite al servidor leer los mensajes que el cliente envía.
4. Recepción de mensajes: El servidor entra en un bucle `do-while` para leer continuamente los mensajes del cliente utilizando `in.readUTF()`.
5. Procesamiento y finalización: Cada mensaje recibido se imprime en la consola. El bucle se mantiene activo hasta que el servidor recibe un mensaje que comienza con la cadena "fin". En ese momento, la variable booleana `fin` se establece en `true`, lo que permite que el bucle termine. Una vez finalizado el bucle, el servidor cierra los sockets (`socket.close()` y `serverSocket.close()`) para liberar los recursos del sistema, asegurando una terminación limpia.

Protocolo UDP (Sin conexión):

La implementación UDP es un protocolo sin conexión, lo que implica que la comunicación se realiza mediante el envío de datagramas de forma independiente. Este método es más rápido, pero carece de la confiabilidad y el control de flujo de TCP.

- Cliente UDP: `socketudpcli.java`

Este programa es un cliente que envía datagramas al servidor en el puerto 6000.

Flujo de ejecución:

1. Validación de argumentos: Al igual que el cliente TCP, requiere la dirección del servidor como argumento.
2. Creación del socket: Se crea un `DatagramSocket` para el envío de datagramas. A diferencia de TCP, este socket no establece una conexión persistente.
3. Encapsulación de datos: Dentro de un bucle `do-while`, se leen mensajes de la consola. Cada mensaje se convierte en un array de bytes y se encapsula en un objeto `DatagramPacket`.
4. Envío del datagrama: El `DatagramPacket` se construye con los datos, la dirección del servidor y el puerto de destino (6000), y se envía mediante `socket.send()`.
5. Finalización: El programa continúa enviando datagramas hasta que el mensaje introducido comience con "fin".

- Servidor UDP: `socketudpser.java`

Este programa es el servidor que escucha y recibe datagramas en el puerto 6000.

Flujo de ejecución:

1. Vinculación del socket: Se crea un DatagramSocket y se vincula al puerto 6000 (new DatagramSocket(6000)). Esto permite al servidor escuchar y recibir tráfico UDP en ese puerto específico.
2. Recepción de datagramas: El servidor entra en un bucle de recepción continuo. Dentro del bucle:
 - a. Se prepara un DatagramPacket vacío, con un buffer de 256 bytes, para recibir el datagrama entrante.
 - b. La llamada a socket.receive(paquete) bloquea la ejecución del programa, esperando la llegada de un datagrama.
3. Procesamiento de datos: Una vez que se recibe un paquete, los datos (byte[]) se extraen y se convierten en una cadena de caracteres. El mensaje es impreso en la consola.
4. Terminación: La recepción de paquetes continúa hasta que uno de los mensajes recibidos comience con "fin", momento en el que el servidor termina su ejecución.

Al igual que para el programa TCP, se realizan dos pruebas, la prueba udp local:

Y luego la prueba remota, donde, de igual manera, la ip del servidor es el argumento de la ejecución del cliente.

1.3 Método de prueba

Las pruebas se realizarán para verificar la correcta comunicación entre los clientes y servidores, tanto en un entorno de red local como en uno remoto. Se verificará que cada protocolo se comporte según lo esperado (TCP para conexión confiable y UDP para envío).

Para cada protocolo se realiza:

- Prueba local: en la misma máquina se crea un servidor y un cliente, se envían mensajes y se comprueba que lleguen al servidor.
- Prueba remota: Para esta prueba, se hace con la IP del servidor. Para ejecutar el cliente, debe escribirse la ip como argumento.

1.4 Resultados esperados

Después de la ejecución del programa se espera lo siguiente:

- Salida en el cliente: Nada es mostrado en pantalla.
- Salida en el servidor: Registro de conexión del cliente y el mensaje procesado.

