



INSTITUTO POLITÉCNICO NACIONAL

---

---

Escuela Superior de Cómputo

**Carrera:**

Ingeniería en Sistemas Computacionales

**Unidad de aprendizaje:**

Sistemas Distribuidos

**Practica 4:**

Arquitectura Cliente-Servidor Escalable

**Alumno:**

Cardoso Osorio Atl Yosafat

**Profesor:**

Chadwick Carreto Arellano

**Fecha:**

16/03/2025

INSTITUTO POLITÉCNICO NACIONAL



## Índice de contenido

Antecedentes .....	4
Modelo Cliente/Servidor .....	4
Tipos de modelos Cliente/Servidor .....	5
Tipos de configuraciones en el Modelo Multicliente – Multiserver .....	6
Protocolos de comunicación .....	8
Socket .....	9
Planteamiento problema .....	10
Propuesta de solución .....	11
Materiales y métodos empleados.....	12
Materiales .....	12
Métodos .....	13
Desarrollo .....	15
Resultados.....	26
Conclusión .....	31
Referencias .....	32
Anexos .....	32
Código Fuente ServidorCine.java.....	32
Código Fuente MultiClienteCine.java.....	37
Código Fuente BalanceadorCarga.java .....	42

## Índice de figuras

<b>Figura 1. Diagrama de la arquitectura Cliente/Servidor.....</b>	<b>4</b>
<b>Figura 2. Modelo de N capas Cliente/Servidor .....</b>	<b>5</b>
<b>Figura 3. Diagrama de protocolos de comunicación .....</b>	<b>9</b>
<b>Figura 4. Diagrama de Sockets Cliente/Servidor .....</b>	<b>10</b>
<b>Figura 5. Bibliotecas empleadas.....</b>	<b>15</b>
<b>Figura 6. Class cine y metodo de boletos disponibles.....</b>	<b>15</b>
<b>Figura 7. Clase para manejar la reposición de boletos .....</b>	<b>16</b>
<b>Figura 8. Primera sección de la clase HiloCliente .....</b>	<b>17</b>
<b>Figura 9. Segunda sección de la clase HiloCliente.....</b>	<b>18</b>
<b>Figura 10. Clase principal del servidor .....</b>	<b>18</b>
<b>Figura 11. Clase BalanceadorCarga para gestionar las conexiones .....</b>	<b>19</b>
<b>Figura 12. Gestión de los servidores .....</b>	<b>20</b>
<b>Figura 13. Clase ManejadorConexion.....</b>	<b>20</b>
<b>Figura 14. Creación de hilos proxy para gestionar la transmisión de datos.....</b>	<b>21</b>
<b>Figura 15. Clase Proxy para gestionar el paso de datos .....</b>	<b>21</b>
<b>Figura 16. Hilos clienteAServidor y servidorACliente.....</b>	<b>22</b>
<b>Figura 17. Librerías utilizadas en el cliente .....</b>	<b>22</b>
<b>Figura 18. Class MultiClienteCine en el código del cliente.....</b>	<b>24</b>
<b>Figura 19. Método conectar () .....</b>	<b>24</b>
<b>Figura 20. Método verBoletos ().....</b>	<b>25</b>
<b>Figura 21. Método comprarBoletos().....</b>	<b>25</b>
<b>Figura 22. Método salir() .....</b>	<b>25</b>
<b>Figura 23. Método main.....</b>	<b>25</b>
<b>Figura 24. Ejecución del balanceador de carga.....</b>	<b>26</b>
<b>Figura 25. Ejecución del primer servidor en Windows desde VS code .....</b>	<b>26</b>
<b>Figura 26. Ejecución del segundo servidor alojado en una máquina virtual con Debian .....</b>	<b>26</b>
<b>Figura 27. Balanceador de carga redirigiendo la conexión del cliente .....</b>	<b>27</b>
<b>Figura 28. Servidor recibiendo la conexión del cliente exitosamente .....</b>	<b>27</b>
<b>Figura 29. Inicialización del cliente y su menú .....</b>	<b>28</b>
<b>Figura 30. Vista del cliente al seleccionar un servidor para conectarse.....</b>	<b>28</b>
<b>Figura 31. Cliente conectado .....</b>	<b>28</b>
<b>Figura 32. Vista del servidor al realizar una compra .....</b>	<b>29</b>
<b>Figura 33. Balanceador de carga con las distintas conexiones realizadas.....</b>	<b>29</b>
<b>Figura 34. Vista desde el servidor con la salida de un usuario.....</b>	<b>29</b>
<b>Figura 35. Conexión de múltiples clientes al servidor.....</b>	<b>30</b>
<b>Figura 36. Errores si es que los servidores no están iniciados.....</b>	<b>30</b>

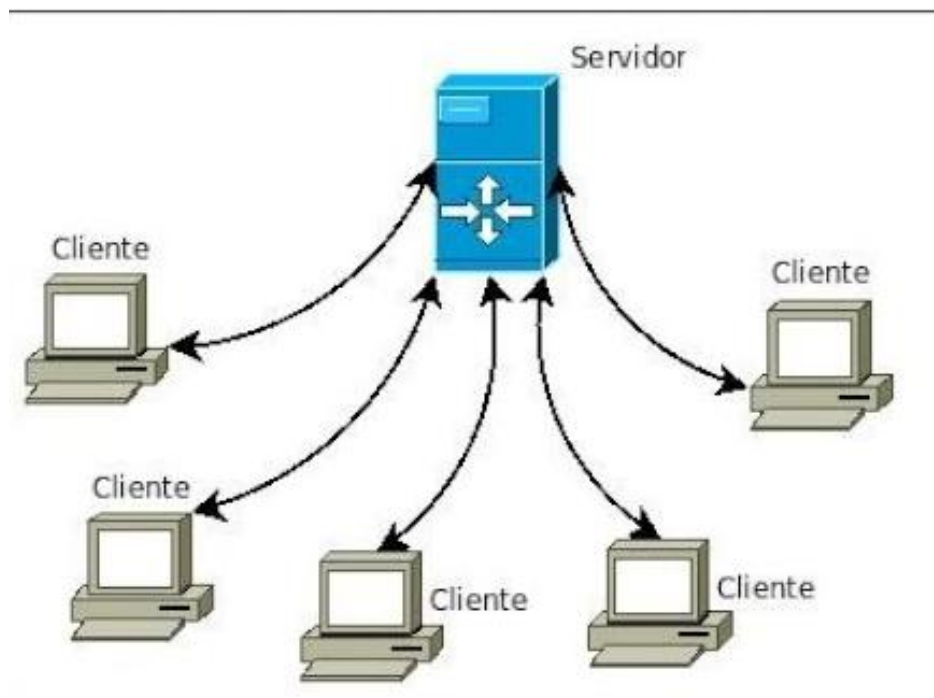
## Antecedentes

El modelo cliente-servidor es una arquitectura fundamental en la computación distribuida, y su diseño puede variar según la distribución de responsabilidades entre el cliente y el servidor, así como la cantidad de capas o niveles que lo componen.

## Modelo Cliente/Servidor

El modelo Cliente/Servidor es una arquitectura clave en el desarrollo de sistemas informáticos y redes, permitiendo la comunicación eficiente entre dispositivos mediante la diferenciación de roles. En este esquema, el servidor es el encargado de gestionar y ofrecer recursos, mientras que el cliente es quien realiza peticiones para acceder a dichos servicios. Esta estructura facilita la administración centralizada, optimiza el acceso a la información y permite distribuir eficientemente las cargas de trabajo en entornos computacionales.

Desde sus inicios, el concepto Cliente/Servidor ha sido esencial en la evolución de la informática. Durante las primeras etapas de la computación, los sistemas centralizados dominaban el panorama con el uso de mainframes, donde múltiples terminales se conectaban a un único sistema de procesamiento. Con el avance de la tecnología y la aparición de computadoras personales, la arquitectura Cliente/Servidor se convirtió en la solución ideal para distribuir tareas entre dispositivos, mejorando la eficiencia y flexibilidad en los sistemas informáticos. Actualmente, esta arquitectura es la base de diversas aplicaciones, desde plataformas web hasta servicios en la nube y videojuegos en línea.



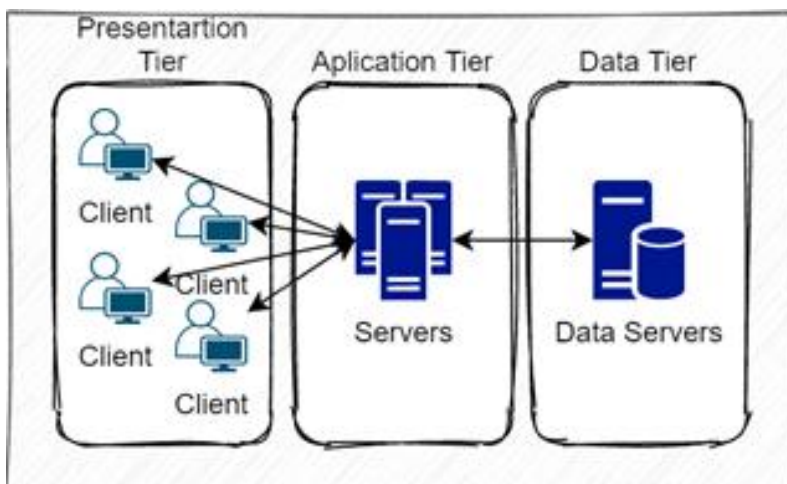
*Figura 1. Diagrama de la arquitectura Cliente/Servidor*

## Tipos de modelos Cliente/Servidor

El modelo de dos capas (two-tier) es la forma más básica de esta arquitectura. En este enfoque, solo existen dos componentes principales: el cliente y el servidor. El cliente se encarga de la interfaz de usuario y la presentación de datos, mientras que el servidor gestiona la lógica de negocio y el almacenamiento de la información. Este modelo es sencillo de implementar y mantener, lo que lo hace ideal para aplicaciones pequeñas o medianas. Sin embargo, su escalabilidad es limitada, ya que el servidor puede sobrecargarse si hay una gran cantidad de clientes conectados. Un ejemplo común de este modelo es una aplicación de escritorio que se conecta a una base de datos centralizada.

En contraste, el modelo de tres capas (three-tier) introduce una capa intermedia, conocida como middleware, entre el cliente y el servidor. Esta capa adicional se encarga de procesar las solicitudes del cliente y aplicar las reglas de negocio, separando así las responsabilidades en tres niveles: la capa de presentación (cliente), la capa de lógica de negocio (middleware) y la capa de datos (servidor). Este enfoque ofrece una mayor escalabilidad y flexibilidad, ya que permite una separación clara de funciones y facilita el mantenimiento y las actualizaciones. No obstante, su implementación es más compleja y costosa debido a la infraestructura adicional requerida. Un ejemplo típico es una aplicación web en la que el navegador (cliente) interactúa con un servidor de aplicaciones (middleware) que, a su vez, se conecta a una base de datos.

El modelo de N capas (N-tier) es una extensión del modelo de tres capas, donde las responsabilidades se dividen en múltiples niveles para mejorar la modularidad y escalabilidad. En este enfoque, las capas pueden incluir, además de las tres mencionadas, una capa de servicios (como APIs o servicios web) y otras capas especializadas según las necesidades de la aplicación. Este modelo es altamente escalable y modular, lo que lo hace ideal para aplicaciones empresariales complejas, como sistemas ERP (Enterprise Resource Planning). Sin embargo, su implementación es más costosa y requiere un diseño cuidadoso para evitar cuellos de botella.



*Figura 2. Modelo de N capas Cliente/Servidor*

## **Tipos de configuraciones en el Modelo Multicliente – Multiserver**

El modelo multicliente-multiserver se puede implementar de diversas formas dependiendo de los requerimientos del sistema, tales como el balance de carga, la redundancia, la alta disponibilidad, y la especialización de las tareas. Las siguientes configuraciones son comunes en este tipo de arquitecturas y cada una de ellas se adapta a diferentes necesidades de escalabilidad y rendimiento.