2. Diseño

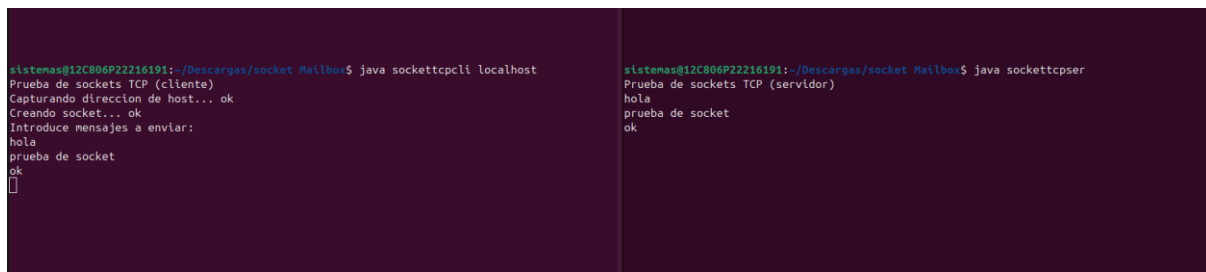
2.1 Diseño general

- El programa está escrito en Java.
- El servidor utiliza:
 - ServerSocket para aceptar conexiones.
 - DataInputStream para la comunicación.
- El cliente utiliza:
 - Socket para conectarse al servidor.
 - Entrada por consola (BufferedReader).
 - Flujos de datos para enviar y recibir mensajes.
 - DataOutputStream para la comunicación.

2.2 Comunicación cliente-servidor

- El cliente envía un mensaje en formato texto al servidor.
- El servidor no responde, solo recibe el mensaje.
- La comunicación se mantiene hasta que el cliente envía el mensaje "fin."

3. Resultados



```
sistemas@12CB06P22216191: ~/Descargas/socket Mailbox$ java sockettcpcli localhost
Prueba de sockets TCP (cliente)
Capturando direccion de host... ok
Creando socket... ok
Introduce mensajes a enviar:
hola
prueba de socket
ok
█

sistemas@12CB06P22216191: ~/Descargas/socket Mailbox$ java sockettcpser
Prueba de sockets TCP (servidor)
hola
prueba de socket
ok
```

Figura 1: Ejecución cliente servidor TCP local

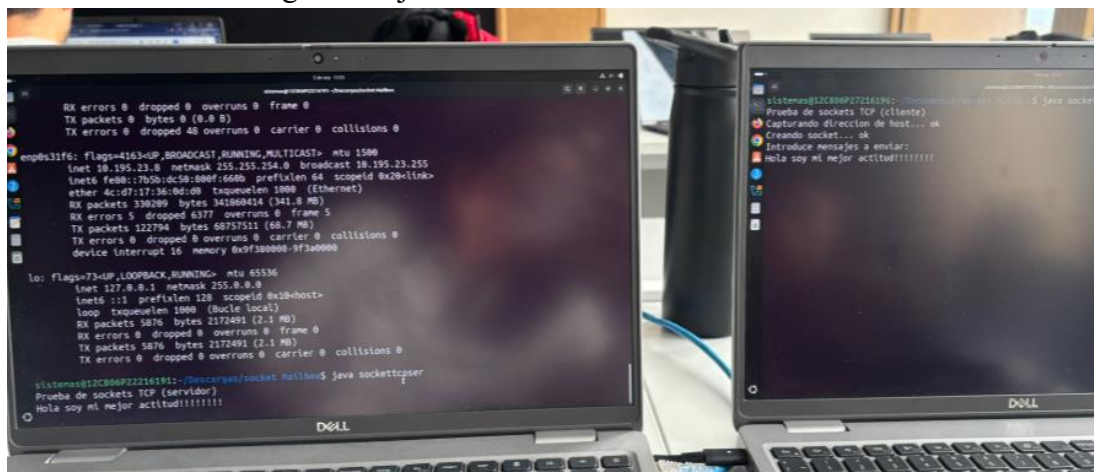


Figura 2: Ejecución cliente servidor TCP remoto

```
5 de sep 15:33
sistemas@12C806P22216191: ~/Descargas/socket Mailbox

RX errors 0 dropped 0 overruns 0 frame 0
TX packets 0 bytes 0 (0.0 B)
TX errors 0 dropped 48 overruns 0 carrier 0 collisions 0

enp0s31f6: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
inet 10.195.23.8 netmask 255.255.254.0 broadcast 10.195.23.255
inet6 fe80::7b5b:dc50:800f:660b prefixlen 64 scopeid 0x20<link>
ether 4c:d7:17:36:0d:d0 txqueuelen 1000 (Ethernet)
RX packets 330209 bytes 341860414 (341.8 MB)
RX errors 5 dropped 6377 overruns 0 frame 5
TX packets 122794 bytes 68757511 (68.7 MB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
device interrupt 16 memory 0x9f380000-9f3a0000

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
inet 127.0.0.1 netmask 255.0.0.0
inet6 ::1 prefixlen 128 scopeid 0x10<host>
loop txqueuelen 1000 (Bucle local)
RX packets 5876 bytes 2172491 (2.1 MB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 5876 bytes 2172491 (2.1 MB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

sistemas@12C806P22216191:~/Descargas/socket Mailbox$ java sockettcpser
Prueba de sockets TCP (servidor)
Hola soy mi mejor actitud!!!!!!!
^Csistemas@12C806P22216191:~/Descargas/socket Mailbox$
```

Figura 3: Ejecución servidor TCP remoto

```
sistemas@12C806P27216196: ~/Documentos/socket Mailbox$ java sockettcpcli 10.195.23.8
Prueba de sockets TCP (cliente)
Capturando direccion de host... ok
Creando socket... ok
Introduce mensajes a enviar:
Hola soy mi mejor actitud!!!!!!!

```

Figura 4: Ejecución cliente TCP remoto

```
sistemas@12C806P22216191: ~/Descargas/socket Mailbox$ java socketudpser
Prueba de sockets UDP (servidor)
Creando socket... ok
Recibiendo mensajes...
prueba 2
hola

sistemas@12C806P22216191: ~/Descargas/socket Mailbox$ java socketudpcli localhost
Prueba de sockets UDP (cliente)
Creando socket... ok
Capturando direccion de host... ok
Introduce mensajes a enviar:
prueba 2
hola