### **Balanceo de Carga**

El balanceo de carga es una de las configuraciones más utilizadas en sistemas multicliente-multiserver, ya que distribuye de manera equitativa las solicitudes de los clientes entre múltiples servidores disponibles. Esto asegura que ningún servidor esté sobrecargado, lo que a su vez mejora el tiempo de respuesta y el rendimiento general del sistema.

En esta configuración, un componente denominado balanceador de carga es responsable de gestionar la distribución de las peticiones de los clientes. Este balanceador puede ser hardware o software y utiliza diferentes algoritmos para repartir las solicitudes, como round-robin, por carga de trabajo o basándose en la capacidad actual de los servidores.

El balanceo de carga mejora la escalabilidad horizontal, ya que nuevos servidores pueden ser añadidos al sistema sin necesidad de reconfigurar el conjunto de clientes. Además, permite la tolerancia a fallos, ya que, si uno de los servidores no está disponible, el balanceador puede redirigir las solicitudes a otros servidores activos.

### **Redundancia y Alta Disponibilidad**

La redundancia es un principio fundamental en los sistemas distribuidos que busca asegurar que, en caso de que un servidor falle, el sistema continúe funcionando sin interrupciones. En un modelo multicliente-multiserver, se implementa la redundancia mediante la replicación de servidores, de modo que varios servidores proporcionen el mismo servicio o recurso.

Los servidores redundantes pueden estar configurados en modo activo-activo o activo-pasivo. En el modo activo-activo, todos los servidores están activos y comparten la carga, mientras que, en el modo activo-pasivo, solo uno de los servidores está activo en un momento dado, y el servidor pasivo entra en funcionamiento solo cuando el activo falla. Esta configuración garantiza que, incluso en situaciones de fallo, el sistema siga disponible para los clientes sin que se afecte la experiencia de usuario.

La alta disponibilidad (HA) es la capacidad del sistema para mantener el servicio disponible con el mínimo tiempo de inactividad. La combinación de balanceo de carga y redundancia es esencial para lograr una arquitectura de alta disponibilidad, que puede ser crítica en aplicaciones donde el tiempo de inactividad no es aceptable, como en sistemas financieros o plataformas de comercio electrónico.

## **Distribución de Tareas Especializadas**

En algunos sistemas, los servidores están especializados en diferentes tipos de tareas, lo que permite optimizar los recursos y mejorar la eficiencia. En este tipo de configuración, cada servidor se encarga de una parte específica del procesamiento, como bases de datos, servidores de aplicaciones o servidores de archivos.

Por ejemplo, un sistema multicliente-multiserver podría tener servidores dedicados solo a la gestión de bases de datos, mientras que otros manejan la lógica de la aplicación. Este tipo de especialización permite que cada servidor esté optimizado para un conjunto específico de tareas, lo que puede mejorar el rendimiento general del sistema. Además, permite una mayor escalabilidad, ya que, si se requiere más capacidad para una tarea en particular, se puede agregar un servidor adicional para esa función específica sin afectar otras partes del sistema.

Esta configuración también mejora la seguridad, ya que se pueden implementar controles de acceso específicos para cada tipo de servidor, protegiendo así los recursos más sensibles, como las bases de datos, de accesos no autorizados.

## **Escalabilidad Vertical y Horizontal**

El modelo multicliente-multiserver se puede escalar de dos formas principales: vertical y horizontal. La escalabilidad vertical implica la mejora de los recursos de un servidor existente (por ejemplo, agregar más CPU, memoria o almacenamiento) para manejar más clientes. Sin embargo, la escalabilidad vertical tiene limitaciones físicas y económicas.

Por otro lado, la escalabilidad horizontal permite agregar más servidores al sistema para distribuir la carga. Este tipo de escalabilidad es más flexible, ya que el sistema puede seguir creciendo, agregando más servidores sin necesidad de hacer cambios importantes en la infraestructura. El balanceo de carga juega un papel crucial en la escalabilidad horizontal, ya que garantiza que las solicitudes se distribuyan correctamente entre todos los servidores disponibles.

## **Configuración de Servidores Dedicados y Servidores Compartidos**

Otra posible configuración dentro del modelo multicliente-multiserver es la combinación de servidores dedicados y servidores compartidos. Los servidores dedicados son aquellos que están exclusivamente asignados a un cliente o grupo de clientes, ofreciendo una mayor personalización y control sobre el servicio. En cambio, los servidores compartidos gestionan las solicitudes de varios clientes, lo que puede ser más eficiente en términos de costos, pero podría presentar un rendimiento menos predecible.

La combinación de ambos tipos de servidores permite optimizar los costos, asignando servidores dedicados a los clientes más críticos o de mayor volumen, mientras que los clientes de menor carga pueden utilizar servidores compartidos. Esta estrategia también facilita la

segregación de recursos, lo que puede ser beneficioso para la seguridad y el cumplimiento de normativas, como en el caso de aplicaciones financieras o de salud.

## Protocolos de comunicación

Los sistemas cliente-servidor dependen en gran medida de los protocolos de comunicación para facilitar la transferencia de datos entre dispositivos. Estos protocolos establecen las reglas y los procedimientos para la transmisión de información, asegurando que la comunicación sea eficiente, confiable y segura. Algunos de los protocolos más importantes aplicados en estos sistemas:

**TCP/IP (Transmission Control Protocol/Internet Protocol):** Este protocolo es fundamental en Internet y en los sistemas cliente-servidor. TCP garantiza una conexión confiable al dividir los datos en paquetes y asegurarse de que se entreguen correctamente, mientras que IP se encarga de direccionar los paquetes a través de la red.

**HTTP (Hypertext Transfer Protocol):** Ampliamente utilizado en la web, HTTP define cómo se comunican los navegadores web y los servidores. Establece un formato para las solicitudes y respuestas, permitiendo la transferencia de recursos como páginas web, imágenes y otros archivos.

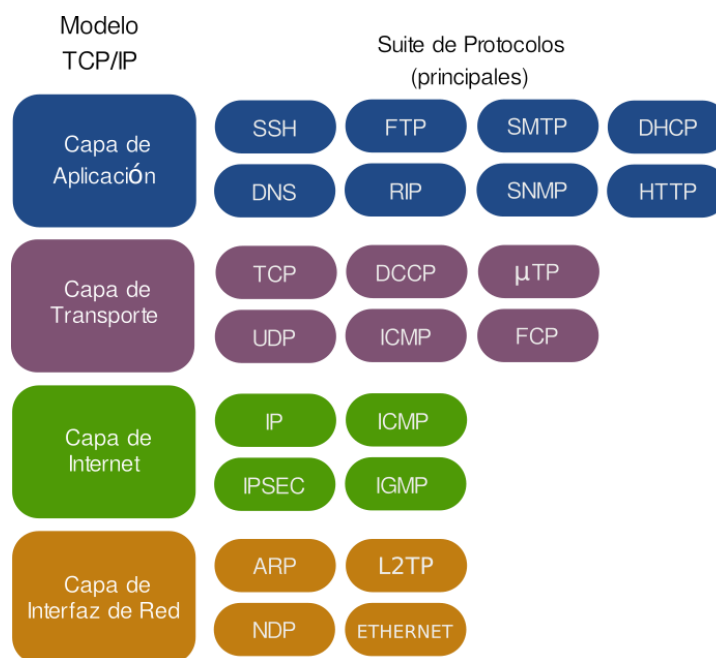
**FTP (File Transfer Protocol):** Especializado en la transferencia de archivos, FTP facilita la carga y descarga de archivos entre un cliente y un servidor. Proporciona un método seguro y controlado para gestionar archivos remotos.

**SMTP (Simple Mail Transfer Protocol):** Esencial para el envío de correos electrónicos, SMTP define cómo se transmiten los mensajes de correo entre servidores. Asegura que los correos se entreguen de manera confiable y eficiente.

**SSH (Secure Shell):** Utilizado para acceder de forma segura a servidores remotos, SSH cifra la comunicación entre el cliente y el servidor, protegiendo los datos sensibles durante la transmisión.

Estos protocolos son fundamentales para el funcionamiento efectivo de los sistemas cliente-servidor, ya que permiten la comunicación fluida y segura entre dispositivos. Desde la transferencia de archivos hasta el intercambio de correos electrónicos, estos protocolos juegan un papel crucial en numerosos aspectos de la interacción en línea. Su implementación adecuada garantiza una experiencia de usuario óptima y protege la integridad de los datos en todo momento. En general los protocolos de comunicación son el corazón de los sistemas cliente-servidor, proporcionando el marco necesario para la transmisión confiable de información en esta era, sin embargo, para el desarrollo de esta práctica únicamente nos enfocamos en el protocolo FTP y TCP.





*Figura 3. Diagrama de protocolos de comunicación*

## Socket

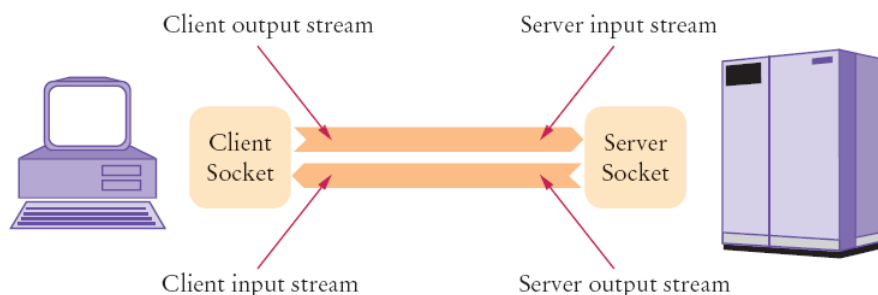
Un socket, en el contexto de las redes informáticas, es un concepto fundamental que sirve como interfaz de comunicación entre dos programas que se ejecutan en la red. Esencialmente, actúa como un punto final de una conexión entre dos nodos de una red, permitiendo la transferencia de datos de manera bidireccional.

Este término proviene del mundo físico, donde un socket es el lugar donde se conectan los dispositivos eléctricos para recibir energía o comunicarse. En el ámbito de las redes informáticas, un socket es una abstracción que representa un extremo de una conexión de red, permitiendo que los programas se comuniquen entre sí, ya sea en la misma máquina (comunicación interna) o en diferentes máquinas (comunicación externa a través de la red).

Los sockets son una parte integral de la arquitectura de red de los sistemas informáticos modernos y son utilizados por una amplia gama de aplicaciones, desde navegadores web hasta aplicaciones de mensajería instantánea y servidores de archivos. Facilitan la comunicación entre procesos de manera eficiente y flexible, lo que permite la creación de aplicaciones y sistemas distribuidos complejos.

En términos de programación, los sockets se implementan mediante API (Interfaz de Programación de Aplicaciones) proporcionadas por el sistema operativo. Estas API permiten a los desarrolladores crear y gestionar sockets, establecer conexiones, enviar y recibir datos, y cerrar conexiones cuando ya no son necesarias.

Los sockets pueden ser de diferentes tipos, dependiendo del protocolo de transporte que utilicen. Los más comunes son los sockets de flujo (stream sockets) y los sockets de datagrama (datagram sockets). Los sockets de flujo utilizan el protocolo TCP (Protocolo de Control de Transmisión) para establecer una conexión orientada a la conexión y garantizar la entrega ordenada de datos, mientras que los sockets de datagrama utilizan el protocolo UDP (Protocolo de Datagrama de Usuario) y ofrecen una comunicación sin conexión y no garantizan la entrega ordenada de datos.