```

Figura 5: Ejecución cliente servidor UDP local

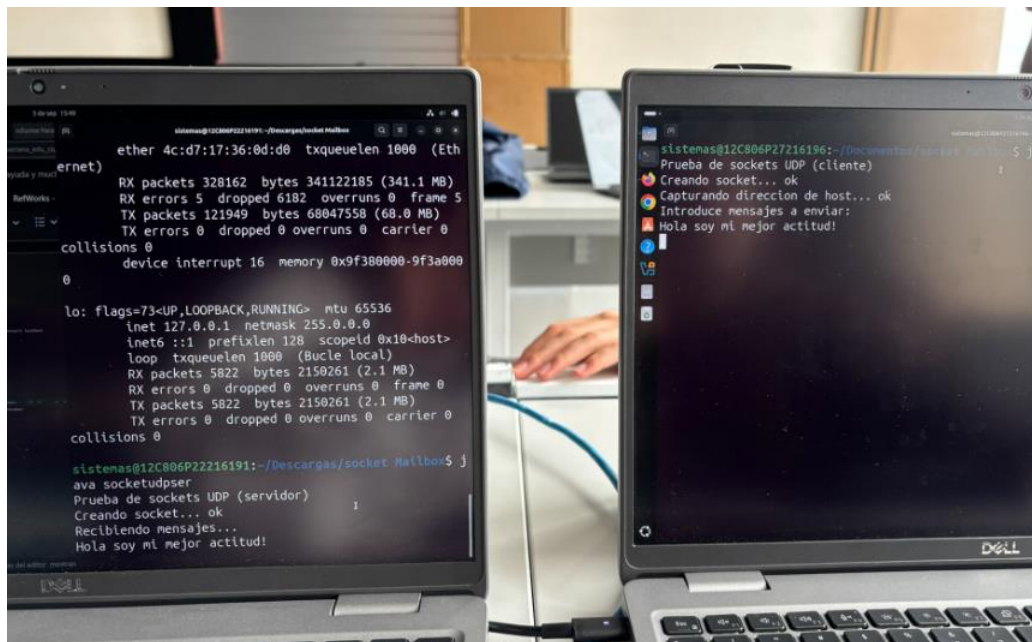


Figura 6: Ejecución cliente servidor UDP remoto

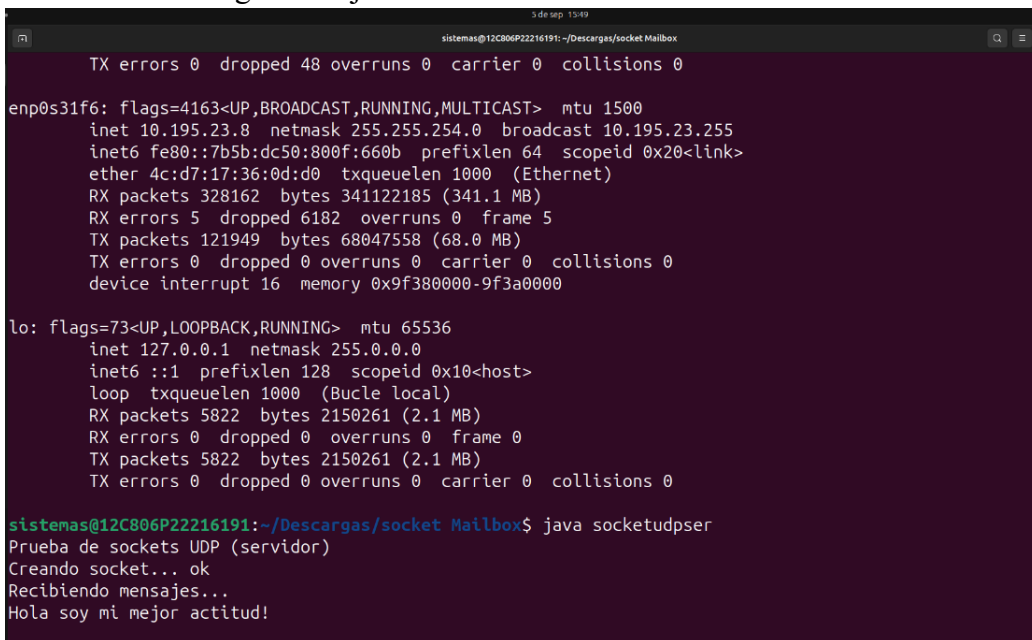


Figura 7: Ejecución servidor UDP remoto

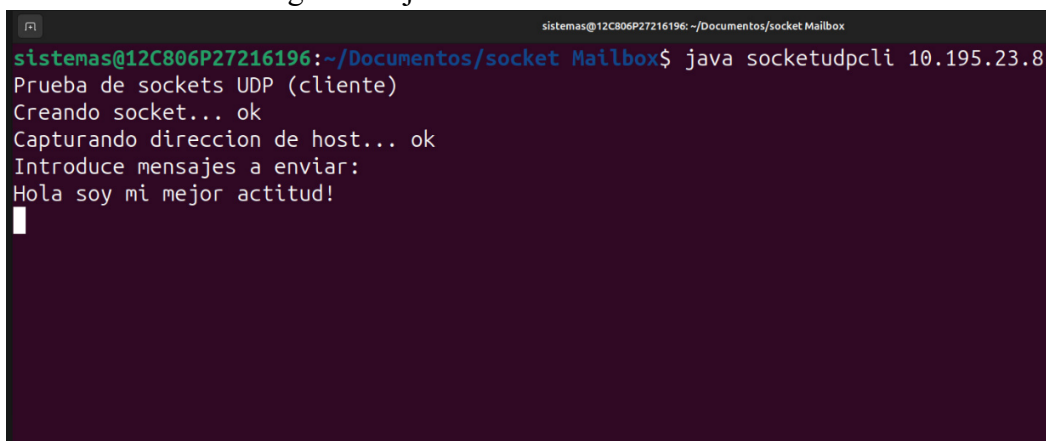


Figura 8: Ejecución cliente UDP remoto

4. Análisis de resultados

El análisis de los resultados de las pruebas locales y remotas confirma que las implementaciones de los protocolos TCP y UDP funcionan según lo esperado.

Para el protocolo TCP, las pruebas locales y remotas (Figura 1 y Figura 2) mostraron que el cliente se conecta exitosamente al servidor. El flujo de mensajes se mantuvo secuencial y confiable, sin pérdidas de datos. El servidor (Figura 3) esperó la conexión, y el cliente (Figura 4) estableció el enlace para el envío de mensajes.

En cuanto al protocolo UDP, los resultados de las pruebas (Figura 5 y Figura 6) también validaron el comportamiento esperado. La comunicación se estableció sin necesidad de un enlace de conexión formal. Cada datagrama enviado por el cliente (Figura 8) fue recibido por el servidor (Figura 7), demostrando la capacidad del protocolo para transmitir datos de forma independiente. A diferencia de TCP, UDP no garantiza la entrega ni el orden, pero para el caso de uso de envío de mensajes simples, esta implementación demostró ser eficiente y funcional. La prueba remota confirmó que el sistema opera adecuadamente a través de una red distribuida, lo cual valida que la implementación de sockets está correctamente configurada para la comunicación inter-máquinas.

5. Conclusiones

La implementación y las pruebas realizadas han permitido concluir lo siguiente:

- Diferencia de protocolos: La experiencia práctica con los códigos confirma la diferencia fundamental entre los protocolos TCP y UDP. Mientras TCP es ideal para aplicaciones que requieren fiabilidad y orden (como la transferencia de archivos o el correo electrónico), UDP es más adecuado para aplicaciones donde la velocidad es crítica y la pérdida ocasional de datos es aceptable (como el streaming de video o los videojuegos en línea).
- Funcionalidad de Sockets en Java: Las bibliotecas `java.net` y `java.io` son herramientas robustas y eficientes para el desarrollo de aplicaciones de red. La capacidad de utilizar `ServerSocket` para TCP y `DatagramSocket` para UDP demuestra la flexibilidad de la plataforma para adaptarse a diferentes necesidades de comunicación.
- Limitaciones y mejoras: Los códigos presentados son un excelente punto de partida para comprender los fundamentos de los sockets. Sin embargo, para una implementación más robusta, sería necesario incorporar un modelo multihilo en el servidor TCP para poder atender a varios clientes de forma concurrente, evitando bloqueos. Esto escalaría la solución a escenarios más complejos y realistas. En el caso de UDP, se podría considerar la implementación de una capa de aplicación que maneje la confirmación de recepción y la retransmisión de paquetes si se necesitara una mayor fiabilidad.

- Resultados de las pruebas: Las pruebas, tanto locales como remotas, confirmaron que los programas cumplen con los objetivos de comunicación establecidos. Los códigos son funcionales y demuestran una comprensión clara de la lógica subyacente de cada protocolo.

Parte 2: ThreadsJar

1. Resumen

Este programa en Java tiene como simula la atención de clientes en un supermercado utilizando procesamiento secuencial y paralelo mediante hilos. El sistema cuenta con clases que representan clientes y cajeras. Cada cliente tiene un carrito de compras con productos, donde cada producto tiene un tiempo de procesamiento asociado. Hay dos versiones: una secuencial, donde una sola cajera procesa a todos los clientes uno tras otro, y otra concurrente, donde varias cajeras procesan simultáneamente a distintos clientes gracias al uso de multi hilos. Esto permite observar cómo el paralelismo reduce el tiempo total de atención.