*Figura 4. Diagrama de Sockets Cliente/Servidor*

En pocas palabras un socket es una interfaz de comunicación que permite a los programas intercambiar datos a través de una red, ya sea en la misma máquina o en máquinas remotas. Es una abstracción fundamental en el desarrollo de aplicaciones de red y proporciona la base para la implementación de diversos servicios y protocolos de comunicación. Sin los sockets, la comunicación entre programas en una red sería extremadamente difícil, si no imposible, lo que subraya su importancia en el ámbito de la informática moderna.

## Planteamiento problema

En el contexto de los sistemas distribuidos, la gestión eficiente de múltiples solicitudes de clientes hacia diversos servidores se convierte en un desafío crucial. Las aplicaciones modernas, como los sistemas de reserva de boletos en cines, requieren una infraestructura robusta y escalable para garantizar la disponibilidad y respuesta oportuna ante picos de demanda.

En esta práctica, se ha desarrollado un sistema Multicliente-Multiserver que simula un entorno de reserva de boletos de cine, utilizando un balanceador de carga para distribuir las solicitudes entre dos servidores. Cada servidor gestiona las solicitudes de las clientes relacionadas con la visualización y compra de boletos, así como la reposición automática de boletos después de cierto tiempo. El problema que se aborda en esta práctica está relacionado con la gestión de recursos compartidos en un entorno multicliente, en el que se debe garantizar la integridad de los datos (boletos) en presencia de múltiples solicitudes simultáneas, así como la correcta comunicación entre el cliente y el servidor.

Las principales problemáticas o desafíos que se enfrentan al realizar un sistema como este, el cual es Multicliente-Multiservidor son varios como la sobrecarga de servidores que ocurre cuando múltiples clientes intentan acceder simultáneamente al sistema. En estos casos, un único servidor podría experimentar una sobrecarga, lo que degrada significativamente el rendimiento y aumenta los tiempos de respuesta. Esto genera una experiencia negativa para los usuarios y puede afectar la disponibilidad del servicio.

La distribución ineficiente de recursos es otro desafío relevante. Sin un mecanismo de balanceo de carga, los recursos del sistema no se aprovechan de manera equitativa. Esto significa que algunos servidores pueden operar a su máxima capacidad mientras otros permanecen infrautilizados, reduciendo la eficiencia general del sistema.

La falta de escalabilidad también representa un problema crítico. Un sistema sin un balanceador de carga tiene una capacidad limitada para manejar un crecimiento en la cantidad de usuarios. Esto puede ocasionar tiempos de inactividad, caídas del servicio o una reducción en la calidad de la experiencia del usuario durante los picos de demanda.

Por último, la gestión de concurrencia es esencial en un entorno donde múltiples clientes intentan comprar boletos al mismo tiempo. La compra de boletos es una operación crítica que debe manejarse de manera sincronizada para evitar condiciones de carrera y garantizar la consistencia de los datos. Sin una adecuada gestión de concurrencia, podrían surgir inconsistencias que afecten tanto a los clientes como a la administración del cine.

## **Propuesta de solución**

La solución propuesta en esta práctica fue desarrollar un sistema distribuido compuesto por múltiples clientes, un balanceador de carga y múltiples servidores. El objetivo fue simular un sistema de venta de boletos para un cine, permitiendo a los usuarios consultar la disponibilidad de boletos, realizar compras y recibir confirmación en tiempo real.

El sistema debía manejar múltiples solicitudes de clientes de manera concurrente, distribuyendo la carga entre diferentes servidores de cine. Además, debía asegurar la consistencia en la cantidad de boletos disponibles, evitando condiciones de carrera y manteniendo la información actualizada para todos los usuarios.

La arquitectura implementada consta de tres componentes principales: el cliente, el balanceador de carga y el servidor de cine. El cliente proporciona una interfaz gráfica para conectarse a una sucursal, consultar boletos, comprarlos y cerrar sesión. El balanceador de carga distribuye las conexiones de los clientes a los servidores disponibles mediante un enfoque de selección, garantizando una distribución equitativa de las solicitudes. Por su parte, el servidor de cine gestiona las solicitudes, controla la disponibilidad de boletos y ejecuta un proceso de reposición automática cada 60 segundos.

Entre las principales funcionalidades implementadas, el cliente establece una conexión segura al balanceador, que redirige al servidor adecuado que se haya seleccionado o en dado caso que de estar lleno redirige al usuario a uno que aun tenga capacidad. Desde allí, los usuarios pueden verificar la disponibilidad de boletos, realizar compras y recibir una confirmación inmediata. Además, un mecanismo de reposición periódica garantiza la disponibilidad continua de boletos para nuevas funciones.

Para garantizar la sincronización y evitar condiciones de carrera, se utilizó el mecanismo de sincronización mediante bloques synchronized en el servidor. Esto aseguró la consistencia en el número de boletos disponibles durante las compras y reposiciones. Además, el uso de un ThreadPoolExecutor permitió manejar múltiples conexiones de clientes de forma eficiente y escalable.

En cuanto al balanceo de carga, se implementó una estrategia de selección para distribuir las conexiones de manera uniforme entre los servidores disponibles. Este enfoque evitó la sobrecarga de un único servidor y mejoró la disponibilidad del sistema.

## **Materiales y métodos empleados**

Para llevar a cabo la práctica y desarrollar la simulación de la venta de boletos con un modelo Cliente/Servidor se emplearon las siguientes herramientas y métodos:

### **Materiales**

#### ***1. Lenguaje de programación: Java***

Se eligió Java debido a su capacidad para el manejo de hilos y concurrencia, lo cual es esencial para la implementación de sistemas multicliente-multiserver. Además, Java proporciona bibliotecas como Socket, ServerSocket, BufferedReader y PrintWriter, que facilitan la comunicación a través de sockets. Por otro lado, para la creación de la interfaz gráfica del cliente, se utilizó la librería Swing de Java. Swing permite crear interfaces gráficas interactivas con componentes como botones, cuadros de texto, áreas de texto, etc., facilitando la interacción del usuario con el sistema.

#### ***2. Entorno de desarrollo: Visual Studio Code (VS Code)***

Fue seleccionado como entorno de desarrollo por su flexibilidad, amplia disponibilidad de extensiones y facilidad de uso. Además, cuenta con herramientas integradas para la depuración y la administración de proyectos en Java. Así como también es bastante cómodo trabajar en él.

### 3. *JDK (Java Development Kit):*

El JDK fue utilizado para compilar y ejecutar el código Java. Su compatibilidad con las bibliotecas y la implementación de concurrencia lo convierten en una herramienta esencial para esta práctica.

### 4. *Sistema Operativo: Windows y Debian.*

La práctica se realizó en un sistema operativo Windows debido a su amplia compatibilidad con Visual Studio Code y el JDK. Además de ello se emplearon máquinas virtuales para facilitar la ejecución y pruebas de la aplicación Multicliente-Multiservidor, dichas maquinas utilizaron distribuciones Linux como Debian.

## Métodos

### Manejo de concurrencia con hilos

Para garantizar que el servidor atienda múltiples clientes de manera simultánea, se utilizó el manejo de concurrencia mediante hilos. El servidor emplea un `ThreadPoolExecutor`, el cual administra un conjunto de hilos que permite procesar las solicitudes de varios clientes sin afectar el rendimiento del sistema.

**Hilos para Clientes:** Cada cliente es gestionado por un hilo independiente, implementado a través de la clase `HiloCliente`. Esta clase maneja la comunicación entre el cliente y el servidor, asegurando que cada usuario reciba respuestas personalizadas.

**Hilos del Servidor:** El servidor crea un pool de hilos utilizando `Executors.newFixedThreadPool()`, lo que optimiza la utilización de recursos del sistema al limitar el número máximo de conexiones concurrentes. Esto permite atender eficientemente a los clientes.

### Programación con Sockets TCP

La comunicación entre clientes y servidores se estableció mediante el protocolo TCP utilizando la API de sockets de Java. Este protocolo garantiza la entrega fiable de los datos.

**Servidor:** El servidor emplea un `ServerSocket` en el puerto 8889 para escuchar solicitudes de conexión. Una vez recibida una conexión, la clase `ServerSocket.accept()` crea un nuevo socket para gestionar la comunicación con el cliente.

**Cliente:** El cliente utiliza un `Socket` para conectarse al servidor a través del puerto 8888, administrado por el balanceador de carga. Para el intercambio de mensajes, se utilizaron `BufferedReader` y `PrintWriter`, facilitando una comunicación eficiente y confiable.

## Balanceo de Carga

Para distribuir equitativamente las solicitudes entre los servidores disponibles, se implementó un balanceador de carga. Este balanceador recibe las solicitudes de los clientes y redirige cada conexión a uno de los servidores disponibles de forma seleccionada, y posteriormente si está lleno lo redirige de manera aleatoria utilizando la clase Random.

**Manejador de Conexiones:** La clase ManejadorConexion se encarga de transferir datos entre el cliente y el servidor seleccionado, utilizando hilos independientes para gestionar la comunicación bidireccional.

## Sincronización de recursos compartidos

Dado que los boletos disponibles son un recurso compartido entre múltiples clientes, se emplearon métodos sincronizados (synchronized) en la clase Cine para garantizar la seguridad y consistencia de los datos.

**Compra de Boletos:** La función comprarBoleto() verifica la disponibilidad de boletos antes de realizar una compra, evitando condiciones de carrera.

**Consulta de Boletos Disponibles:** El método getBoletosDisponibles() retorna de manera segura el número actual de boletos sin interferir con otras operaciones.

## Implementación de un productor y consumidores

Siguiendo el modelo de Productor-Consumidor, se implementó un hilo adicional denominado ReposicionBoletos. Este hilo funciona como un productor, reponiendo boletos cada 60 segundos. Mientras tanto, los clientes actúan como consumidores al comprar boletos.

**Reposición de Boletos:** El método reponerBoletos() agrega nuevos boletos al inventario y utiliza notifyAll() para informar a los hilos en espera sobre la disponibilidad de boletos.

## Manejo de Entrada y Salida de Datos

El sistema implementa un manejo eficiente de entrada y salida de datos tanto en el cliente como en el servidor.

**Servidor:** Utiliza BufferedReader para recibir datos de los clientes y PrintWriter para enviar respuestas. El servidor proporciona un menú de opciones que incluye la consulta de boletos disponibles, la compra de boletos y la salida del sistema.

**Cliente:** La aplicación cliente cuenta con una interfaz gráfica desarrollada en Java Swing. Esta interfaz facilita la interacción del usuario mediante botones y cuadros de

texto, mejorando significativamente la experiencia de uso. Los mensajes del servidor se muestran en un área de texto (JTextArea) para una mejor visualización.

## Desarrollo

En esta práctica se implementó un sistema de venta de boletos para un cine utilizando el modelo cliente-servidor en Java, solamente que en este caso se utilizó el tipo de modelo Multicliente/Multiservidor. El sistema consta de tres componentes principales. El cliente es una aplicación con interfaz gráfica (GUI) que permite a los usuarios conectarse a una sucursal, visualizar la disponibilidad de boletos, realizar compras y salir del sistema. Los servidores son varias instancias independientes Sucursal Norte y Sucursal Sur, aunque que manejan la disponibilidad de boletos y procesan las solicitudes de los clientes de manera concurrente mediante el uso de hilos. Por último, el balanceador de carga es responsable de distribuir las conexiones de los clientes entre los servidores utilizando un algoritmo de selección aleatoria.

Primeramente, se realizó la clase Cine, la cual representa el recurso compartido entre los clientes, es decir el servidor. Su principal función es administrar la cantidad de boletos disponibles y gestionar las compras de manera segura y sincronizada.