1.1 Planteamiento del problema

El problema para resolver consiste en simular la atención de clientes en un supermercado, donde cada cajera debe procesar los productos de los clientes. Cada producto requiere un tiempo específico de registro, y mientras más productos tenga un cliente, mayor será el tiempo necesario. En una solución secuencial, una cajera procesa a un cliente completo antes de atender al siguiente, lo que provoca tiempos de espera elevados. El objetivo es implementar una solución que permita que varias cajeras trabajen simultáneamente, reduciendo los tiempos de espera totales mediante el uso de hilos.

Entrada:

- Listado de clientes, cada uno con un arreglo de tiempos que representan el procesamiento de cada producto.

Salida:

- Mensajes en consola que muestran el inicio, progreso y finalización del procesamiento de cada cliente, junto con el tiempo transcurrido.

Requisitos:

- Implementar una versión secuencial y otra paralela.
- Cada cajera debe operar como un hilo independiente.

- Mostrar tiempos de procesamiento para comparar la eficiencia.

1.2 Propuesta de solución

Se propusieron dos enfoques para resolver el problema:

Versión secuencial: Una cajera procesa completamente la compra de un cliente antes de iniciar con el siguiente. Esto se implementa en la clase Cajera, donde se utiliza el método procesarCompra(). Cada producto del carrito se procesa simulando el tiempo de registro mediante Thread.sleep().

Versión paralela: Se implementa el uso de múltiples hilos para que varias cajeras atiendan simultáneamente a distintos clientes. Se presentan dos formas de crear hilos en Java:

1. Heredando de la clase Thread (CajeraThread).
2. Implementando la interfaz Runnable (MainRunnable).

Ambas soluciones permiten ejecutar múltiples procesos al mismo tiempo, optimizando el tiempo total de atención.

1.3 Método de prueba

Para probar el sistema se crearon dos clientes:

- Cliente 1 con los tiempos de productos: {2, 2, 1, 5, 2, 3}
- Cliente 2 con los tiempos de productos: {1, 3, 5, 1, 1}

En el escenario secuencial, una cajera atiende a ambos clientes en orden, sumando los tiempos totales de todos los productos. En el escenario paralelo, dos cajeras procesan simultáneamente, lo que reduce el tiempo total.

Se midió el tiempo transcurrido desde el inicio hasta la finalización de cada cliente.

1.4 Resultados esperados

Se espera que en la versión secuencial el tiempo total sea la suma de todos los tiempos de los productos, mientras que en la versión paralela este tiempo se reduzca significativamente al ejecutar las tareas en simultáneo. En consola se visualizarán mensajes indicando el progreso de cada cajera, con tiempos menores para la versión paralela.

2. Diseño

2.1 Diseño general

El sistema está compuesto por las siguientes clases:

- Cliente: representa a un cliente con un nombre y un arreglo de productos con tiempos de procesamiento.
- Cajera: procesa a los clientes de forma secuencial.
- CajeraThread: representa una cajera que opera como un hilo independiente.
- MainThread: clase principal para ejecutar el programa con hilos.
- MainRunnable: implementa la lógica de procesamiento usando la interfaz Runnable.

2.2 Flujo de ejecución

1. Se crean los clientes y sus carritos de compras.
2. Se crean las cajeras (ya sea como objetos normales o como hilos).
3. Se inicia el proceso de compra:
 - a. En la versión secuencial, una cajera procesa a un cliente y luego al otro.
 - b. En la versión paralela, cada cajera procesa simultáneamente a un cliente.
4. El programa imprime los tiempos de inicio, progreso y finalización de cada cliente.