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

*Figura 5. Bibliotecas empleadas*

El atributo “boletosDisponibles” almacena la cantidad de boletos en stock. La sincronización en el método “comprarBoleto” garantiza que solo un hilo pueda modificar la cantidad de boletos a la vez, evitando problemas de concurrencia. Adicionalmente, se proporciona un método para reponer boletos cada cierto tiempo.

```
class Cine {
    private int boletosDisponibles;

    public Cine(int boletosDisponibles) {
        this.boletosDisponibles = boletosDisponibles;
    }

    public synchronized String comprarBoleto(String cliente, int cantidad) {
        if (boletosDisponibles < cantidad) {
            return "No hay suficientes boletos. Espere a la próxima función.";
        }
        boletosDisponibles -= cantidad;
        System.out.println(cliente + " compró " + cantidad + " boletos. Boletos restantes: " + boletosDisponibles);
        return "Compra exitosa. Boletos restantes: " + boletosDisponibles;
    }

    public synchronized int getBoletosDisponibles() {
        return boletosDisponibles;
    }

    public synchronized void reponerBoletos(int cantidad) {
        boletosDisponibles += cantidad;
        notifyAll();
        System.out.println("Se han repuesto " + cantidad + " boletos. Boletos disponibles: " + boletosDisponibles);
    }
}
```

*Figura 6. Class cine y metodo de boletos disponibles*

Para evitar que los boletos se agoten permanentemente, se implementó un mecanismo de reposición automática mediante la clase “ReposicionBoletos”. Este hilo repone boletos cada 60 segundos para simular la llegada de nuevas funciones en el cine, asegurando que los clientes tengan nuevas oportunidades de compra. Aquí es donde se maneja el hilo para la reposición de boletos, que es importante para no quedarnos sin boletos que vender.

```
class ReposicionBoletos extends Thread { // Este metodo se encarga de reponer los boletos
    private final Cine cine; // Se crea un objeto de la clase Cine

    public ReposicionBoletos(Cine cine) { // Se crea un constructor que recibe un objeto de la clase Cine
        this.cine = cine;
    }

    @Override
    public void run() { // Se crea un metodo run
        while (true) {
            try { // Se utiliza un try-catch para manejar excepciones
                Thread.sleep(60000); // Se duerme el hilo por 60 segundos
                cine.reponerBoletos(cantidad:10); // Se llama al metodo reponerBoletos de la clase Cine
                System.out.println("Se han repuesto 10 boletos. Boletos disponibles: " + cine.getBoletosDisponibles());
            } catch (InterruptedException e) {
                System.err.println("Error al reponer boletos: " + e.getMessage());
            }
        }
    }
}
```

*Figura 7. Clase para manejar la reposición de boletos*

La clase HiloCliente es un componente clave en el servidor, ya que permite gestionar la comunicación con múltiples clientes de manera simultánea mediante hilos. Su propósito principal es recibir solicitudes de los clientes, procesarlas y devolver respuestas adecuadas, garantizando una interacción fluida con el sistema de compra de boletos de un cine.

Al iniciar, el hilo establece la comunicación con el cliente a través de un socket, permitiendo el intercambio de mensajes. Primero, recibe el nombre del cliente y le envía un mensaje de bienvenida. Luego, entra en un bucle donde espera recibir opciones del cliente, procesando cada solicitud según corresponda.



```

class HiloCliente extends Thread { // Se crea una clase HiloCliente que extiende de Thread
    private final Socket socket;
    private final Cine cine;

    public HiloCliente(Socket socket, Cine cine) {
        this.socket = socket;
        this.cine = cine;
    }

    @Override
    public void run() { // Se crea un metodo run
        try (
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(), autoFlush:true)
        ) {
            String nombreCliente = in.readLine(); // Se lee el nombre del cliente
            System.out.println(nombreCliente + " se ha conectado desde " + socket.getInetAddress());
            out.println("Bienvenido al Cine, " + nombreCliente + "!");

            boolean seguir = true; // Se crea una variable booleana seguir
            while (seguir) {
                String opcionStr = in.readLine(); // Se lee la opcion del cliente
                if (opcionStr == null) break;

                int opcion; // Se crea una variable opcion
                try {
                    opcion = Integer.parseInt(opcionStr);
                } catch (NumberFormatException e) {
                    out.println(x:"Opción no válida.");
                    continue;
                }
            }
        }
    }
}

```

*Figura 8. Primera sección de la clase HiloCliente*

El cliente puede elegir entre tres acciones principales: consultar la cantidad de boletos disponibles, comprar boletos o salir del sistema. Para la compra, se verifica que el usuario ingrese un número válido y se intenta completar la transacción. Si la cantidad de boletos solicitada es mayor a la disponible, se informa al cliente.

El código también maneja posibles errores, como entradas no válidas o problemas en la conexión, evitando que el servidor se vea afectado por fallas en la comunicación. Además, cuando el cliente decide salir, el hilo finaliza y libera los recursos utilizados.

Gracias a esta implementación, el servidor puede atender a varios clientes de forma independiente, asegurando que cada usuario tenga una experiencia adecuada al interactuar con el sistema de compra de boletos, es decir con la sucursal que el haya seleccionado.

```

        switch (opcion) {
            case 1 -> out.println("Boletos disponibles: " + cine.getBoletosDisponibles());
            case 2 -> {
                String cantidadStr = in.readLine();
                int cantidad;
                try {
                    cantidad = Integer.parseInt(cantidadStr);
                } catch (NumberFormatException e) {
                    out.println(x:"Cantidad no válida.");
                    continue;
                }
                out.println(cine.comprarBoleto(nombreCliente, cantidad));
            }
            case 3 -> {
                out.println(x:"Gracias por su compra. ¡Hasta luego!");
                seguir = false;
                System.out.println(nombreCliente + " ha salido.");
            }
            default -> out.println(x:"Opción no válida.");
        }
    }
} catch (IOException e) { // Se utiliza un catch para manejar excepciones
    System.err.println("Error en la conexión con el cliente: " + e.getMessage());
} finally {
    try { // Se utiliza un try-catch para manejar excepciones
        socket.close(); // Se cierra el socket
    } catch (IOException e) {
        System.err.println("Error al cerrar el socket: " + e.getMessage());
    }
}

```

*Figura 9. Segunda sección de la clase HiloCliente*

Por último, de la parte del servidor tenemos a la clase `ServidorCine`, la cual se encarga de iniciar el servidor y administrar las conexiones de los clientes. Utiliza un `ServerSocket` que escucha en el puerto 8889, así como también se especifica la IP del servidor para que posteriormente el balanceador de carga pueda redirigir al cliente y este pueda acceder a él, al recibir una conexión, asigna un hilo `HiloCliente` a cada nuevo cliente. Además, se utiliza un `ThreadPoolExecutor` para administrar la concurrencia de manera eficiente y limitar la cantidad de clientes atendidos simultáneamente.

```

public class ServidorCine { // Se crea una clase ServidorCine
    Run main | Debug main | Run | Debug
    public static void main(String[] args) { // Se crea un metodo main
        String ipServidor = "192.168.1.74"; //localhost"; // Se crea una variable ipServidor
        int puerto = 8889; // Se crea una variable puerto
        Cine cine = new Cine(boletosDisponibles:30); // Se crea un objeto de la clase Cine
        ThreadPoolExecutor pool = (ThreadPoolExecutor) Executors.newFixedThreadPool(nThreads:3);

        new ReposicionBoletos(cine).start(); // Se crea un objeto de la clase ReposicionBoletos y se inicia

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

            while (true) { // Se crea un ciclo while
                Socket socketCliente = serverSocket.accept();
                pool.execute(new HiloCliente(socketCliente, cine));
            }
        } catch (IOException e) { // Se utiliza un catch para manejar excepciones
            System.err.println("Error en el servidor: " + e.getMessage());
        }
    }
}

```

*Figura 10. Clase principal del servidor*

Por otro lado, tenemos a nuestro balanceador de carga, el cual lo realizamos para gestionar conexiones de clientes a varios servidores en un entorno de sistemas distribuidos. El balanceador se encarga de recibir las conexiones de los clientes, redirigirlas de manera eficiente a uno de los servidores disponibles, y gestionar la comunicación entre los clientes y los servidores. El objetivo es distribuir la carga de trabajo entre diferentes servidores para mejorar el rendimiento y la disponibilidad del servicio.

Primeramente, en este tenemos la clase `BalanceadorCarga`, la cual es la clase principal en este código y se encarga de recibir las conexiones entrantes de los clientes. Este se ejecuta como un servidor escuchando en el puerto 8888. El balanceador tiene una lista de servidores disponibles, representados por sus direcciones IP.

```
public class BalanceadorCarga { // Clase BalanceadorCarga
    private static final int PUERTO_BALANCEADOR = 8888; // Puerto del balanceador de carga
    private static final List<String> SERVIDORES = Arrays.asList(...a: "192.168.1.74", "192.168.1.82", "192.168.1.87");

    Run main | Debug main | Run | Debug
    public static void main(String[] args) { // Metodo main de la clase BalanceadorCarga
        try (ServerSocket serverSocket = new ServerSocket(PUERTO_BALANCEADOR)) { // Se crea un objeto de la clase ServerSocket
            System.out.println("Balanceador de carga iniciado en el puerto " + PUERTO_BALANCEADOR);

            while (true) { // Se crea un ciclo infinito
                try { // Se utiliza un try-catch para manejar excepciones
                    Socket clienteSocket = serverSocket.accept();
                    System.out.println("Conexión entrante desde: " + clienteSocket.getInetAddress());

                    String servidorElegido = seleccionarServidor();
                    System.out.println("Conexión redirigida al servidor: " + servidorElegido);

                    new Thread(new ManejadorConexion(clienteSocket, servidorElegido, puertoServidor:8889)).start();
                } catch (IOException e) {
                    System.err.println("Error al aceptar conexión del cliente: " + e.getMessage());
                }
            }
        } catch (IOException e) {
            System.err.println("Error al iniciar el balanceador de carga: " + e.getMessage());
        }
    }
}
```

*Figura 11. Clase `BalanceadorCarga` para gestionar las conexiones*

Cuando un cliente se conecta al balanceador, este selecciona uno de los servidores disponibles y si tiene mucha carga este lo redirigirá aleatoriamente a uno mediante el método `seleccionarServidor()`. Esto permite distribuir las conexiones de manera simple y balanceada entre los servidores.

Al aceptar una conexión, el balanceador crea un nuevo hilo (`Thread`) para gestionar la comunicación entre el cliente y el servidor seleccionado. Este hilo invoca la clase `ManejadorConexion`, la cual es responsable de manejar la comunicación real entre el cliente y el servidor.

Después tenemos al método `seleccionarServidor()` que tiene como objetivo seleccionar un servidor disponible para redirigir al cliente. Utiliza un generador de números aleatorios para escoger uno de los servidores en la lista `SERVIDORES`. Esto permite que las conexiones de

los clientes se distribuyan aleatoriamente entre los servidores, un enfoque simple de balanceo de carga.

```
private static String seleccionarServidor() {  
    // El balanceador redirige aleatoriamente a uno de los servidores disponibles,  
    // en caso de tener mucha carga en el solicitado.  
    Random rand = new Random();  
    return SERVIDORES.get(rand.nextInt(SERVIDORES.size())); // Se regresa un servidor aleatorio  
}
```

*Figura 12. Gestión de los servidores*

Ahora tenemos a la clase `ManejadorConexion` que es la que se encarga de gestionar la conexión entre el cliente y el servidor una vez que el balanceador ha seleccionado el servidor adecuado. Esta clase implementa la interfaz `Runnable`, lo que permite que cada conexión se maneje en un hilo separado, asegurando que múltiples clientes puedan ser atendidos simultáneamente.

```
class ManejadorConexion implements Runnable {  
    private final Socket clienteSocket;  
    private final String servidorDestino;  
    private final int puertoServidor;  
  
    public ManejadorConexion(Socket clienteSocket, String servidorDestino, int puertoServidor) {  
        this.clienteSocket = clienteSocket;  
        this.servidorDestino = servidorDestino;  
        this.puertoServidor = puertoServidor;  
    }  
  
    @Override
```

*Figura 13. Clase ManejadorConexion*

Dentro de su método `run()`, `ManejadorConexion` establece una nueva conexión a un servidor a través de un `Socket` (en este caso, el servidor seleccionado aleatoriamente). Después, crea dos hilos de proxy que gestionan la transmisión de datos entre el cliente y el servidor, uno de cliente a servidor (`clienteAServidor`) y otro de servidor a cliente (`servidorACliente`). Los hilos se ejecutan de manera simultánea, asegurando que los datos se envíen en ambas direcciones sin bloqueos.

```
@Override  
public void run() {  
    try (Socket servidorSocket = new Socket(servidorDestino, puertoServidor)) {  
        InputStream clienteInput = clienteSocket.getInputStream();  
        OutputStream clienteOutput = clienteSocket.getOutputStream();  
        InputStream servidorInput = servidorSocket.getInputStream();  
        OutputStream servidorOutput = servidorSocket.getOutputStream();  
  
        Thread clienteAServidor = new Thread(new Proxy(clienteInput, servidorOutput));  
        Thread servidorACliente = new Thread(new Proxy(servidorInput, clienteOutput));  
  
        clienteAServidor.start();  
        servidorACliente.start();  
    }  
}
```

```

        clienteAServidor.join();
        servidorACliente.join();
    } catch (IOException e) {
        System.err.println("Error al conectar con el servidor " + servidorDestino + ": " + e.getMessage());
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        System.err.println("Error en la ejecución del hilo de conexión: " + e.getMessage());
    } finally {
        try {
            clienteSocket.close();
        } catch (IOException e) {
            System.err.println("Error al cerrar el socket del cliente: " + e.getMessage());
        }
    }
}

```

*Figura 14. Creación de hilos proxy para gestionar la transmisión de datos.*

Cabe mencionar que clase Proxy es responsable de gestionar el paso de datos entre el cliente y el servidor. Cada hilo en esta clase lee datos desde un `InputStream` (ya sea del cliente o del servidor) y los escribe en un `OutputStream` correspondiente (del servidor o del cliente). Esto se hace de forma continua hasta que no haya más datos para leer (es decir, hasta que el flujo de datos llegue a su fin). Este enfoque asegura que los datos se transmitan de forma eficiente y en tiempo real entre los dos extremos.

La clase Proxy se utiliza en dos hilos en el `ManejadorConexion`: uno para transmitir los datos desde el cliente hacia el servidor, y otro para la dirección opuesta, desde el servidor hacia el cliente. Esto asegura que la comunicación sea bidireccional y eficiente.

```

class Proxy implements Runnable {
    private final InputStream in;
    private final OutputStream out;

    public Proxy(InputStream in, OutputStream out) {
        this.in = in;
        this.out = out;
    }

    @Override
    public void run() {
        try {
            byte[] buffer = new byte[1024];
            int bytesLeidos;
            while ((bytesLeidos = in.read(buffer)) != -1) {
                out.write(buffer, 0, bytesLeidos);
                out.flush();
            }
        } catch (IOException e) {
            System.err.println("Error de I/O en Proxy: " + e.getMessage());
        }
    }
}

```

*Figura 15. Clase Proxy para gestionar el paso de datos*

Por último, cabe mencionar que también se incluyeron mecanismos de manejo de excepciones. En cada clase, se captura cualquier IOException que pueda ocurrir durante la creación de sockets, la transmisión de datos o el cierre de conexiones. En caso de error, se imprime un mensaje informando sobre el problema. Además, al finalizar la comunicación, se cierra adecuadamente el socket del cliente para liberar los recursos.

También cabe aclarar que la utilización de hilos en el balanceador de carga y en el manejador de conexiones es crucial para permitir que el sistema pueda atender múltiples clientes de forma simultánea. Cada cliente es manejado en su propio hilo, lo que permite que el balanceador pueda atender a varios clientes al mismo tiempo sin bloquear el servicio. Los hilos clienteAServidor y servidorACliente permiten que el tráfico de datos fluya en ambas direcciones al mismo tiempo, mejorando el rendimiento y la experiencia del usuario.

```
Thread clienteAServidor = new Thread(new Proxy(clienteInput, servidorOutput));
Thread servidorACliente = new Thread(new Proxy(servidorInput, clienteOutput));

clienteAServidor.start();
servidorACliente.start();

clienteAServidor.join();
servidorACliente.join();

catch (IOException e) {
```

*Figura 16. Hilos clienteAServidor y servidorACliente*

Ahora vamos a ver el código del Multicliente el cual, por otro lado, es representado por la clase ClienteCine, establece una conexión al balanceador de carga para que este lo redirija a un servidor y ahí pueda realizar acciones como consultar la disponibilidad de boletos, comprarlos o salir de la sesión. Utiliza sockets para la comunicación entre el cliente y el servidor, y una interfaz visual permite al usuario interactuar de manera sencilla.

En primer lugar, el programa importa las bibliotecas necesarias para la interfaz gráfica (javax.swing.\* y java.awt.\*), la entrada y salida de datos (java.io.\*), y la gestión de conexiones de red (java.net.\*). Estas bibliotecas permiten manejar tanto la comunicación con el servidor como la presentación visual del cliente.

```
import java.awt.*;
import java.io.*;
import java.net.*;
import javax.swing.*;
```

*Figura 17. Librerías utilizadas en el cliente*

El constructor principal de la clase MultiClienteCine establece la conexión con el Balanceador de carga, cuyo IP y puerto están definidos como constantes (SERVIDOR = 192.168.1.74 y PUERTO = 8888). Si la conexión falla, el programa muestra un mensaje de

error y se cierra. Posteriormente, se configura la ventana principal de la interfaz gráfica (JFrame), que contiene varios componentes, el diseño de la ventana se organiza con un BorderLayout, dividiendo la ventana en diferentes secciones: en la parte superior se coloca un panel (topPanel) con un campo de texto (JTextField) donde el usuario ingresa su nombre. También se incluye un botón llamado "Conectar con Sucursal", que cuando es presionado, ejecuta el método conectar(). así como un área de texto para mostrar los mensajes del servidor y botones para interactuar con el sistema. Los botones permiten al usuario realizar tres acciones principales: ver la cantidad de boletos disponibles, comprar boletos y salir del sistema.

```
public class MultiClienteCine { // Clase MultiClienteCine
    private final int PUERTO = 8888; // Puerto del balanceador de carga
    private Socket socket; // Socket del cliente
    private BufferedReader entrada;
    private PrintWriter salida; // PrintWriter para enviar mensajes al servidor
    private final JFrame frame; // JFrame para la interfaz grafica
    private final JTextField nombreField;
    private final JTextArea displayArea;
    private final JButton verBoletosBtn, comprarBtn, salirBtn;

    public MultiClienteCine() {
        frame = new JFrame(title:"Cliente Cine"); // Se crea un objeto de la clase JFrame
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Se establece la operación predeterminada al cerrar la ventana
        frame.setSize(width:400, height:300); // Se establece el tamaño de la ventana
        frame.setLayout(new BorderLayout()); // Se establece el layout de la ventana

        JPanel topPanel = new JPanel(); // Se crea un objeto de la clase JPanel
        topPanel.setLayout(new BorderLayout());
        nombreField = new JTextField();
        topPanel.add(new JLabel(text:"Ingresa su nombre: "), BorderLayout.WEST);
        topPanel.add(nombreField, BorderLayout.CENTER);

        JButton conectarBtn = new JButton(text:"Conectar con Sucursal");
        conectarBtn.addActionListener(e -> conectar()); // The value of the lambda expression is the method to be executed
        topPanel.add(conectarBtn, BorderLayout.EAST);

        frame.add(topPanel, BorderLayout.NORTH);

        displayArea = new JTextArea();
        displayArea.setEditable(b:false);
        frame.add(new JScrollPane(displayArea), BorderLayout.CENTER);

        JPanel sidePanel = new JPanel();
        sidePanel.setLayout(new GridLayout(rows:3, cols:1));

        verBoletosBtn = new JButton(text:"Ver Boletos");
        verBoletosBtn.addActionListener(e -> verBoletos()); // The value of the lambda expression is the method to be executed
        sidePanel.add(verBoletosBtn);

        comprarBtn = new JButton(text:"Comprar");
        comprarBtn.addActionListener(e -> comprarBoletos()); // The value of the lambda expression is the method to be executed
        sidePanel.add(comprarBtn);

        salirBtn = new JButton(text:"Salir");
        salirBtn.addActionListener(e -> salir()); // The value of the lambda expression is the method to be executed
        sidePanel.add(salirBtn);
    }
}
```

```

        frame.add(sidePanel, BorderLayout.EAST);
        frame.setVisible(b:true);
    }

```

*Figura 18. Class MultiClienteCine en el código del cliente*

Por otro lado el método conectar() se ejecuta cuando el usuario presiona el botón de conexión. En este método, primero se verifica que el campo de nombre no esté vacío. Si está vacío, se muestra un cuadro de diálogo de error solicitando al usuario que ingrese su nombre. Posteriormente, el usuario selecciona una sucursal de cine a través de un cuadro de diálogo con las opciones "Sucursal Norte" o "Sucursal Sur". Dependiendo de la selección, se asigna la dirección IP correspondiente del servidor al que el cliente se conectará. Si todo es correcto, se establece una conexión con el servidor usando un Socket, y se crean los flujos de entrada y salida necesarios para comunicarse con el servidor.

```

private void conectar() { // Metodo conectar
    String nombre = nombreField.getText().trim();
    if (nombre.isEmpty()) { // Se verifica si el nombre esta vacio
        JOptionPane.showMessageDialog(frame, message:"Por favor, ingrese su nombre.", title:"Error", JOptionPane.ERROR_MESSAGE);
        return;
    }

    // Seleccionar servidor (sucursal)
    String[] servidores = {"Sucursal Norte (192.168.1.74)", "Sucursal Sur (192.168.1.82)"};
    String sucursalSeleccionada = (String) JOptionPane.showInputDialog( // Se crea un cuadro de dialogo
        frame,
        message:"Seleccione una sucursal", //
        title:"Conexión al Cine",
        JOptionPane.QUESTION_MESSAGE,
        icon:null,
        servidores,
        servidores[0]
    );

    if (sucursalSeleccionada == null) { // Se verifica si no se selecciono una sucursal
        JOptionPane.showMessageDialog(frame, message:"Debe seleccionar una sucursal", title:"Error", JOptionPane.ERROR_MESSAGE);
        return;
    }

    String ipServidor = sucursalSeleccionada.contains(s:"1") ? "192.168.1.74" : "192.168.1.82"; // Se selecciona la ip del servidor

    try {
        socket = new Socket(ipServidor, PUERTO); // Se crea un objeto de la clase Socket
        entrada = new BufferedReader(new InputStreamReader(socket.getInputStream())); // Se crea un objeto de la clase BufferedRead
        salida = new PrintWriter(socket.getOutputStream(), autoFlush:true);
        salida.println(nombre); // Enviar nombre al servidor
        displayArea.append(entrada.readLine() + "\n"); // Mostrar mensaje del servidor
    } catch (IOException e) { // Se utiliza un catch para manejar excepciones
        JOptionPane.showMessageDialog(frame, message:"No se pudo conectar al servidor.", title:"Error", JOptionPane.ERROR_MESSAGE);
        System.exit(status:1);
    }
}