3. Resultados

En las pruebas realizadas, se observaron los siguientes resultados:

```
juane@juanes: /mnt/c/Users/juane/Downloads/ThreadsJarroba-master/ThreadsJarroba-master/src$ java threadsJarroba.MainRunnable
La cajera Cajera 1 comienza a procesar la compra del cliente Cliente 1 en el tiempo: 0seg
La cajera Cajera 2 comienza a procesar la compra del cliente Cliente 2 en el tiempo: 0seg
Procesado el producto 1 del cliente Cliente 2 ->Tiempo: 1seg
Procesado el producto 1 del cliente Cliente 1 ->Tiempo: 2seg
Procesado el producto 2 del cliente Cliente 2 ->Tiempo: 4seg
Procesado el producto 2 del cliente Cliente 1 ->Tiempo: 4seg
Procesado el producto 3 del cliente Cliente 1 ->Tiempo: 5seg
Procesado el producto 3 del cliente Cliente 2 ->Tiempo: 9seg
Procesado el producto 4 del cliente Cliente 1 ->Tiempo: 10seg
Procesado el producto 4 del cliente Cliente 2 ->Tiempo: 10seg
Procesado el producto 5 del cliente Cliente 2 ->Tiempo: 11seg
La cajera Cajera 2 ha terminado de procesar Cliente 2 en el tiempo: 11seg
Procesado el producto 5 del cliente Cliente 1 ->Tiempo: 12seg
Procesado el producto 6 del cliente Cliente 1 ->Tiempo: 15seg
La cajera Cajera 1 ha terminado de procesar Cliente 1 en el tiempo: 15seg
```

Figure 9. Procesamiento simultáneo con varias cajeras

```

juane@juanes: /mnt/c/Users/juane/Downloads/ThreadsJarroba-master/ThreadsJarroba-master/src$ java threadsJarroba.Main
La cajera Cajera 1 comienza a procesar la compra del cliente Cliente 1 en el tiempo: 0seg
Procesado el producto 1 del cliente Cliente 1 ->Tiempo: 2seg
Procesado el producto 2 del cliente Cliente 1 ->Tiempo: 2seg
Procesado el producto 3 del cliente Cliente 1 ->Tiempo: 3seg
Procesado el producto 4 del cliente Cliente 1 ->Tiempo: 8seg
Procesado el producto 5 del cliente Cliente 1 ->Tiempo: 10seg
Procesado el producto 6 del cliente Cliente 1 ->Tiempo: 13seg
La cajera Cajera 1 ha terminado de procesar Cliente 1 en el tiempo: 13seg
La cajera Cajera 2 comienza a procesar la compra del cliente Cliente 2 en el tiempo: 13seg
Procesado el producto 1 del cliente Cliente 2 ->Tiempo: 14seg
Procesado el producto 2 del cliente Cliente 2 ->Tiempo: 17seg
Procesado el producto 3 del cliente Cliente 2 ->Tiempo: 22seg
Procesado el producto 4 del cliente Cliente 2 ->Tiempo: 23seg
Procesado el producto 5 del cliente Cliente 2 ->Tiempo: 24seg
La cajera Cajera 2 ha terminado de procesar Cliente 2 en el tiempo: 24seg
juane@juanes: /mnt/c/Users/juane/Downloads/ThreadsJarroba-master/ThreadsJarroba-master/src$

```

Figure 10. Procesamiento secuencial de los productos

La consola mostró mensajes indicando cómo cada cajera trabajaba de manera independiente en paralelo.

4. Análisis de resultados

El análisis evidencia que el uso de hilos permite optimizar los tiempos de ejecución cuando varias tareas pueden realizarse de manera concurrente. En la versión secuencial, mientras una cajera procesaba a un cliente, el otro debía esperar. Por otro lado, en la versión paralela, cada cliente fue atendido simultáneamente por cajeras distintas.

Esto muestra un ejemplo de cómo la concurrencia mejora el rendimiento en sistemas distribuidos o en problemas donde múltiples procesos pueden ejecutarse en paralelo.

5. Conclusiones

1. El proyecto demostró la diferencia entre procesamiento secuencial y concurrente mediante hilos.
2. El uso de la clase Thread y la interfaz Runnable son dos métodos efectivos para implementar paralelismo en Java.
3. La simulación evidenció que el paralelismo reduce los tiempos de espera totales, mejorando la eficiencia.
4. Este ejemplo es aplicable a escenarios reales como cajas registradoras, sistemas bancarios, servidores web, entre otros.