```

*Figura 19. Método conectar ()*

El nombre del usuario se envía al servidor, y el cliente recibe un mensaje que se muestra en el área de texto. El método verBoletos() envía una solicitud al servidor para obtener la cantidad de boletos disponibles, y luego muestra la respuesta en el área de texto. El método comprarBoletos() solicita al usuario la cantidad de boletos que desea comprar y envía esa información al servidor, mostrando la respuesta del servidor (ya sea confirmando la compra



o indicando un error). Finalmente, el método salir() cierra la conexión con el servidor y la interfaz gráfica, despidiéndose del usuario.

```
private void verBoletos() { // Metodo verBoletos
    salida.println(x:"1"); // Se envia un 1 al servidor
    try {
        displayArea.append("Servidor: " + entrada.readLine() + "\n");
    } catch (IOException e) { // Se utiliza un catch para manejar excepciones
        displayArea.append(str:"Error al recibir información.\n");
    }
}
```

*Figura 20. Método verBoletos ()*

```
private void comprarBoletos() { // Metodo comprarBoletos
    String cantidadStr = JOptionPane.showInputDialog(frame, message:"Ingrese la cantidad de boletos:");
    if (cantidadStr == null) return; // Se verifica si la cantidad es nula
    salida.println(x:"2"); // Se envia un 2 al servidor
    salida.println(cantidadStr);
    try {
        displayArea.append("Servidor: " + entrada.readLine() + "\n");
    } catch (IOException e) {
        displayArea.append(str:"Error al recibir respuesta.\n");
    }
}
```

*Figura 21. Método comprarBoletos()*

```
private void salir() { // Metodo salir
    salida.println(x:"3"); // Se envia un 3 al servidor
    try {
        displayArea.append("Servidor: " + entrada.readLine() + "\n");
        socket.close(); // Se cierra el socket
    } catch (IOException e) {
        displayArea.append(str:"Error al cerrar la conexión.\n");
    }
    frame.dispose();
}
```

*Figura 22. Método salir()*

Por último, el método main() garantiza que la interfaz gráfica se ejecute en el hilo correcto utilizando `SwingUtilities.invokeLater(MultiClienteCine::new)`, lo que asegura que la aplicación funcione correctamente en plataformas gráficas. Este enfoque es necesario para mantener la UI interactiva y responsiva.

```
Run | Debug | Run main | Debug main
public static void main(String[] args) { // Metodo main
    SwingUtilities.invokeLater(MultiClienteCine::new);
}
```

*Figura 23. Método main*

## Resultados

La implementación del sistema Multicliente-Multiservidor para la compra de boletos de cine permitió evaluar la comunicación eficiente entre múltiples clientes y múltiples servidores asegurando que el tráfico de clientes se distribuyera de manera adecuada entre los servidores disponibles.

La arquitectura del sistema se basa en un balanceador de carga que recibe las conexiones de los clientes y las redirige a uno de los servidores disponibles en este caso, los servidores definidos en la lista de servidores. El balanceador de carga fue capaz de distribuir las conexiones de manera aleatoria entre los servidores, lo que permitió comprobar que ambos servidores estaban recibiendo tráfico de manera equilibrada. Esta característica es crucial para mantener un rendimiento adecuado cuando hay múltiples clientes interactuando con el sistema de cine. En las siguientes imágenes se muestra el balanceador de carga en ejecución, así como los servidores, en este caso empleamos dos, pero podemos utilizar muchos más.

```
PS C:\Users\Atl1God\Desktop\ESCOM ATL\OCTAVO SEM  
CodeDetailsInExceptionMessages' '-cp' 'C:\Users\  
bcad47\bin' 'BalanceadorCarga'  
Balanceador de carga iniciado en el puerto 8888
```

*Figura 24. Ejecución del balanceador de carga*

```
PS C:\Users\Atl1God\Desktop\ESCOM ATL\OCTAVO SEM  
CodeDetailsInExceptionMessages' '-cp' 'C:\Users\  
bcad47\bin' 'ServidorCine'  
Servidor de cine iniciado en 192.168.1.74:8889
```

*Figura 25. Ejecución del primer servidor en Windows desde VS code*

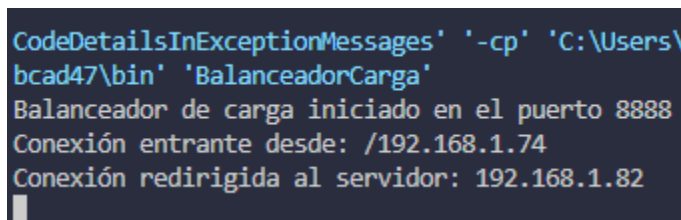
```
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to  
permitted by applicable law.  
Last login: Mon Mar 17 20:42:43 2025 from 192.168.1.74  
administrador@serDebian:~$ su -  
Contraseña:  
root@serDebian:~# cd Escritorio/  
root@serDebian:~/Escritorio# javac ServidorCine.java  
root@serDebian:~/Escritorio# java ServidorCine  
Servidor de cine iniciado en 192.168.1.82:8889
```

*Figura 26. Ejecución del segundo servidor alojado en una máquina virtual con Debian*

Para las pruebas, se utilizaron máquinas virtuales (MV), en las cuales se corrieron tanto los servidores como el cliente en distintas instancias de Windows y Debian. Esta configuración permitió simular un entorno distribuido realista, donde las máquinas virtuales en Debian actuaron como uno de los servidores es decir como una sucursal y en Windows se tuvo otra. Durante las pruebas, se logró verificar que los cliente pudieron conectarse exitosamente a los

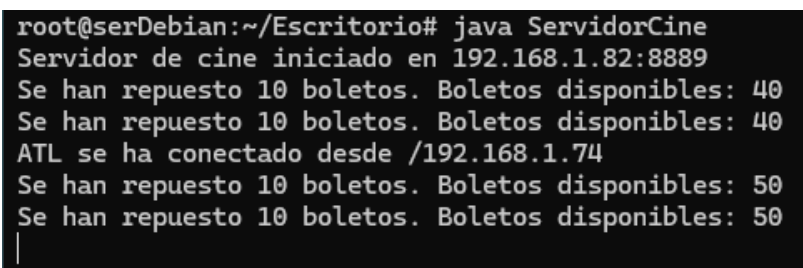
servidores mediante el balanceador de carga, enviar solicitudes de compra de boletos y recibir respuestas adecuadas del servidor al que fue redirigido.

En cuanto a la interacción cliente-servidor, se evidenció que el sistema era capaz de manejar las solicitudes concurrentes de múltiples clientes, como la visualización de boletos disponibles y la compra de boletos, sin presentar errores. Esto se logró gracias a la implementación de hilos en los servidores, que permiten procesar las solicitudes de cada cliente de manera independiente y en paralelo, garantizando que las operaciones no se bloqueen entre sí.



```
CodeDetailsInExceptionMessages' '-cp' 'C:\Users\
bcad47\bin' 'BalanceadorCarga'
Balanceador de carga iniciado en el puerto 8888
Conexión entrante desde: /192.168.1.74
Conexión redirigida al servidor: 192.168.1.82
```

*Figura 27. Balanceador de carga redirigiendo la conexión del cliente*



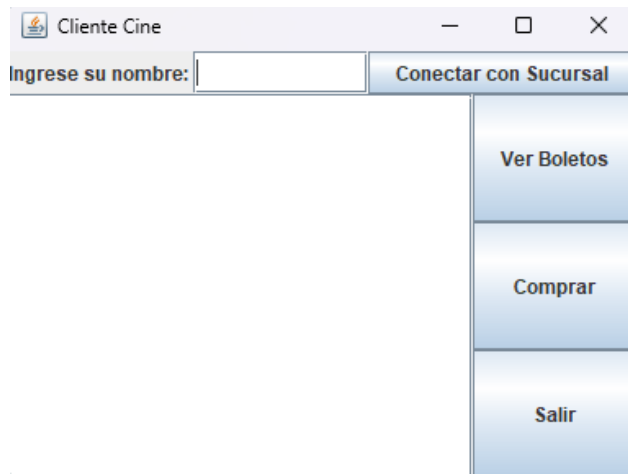
```
root@serDebian:~/Escritorio# java ServidorCine
Servidor de cine iniciado en 192.168.1.82:8889
Se han repuesto 10 boletos. Boletos disponibles: 40
Se han repuesto 10 boletos. Boletos disponibles: 40
ATL se ha conectado desde /192.168.1.74
Se han repuesto 10 boletos. Boletos disponibles: 50
Se han repuesto 10 boletos. Boletos disponibles: 50
```

*Figura 28. Servidor recibiendo la conexión del cliente exitosamente*

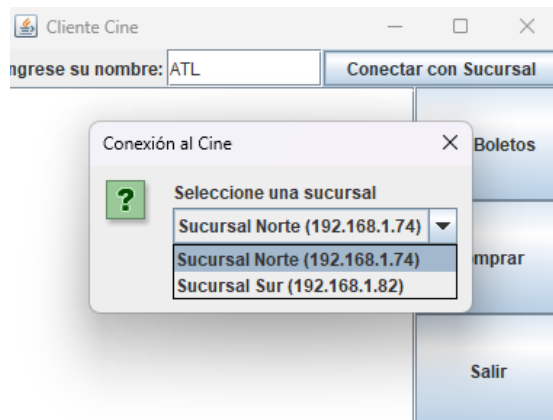
El balanceador de carga también mostró su efectividad al gestionar las conexiones de manera eficiente, redirigiendo las peticiones de los clientes a servidores distintos sin que se presentaran fallos de conexión o de distribución. Además, el servidor de cine estuvo configurado para reponer boletos de manera automática cada cierto intervalo de tiempo, lo que simula la reposición continua de los recursos y permite que los clientes realicen compras en todo momento, incluso en situaciones de alta demanda.

Es por eso por lo que podemos decir que la implementación cumplió con los principios de los sistemas distribuidos, logrando una comunicación efectiva, sincronización de datos y escalabilidad. La correcta gestión de la concurrencia y la automatización en la reposición de boletos demostraron ser estrategias clave para mantener un sistema funcional y eficiente.

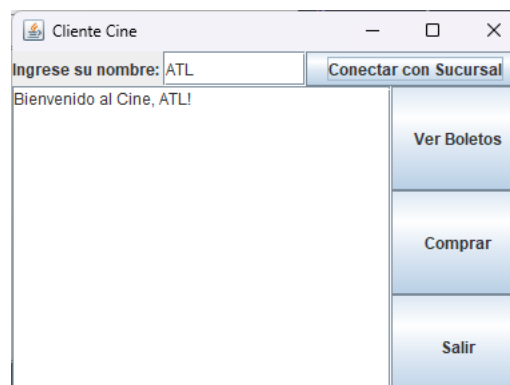
A continuación, se muestran diversas capturas de pantalla en donde observamos los procesos de los servidores y el balanceador de carga, así como las interfaces de los clientes:



*Figura 29. Inicialización del cliente y su menú*



*Figura 30. Vista del cliente al seleccionar un servidor para conectarse*



*Figura 31. Cliente conectado*

```

ATL se ha conectado desde /192.168.1.74
KELLY se ha conectado desde /192.168.1.74
Se han repuesto 10 boletos. Boletos disponibles: 120
Se han repuesto 10 boletos. Boletos disponibles: 120
YOSAFAT se ha conectado desde /192.168.1.74
YOSAFAT ha salido.
Se han repuesto 10 boletos. Boletos disponibles: 130
Se han repuesto 10 boletos. Boletos disponibles: 130
Se han repuesto 10 boletos. Boletos disponibles: 140
Se han repuesto 10 boletos. Boletos disponibles: 140
Se han repuesto 10 boletos. Boletos disponibles: 150
Se han repuesto 10 boletos. Boletos disponibles: 150
Se han repuesto 10 boletos. Boletos disponibles: 160
Se han repuesto 10 boletos. Boletos disponibles: 160
CARDOSO se ha conectado desde /192.168.1.74
Se han repuesto 10 boletos. Boletos disponibles: 170
Se han repuesto 10 boletos. Boletos disponibles: 170
Se han repuesto 10 boletos. Boletos disponibles: 180
Se han repuesto 10 boletos. Boletos disponibles: 180
Se han repuesto 10 boletos. Boletos disponibles: 190
Se han repuesto 10 boletos. Boletos disponibles: 190
KELLY compró 100 boletos. Boletos restantes: 90
Se han repuesto 10 boletos. Boletos disponibles: 100
Se han repuesto 10 boletos. Boletos disponibles: 100
ATL compró 5 boletos. Boletos restantes: 95
CARDOSO compró 54 boletos. Boletos restantes: 41
YOSAFAT compró 8 boletos. Boletos restantes: 33
Se han repuesto 10 boletos. Boletos disponibles: 43
Se han repuesto 10 boletos. Boletos disponibles: 43

```

*Figura 32. Vista del servidor al realizar una compra*

```

Se han repuesto 10 boletos. Boletos disponibles: 73
Se han repuesto 10 boletos. Boletos disponibles: 73
KELLY ha salido.
CARDOSO ha salido.
ATL ha salido.
YOSAFAT ha salido.
Se han repuesto 10 boletos. Boletos disponibles: 83
Se han repuesto 10 boletos. Boletos disponibles: 83

```

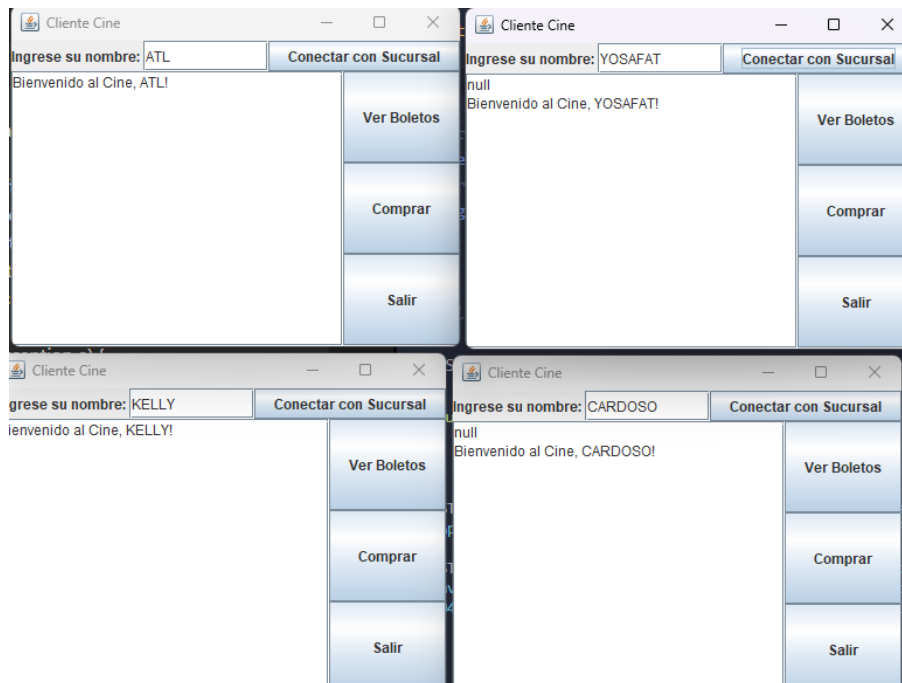
*Figura 34. Vista desde el servidor con la salida de un usuario*

```

Balanceador de carga iniciado en el puerto 8888
Conexión entrante desde: /192.168.1.74
Conexión redirigida al servidor: 192.168.1.74
Error al conectar con el servidor 192.168.1.74: Connection refused: connect
Conexión entrante desde: /192.168.1.74
Conexión redirigida al servidor: 192.168.1.74
Error al conectar con el servidor 192.168.1.74: Connection refused: connect
Conexión entrante desde: /192.168.1.74
Conexión redirigida al servidor: 192.168.1.74
Error al conectar con el servidor 192.168.1.74: Connection refused: connect
Conexión entrante desde: /192.168.1.74
Conexión redirigida al servidor: 192.168.1.82
Conexión entrante desde: /192.168.1.74
Conexión redirigida al servidor: 192.168.1.82
Conexión entrante desde: /192.168.1.74
Conexión redirigida al servidor: 192.168.1.82
Conexión entrante desde: /192.168.1.74
Conexión redirigida al servidor: 192.168.1.74
Error al conectar con el servidor 192.168.1.74: Connection refused: connect
Conexión entrante desde: /192.168.1.74
Conexión redirigida al servidor: 192.168.1.82
Conexión entrante desde: /192.168.1.74
Conexión redirigida al servidor: 192.168.1.74
Error al conectar con el servidor 192.168.1.74: Connection refused: connect
Conexión entrante desde: /192.168.1.74
Conexión redirigida al servidor: 192.168.1.82

```

*Figura 33. Balanceador de carga con las distintas conexiones realizadas*



*Figura 35. Conexión de múltiples clientes al servidor*

```

Balanceador de carga iniciado en el puerto 8888
Conexión entrante desde: /192.168.1.74
Conexión redirigida al servidor: 192.168.1.74
Error al conectar con el servidor 192.168.1.74: Connection refused: connect
Conexión entrante desde: /192.168.1.74
Conexión redirigida al servidor: 192.168.1.74
Error al conectar con el servidor 192.168.1.74: Connection refused: connect
Conexión entrante desde: /192.168.1.74
Conexión redirigida al servidor: 192.168.1.74
Error al conectar con el servidor 192.168.1.74: Connection refused: connect

```

*Figura 36. Errores si es que los servidores no están iniciados*

## Conclusión

La práctica sobre la implementación de un sistema multicliente-multiserver ha sido una experiencia muy enriquecedora, que me permitió comprender mejor los conceptos de comunicación entre múltiples clientes y servidores en un entorno distribuido. A través de la creación de un balanceador de carga y varios servidores, pude experimentar de primera mano cómo se gestionan las conexiones concurrentes, algo esencial en los sistemas distribuidos modernos.

Una de las partes que me resultó más complicada fue la implementación del balanceador de carga. Al principio, no entendía completamente cómo hacer que las conexiones de los clientes se distribuyeran de manera eficiente entre los servidores disponibles. Teniendo en cuenta que debía manejar las conexiones entrantes y redirigirlas correctamente a los servidores con disponibilidad, fue necesario entender en detalle el uso de sockets y el manejo de múltiples hilos. A pesar de la dificultad inicial, el proceso de depuración me ayudó a entender cómo los hilos interactúan y cómo se puede optimizar la comunicación entre los diferentes componentes del sistema.

Otra área que me causó ciertos retos fue el manejo de la sincronización en el servidor, específicamente en la clase Cine. La sincronización de los boletos disponibles para que múltiples clientes pudieran acceder y modificar este recurso compartido sin provocar inconsistencias me hizo reflexionar sobre la importancia de manejar correctamente los bloqueos y las condiciones de carrera. Afortunadamente, pude usar la palabra clave `synchronized` para evitar problemas de concurrencia y asegurar que el sistema funcionara de manera coherente incluso con múltiples clientes intentando comprar boletos al mismo tiempo.

Lo que me pareció más interesante fue la implementación de la reposición automática de boletos en el servidor, ya que me permitió ver cómo se pueden combinar hilos de manera efectiva para manejar tareas periódicas sin afectar el funcionamiento de las conexiones de los clientes. Además, la estructura del balanceador de carga, que utiliza hilos para crear una especie de "proxy" entre el cliente y los servidores, me hizo darme cuenta de cómo las arquitecturas distribuidas pueden ser escaladas y optimizadas para soportar una gran cantidad de usuarios.

Por último, me gustaría mencionar que esta práctica me permitió adentrarme en los principios fundamentales de los sistemas distribuidos, especialmente en lo que respecta a la gestión de múltiples clientes y servidores, la sincronización de recursos compartidos y la implementación de balanceadores de carga. Aunque hubo dificultades, sobre todo en la parte de la gestión de conexiones y la sincronización, la experiencia fue sumamente educativa y me ayudó a consolidar mis conocimientos en este campo.

## Referencias

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: 9 Marzo 2025].

A. Silberschatz, P. Galvin y G. Gagne, *Operating System Concepts*, 10a ed., Wiley, 2018.

Code & Coke, "Sockets," 2024. [En línea]. Available: <https://psp.codeandcoke.com/apuntes:sockets>. [Último acceso: 9 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: 8 Marzo 2025].

Daemon4, "Arquitectura Cliente-Servidor," 2024. [En línea]. Available: <https://www.daemon4.com/empresa/noticias/arquitectura-cliente-servidor/>. [Último acceso: 6 Marzo 2025].

## Anexos

### Código Fuente ServidorCine.java

```
/*
 * Nombre: Cardoso Osorio Atl Yosafat
 * Grupo: 7CM1
 * Materia: Sistemas Distribuidos
 * Practica 4: Modelo Multicliente/MultiServidor
 * Fecha: 17 de Marzo del 2025
 */
import java.io.*;
import java.net.*;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;

class Cine {
    private int boletosDisponibles;

    public Cine(int boletosDisponibles) {
        this.boletosDisponibles = boletosDisponibles;
    }
}
```



```

        public synchronized String comprarBoleto(String cliente, int
cantidad) {
            if (boletosDisponibles < cantidad) {
                return "No hay suficientes boletos. Espere a la
próxima función.";
            }
            boletosDisponibles -= cantidad;
            System.out.println(cliente + " compró " + cantidad + "
boletos. Boletos restantes: " + boletosDisponibles);
            return "Compra exitosa. Boletos restantes: " +
boletosDisponibles;
        }

        public synchronized int getBoletosDisponibles() {
            return boletosDisponibles;
        }

        public synchronized void reponerBoletos(int cantidad) {
            boletosDisponibles += cantidad;
            notifyAll();
            System.out.println("Se han repuesto " + cantidad + "
boletos. Boletos disponibles: " + boletosDisponibles);
        }
    }

class ReposicionBoletos extends Thread { // Este metodo se encarga
de reponer los boletos
    private final Cine cine; // Se crea un objeto de la clase Cine

    public ReposicionBoletos(Cine cine) { // Se crea un constructor
que recibe un objeto de la clase Cine
        this.cine = cine;
    }

    @Override
    public void run() { // Se crea un metodo run
        while (true) {

```

```

        try { // Se utiliza un try-catch para manejar
excepciones
            Thread.sleep(60000); // Se duerme el hilo por 20
segundos
            cine.reponerBoletos(10); // Se llama al metodo
reponerBoletos de la clase Cine
            System.out.println("Se han repuesto 10 boletos.
Boletos disponibles: " + cine.getBoletosDisponibles());
        } catch (InterruptedException e) {
            System.err.println("Error al reponer boletos: " +
e.getMessage());
        }
    }
}

class HiloCliente extends Thread { // Se crea una clase
HiloCliente que extiende de Thread
    private final Socket socket;
    private final Cine cine;

    public HiloCliente(Socket socket, Cine cine) {
        this.socket = socket;
        this.cine = cine;
    }

    @Override
    public void run() { // Se crea un metodo run
        try (
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter out = new
PrintWriter(socket.getOutputStream(), true)
        ) {
            String nombreCliente = in.readLine(); // Se lee el
nombre del cliente
            System.out.println(nombreCliente + " se ha conectado
desde " + socket.getInetAddress());

```

```

        out.println("Bienvenido al Cine, " + nombreCliente +
"!");

        boolean seguir = true; // Se crea una variable
booleana seguir
        while (seguir) {
            String opcionStr = in.readLine(); // Se lee la
opcion del cliente
            if (opcionStr == null) break;

            int opcion; // Se crea una variable opcion
            try {
                opcion = Integer.parseInt(opcionStr);
            } catch (NumberFormatException e) {
                out.println("Opción no válida.");
                continue;
            }

            switch (opcion) {
                case 1 -> out.println("Boletos disponibles: "
+ cine.getBoletosDisponibles());
                case 2 -> {
                    String cantidadStr = in.readLine();
                    int cantidad;
                    try {
                        cantidad =
Integer.parseInt(cantidadStr);
                    } catch (NumberFormatException e) {
                        out.println("Cantidad no válida.");
                        continue;
                    }
                    out.println(cine.comprarBoleto(nombreClien
te, cantidad));
                }
                case 3 -> {
                    out.println("Gracias por su compra. ¡Hasta
luego!");
                    seguir = false;

```

```

        System.out.println(nombreCliente + " ha
salido.");
    }
    default -> out.println("Opción no válida.");
}
}
} catch (IOException e) { // Se utiliza un catch para
manejar excepciones
    System.err.println("Error en la conexión con el
cliente: " + e.getMessage());
} finally {
    try { // Se utiliza un try-catch para manejar
excepciones
        socket.close(); // Se cierra el socket
    } catch (IOException e) {
        System.err.println("Error al cerrar el socket: " +
e.getMessage());
    }
}
}
}

public class ServidorCine { // Se crea una clase ServidorCine
    public static void main(String[] args) { // Se crea un metodo
main
        String ipServidor = "192.168.1.74"; //localhost"; // Se
crea una variable ipServidor
        int puerto = 8889; // Se crea una variable puerto
        Cine cine = new Cine(30); // Se crea un objeto de la clase
Cine
        ThreadPoolExecutor pool = (ThreadPoolExecutor)
Executors.newFixedThreadPool(3);

        new ReposicionBoletos(cine).start(); // Se crea un objeto
de la clase ReposicionBoletos y se inicia

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

```

```

        System.out.println("Servidor de cine iniciado en " +
ipServidor + ":" + puerto);

        while (true) { // Se crea un ciclo while
            Socket socketCliente = serverSocket.accept();
            pool.execute(new HiloCliente(socketCliente,
cine));
        }
    } catch (IOException e) { // Se utiliza un catch para
manejar excepciones
        System.err.println("Error en el servidor: " +
e.getMessage());
    }
}
}

```

## Código Fuente MultiClienteCine.java

```

/*
 * Nombre: Cardoso Osorio Atl Yosafat
 * Grupo: 7CM1
 * Materia: Sistemas Distribuidos
 * Practica 4: Modelo Multicliente/MultiServidor
 * Fecha: 17 de Marzo del 2025
 */
import java.awt.*;
import java.io.*;
import java.net.*;
import javax.swing.*;

public class MultiClienteCine { // Clase MultiClienteCine
    private final int PUERTO = 8888; // Puerto del balanceador de
carga
    private Socket socket; // Socket del cliente

```

```

    private BufferedReader entrada;
    private PrintWriter salida; // PrintWriter para enviar
mensajes al servidor
    private final JFrame frame; // JFrame para la interfaz grafica
    private final JTextField nombreField;
    private final JTextArea displayArea;
    private final JButton verBoletosBtn, comprarBtn, salirBtn;

    public MultiClienteCine() {
        frame = new JFrame("Cliente Cine"); // Se crea un objeto
de la clase JFrame
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //
Se establece la operacion por defecto al cerrar la ventana
        frame.setSize(400, 300); // Se establece el tamaño de la
ventana
        frame.setLayout(new BorderLayout()); // Se establece el
Layout de la ventana

        JPanel topPanel = new JPanel(); // Se crea un objeto de la
clase JPanel
        topPanel.setLayout(new BorderLayout());
        nombreField = new JTextField();
        topPanel.add(new JLabel("Ingrese su nombre: "),
BorderLayout.WEST);
        topPanel.add(nombreField, BorderLayout.CENTER);

        JButton conectarBtn = new JButton("Conectar con
Sucursal");
        conectarBtn.addActionListener(e -> conectar());
        topPanel.add(conectarBtn, BorderLayout.EAST);

        frame.add(topPanel, BorderLayout.NORTH);

        displayArea = new JTextArea();
        displayArea.setEditable(false);
        frame.add(new JScrollPane(displayArea),
BorderLayout.CENTER);

        JPanel sidePanel = new JPanel();

```

```

        sidePanel.setLayout(new GridLayout(3, 1));

        verBoletosBtn = new JButton("Ver Boletos");
        verBoletosBtn.addActionListener(e -> verBoletos());
        sidePanel.add(verBoletosBtn);

        comprarBtn = new JButton("Comprar");
        comprarBtn.addActionListener(e -> comprarBoletos());
        sidePanel.add(comprarBtn);

        salirBtn = new JButton("Salir");
        salirBtn.addActionListener(e -> salir());
        sidePanel.add(salirBtn);

        frame.add(sidePanel, BorderLayout.EAST);
        frame.setVisible(true);
    }

    private void conectar() { // Metodo conectar
        String nombre = nombreField.getText().trim();
        if (nombre.isEmpty()) { // Se verifica si el nombre esta
vacio
            JOptionPane.showMessageDialog(frame, "Por favor,
ingrese su nombre.", "Error", JOptionPane.ERROR_MESSAGE);
            return;
        }

        // Seleccionar servidor (sucursal)
        String[] servidores = {"Sucursal Norte (192.168.1.74)",
"Sucursal Sur (192.168.1.82)"};
        String sucursalSeleccionada = (String)
JOptionPane.showInputDialog( // Se crea un cuadro de dialogo
        frame,
        "Seleccione una sucursal", //
        "Conexión al Cine",
        JOptionPane.QUESTION_MESSAGE,
        null,
        servidores,
        servidores[0]

```

```

    );

    if (sucursalSeleccionada == null) { // Se verifica si no
se selecciono una sucursal
        JOptionPane.showMessageDialog(frame, "Debe seleccionar
una sucursal", "Error", JOptionPane.ERROR_MESSAGE);
        return;
    }

    String ipServidor = sucursalSeleccionada.contains("1") ?
"192.168.1.74" : "192.168.1.82"; // Se selecciona la ip del
servidor

    try {
        socket = new Socket(ipServidor, PUERTO); // Se crea un
objeto de la clase Socket
        entrada = new BufferedReader(new
InputStreamReader(socket.getInputStream())); // Se crea un objeto
de la clase BufferedReader
        salida = new PrintWriter(socket.getOutputStream(),
true);

        salida.println(nombre); // Enviar nombre al servidor
        displayArea.append(entrada.readLine() + "\n"); //
Mostrar mensaje del servidor
    } catch (IOException e) { // Se utiliza un catch para
manejar excepciones
        JOptionPane.showMessageDialog(frame, "No se pudo
conectar al servidor.", "Error", JOptionPane.ERROR_MESSAGE);
        System.exit(1);
    }
}

private void verBoletos() { // Metodo verBoletos
    salida.println("1");
    try {
        displayArea.append("Servidor: " + entrada.readLine() +
"\n"); // Se muestra el mensaje del servidor
    } catch (IOException e) {

```



```

        displayArea.append("Error al recibir información.\n");
// Se muestra un mensaje de error
    }
}

private void comprarBoletos() { // Metodo comprarBoletos
    String cantidadStr = JOptionPane.showInputDialog(frame,
    "Ingrese la cantidad de boletos:");
    if (cantidadStr == null) return;
    salida.println("2");
    salida.println(cantidadStr);
    try {
        displayArea.append("Servidor: " + entrada.readLine() +
"\n");
    } catch (IOException e) {
        displayArea.append("Error al recibir respuesta.\n");
    }
}

private void salir() { // Metodo salir
    salida.println("3");
    try {
        displayArea.append("Servidor: " + entrada.readLine() +
"\n");
        socket.close();
    } catch (IOException e) {
        displayArea.append("Error al cerrar la conexión.\n");
    }
    frame.dispose();
}

public static void main(String[] args) { // Metodo main
    SwingUtilities.invokeLater(MultiClienteCine::new); // Se
    crea un objeto de la clase MultiClienteCine
}
}

```

## Código Fuente BalanceadorCarga.java

```
/*
 * Nombre: Cardoso Osorio Atl Yosafat
 * Grupo: 7CM1
 * Materia: Sistemas Distribuidos
 * Practica 4: Modelo Multicliente/MultiServidor
 * Fecha: 17 de Marzo del 2025
 */
import java.io.*;
import java.net.*;
import java.util.*;

public class BalanceadorCarga { // Clase BalanceadorCarga
    private static final int PUERTO_BALANCEADOR = 8888; // Puerto
    del balanceador de carga
    private static final List<String> SERVIDORES =
Arrays.asList("192.168.1.74", "192.168.1.82");

    public static void main(String[] args) { // Metodo main de la
    clase BalanceadorCarga
        try (ServerSocket serverSocket = new
ServerSocket(PUERTO_BALANCEADOR)) { // Se crea un objeto de la
    clase ServerSocket
            System.out.println("Balanceador de carga iniciado en
el puerto " + PUERTO_BALANCEADOR);

            while (true) { // Se crea un ciclo infinito
                try { // Se utiliza un try-catch para manejar
    excepciones
                    Socket clienteSocket = serverSocket.accept();
                    System.out.println("Conexión entrante desde: "
+ clienteSocket.getInetAddress());

                    String servidorElegido =
seleccionarServidor();
                    System.out.println("Conexión redirigida al
servidor: " + servidorElegido);
```

```

        new Thread(new
ManejadorConexion(clienteSocket, servidorElegido, 8889)).start();
    } catch (IOException e) {
        System.err.println("Error al aceptar conexión
del cliente: " + e.getMessage());
    }
}
} catch (IOException e) {
    System.err.println("Error al iniciar el balanceador de
carga: " + e.getMessage());
}
}

private static String seleccionarServidor() {
    // El balanceador redirige aleatoriamente a uno de los
servidores disponibles,
    //en caso de tener mucha carga en el solicitado.
    Random rand = new Random();
    return SERVIDORES.get(rand.nextInt(SERVIDORES.size())); //
Se regresa un servidor aleatorio
}
}

class ManejadorConexion implements Runnable {
    private final Socket clienteSocket;
    private final String servidorDestino;
    private final int puertoServidor;

    public ManejadorConexion(Socket clienteSocket, String
servidorDestino, int puertoServidor) {
        this.clienteSocket = clienteSocket;
        this.servidorDestino = servidorDestino;
        this.puertoServidor = puertoServidor;
    }

    @Override
    public void run() {

```

```

        try (Socket servidorSocket = new Socket(servidorDestino,
puertoServidor)) {
            InputStream clienteInput =
clienteSocket.getInputStream();
            OutputStream clienteOutput =
clienteSocket.getOutputStream();
            InputStream servidorInput =
servidorSocket.getInputStream();
            OutputStream servidorOutput =
servidorSocket.getOutputStream();

            Thread clienteAServidor = new Thread(new
Proxy(clienteInput, servidorOutput));
            Thread servidorACliente = new Thread(new
Proxy(servidorInput, clienteOutput));

            clienteAServidor.start();
            servidorACliente.start();

            clienteAServidor.join();
            servidorACliente.join();
        } catch (IOException e) {
            System.err.println("Error al conectar con el servidor
" + servidorDestino + ": " + e.getMessage());
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.err.println("Error en la ejecución del hilo de
conexión: " + e.getMessage());
        } finally {
            try {
                clienteSocket.close();
            } catch (IOException e) {
                System.err.println("Error al cerrar el socket del
cliente: " + e.getMessage());
            }
        }
    }
}

```

```

class Proxy implements Runnable {
    private final InputStream in;
    private final OutputStream out;

    public Proxy(InputStream in, OutputStream out) {
        this.in = in;
        this.out = out;
    }

    @Override
    public void run() {
        try {
            byte[] buffer = new byte[1024];
            int bytesLeidos;
            while ((bytesLeidos = in.read(buffer)) != -1) {
                out.write(buffer, 0, bytesLeidos);
                out.flush();
            }
        } catch (IOException e) {
            System.err.println("Error de I/O en Proxy: " +
e.getMessage());
        }
    }
}

```