# INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

## UNIDAD DE APRENDIZAJE

SISTEMAS DISTRIBUIDOS

## PRACTICA No. 4

MODELO MULTICLIENTE - MULTISERVIDOR


## ALUMNO

GÓMEZ GALVAN DIEGO YAEL


## GRUPO

7CM1

## PROFESOR

CARRETO ARELLANO CHADWICK

## FECHA DE ENTREGA

19 DE MARZO DE 2025

# INDICE

# ANTESCEDENTES

El modelo multi-cliente y multi-servidor es una arquitectura ampliamente utilizada en sistemas distribuidos, donde múltiples clientes pueden interactuar con múltiples servidores simultáneamente para realizar diversas operaciones. Este modelo permite la descentralización de tareas, optimizando la distribución de la carga de trabajo y mejorando la eficiencia del sistema. Su implementación es fundamental en entornos donde la concurrencia y la escalabilidad son factores clave.

## MODELO MULTI-CLIENTE Y MULTI-SERVIDOR

En un sistema multi-cliente y multi-servidor, los clientes representan entidades que realizan solicitudes de servicio, mientras que los servidores procesan dichas solicitudes y responden de acuerdo con sus funciones específicas. En este contexto, los servidores pueden especializarse en diferentes tareas, como autenticación, gestión de datos, procesamiento de transacciones, entre otros. Esta especialización permite distribuir la carga de trabajo y evitar cuellos de botella en el sistema.

Este modelo se caracteriza por los siguientes aspectos:

- ✓ **Distribución de carga:** Al contar con múltiples servidores, las solicitudes de los clientes pueden ser atendidas en paralelo, reduciendo tiempos de espera y mejorando la capacidad de respuesta del sistema.
- ✓ **Escalabilidad:** Se pueden agregar más servidores a medida que aumenta la demanda sin afectar el rendimiento general.
- ✓ **Concurrencia:** Diferentes clientes pueden interactuar con distintos servidores al mismo tiempo sin interferencias, siempre que se gestione correctamente la sincronización.
- ✓ **Tolerancia a fallos:** Si un servidor falla, el sistema puede redirigir las solicitudes a otro servidor disponible, garantizando continuidad en el servicio.

A pesar de estas ventajas, un desafío fundamental en este modelo es la correcta sincronización del acceso a los recursos compartidos, evitando conflictos y garantizando la coherencia de los datos. Para abordar este problema, se emplean diferentes algoritmos de sincronización.

## ALGORITMOS DE SINCRONIZACIÓN EN SISTEMAS MULTI-CLIENTE Y MULTI-SERVIDOR

La sincronización es un aspecto crítico en sistemas distribuidos donde múltiples clientes pueden intentar acceder y modificar los mismos recursos de manera simultánea. Existen diversos algoritmos que permiten gestionar la concurrencia y evitar inconsistencias en los datos:

- ✓ **Exclusión Mutua con Bloqueo (Mutex):** Utiliza mecanismos de bloqueo para garantizar que solo un cliente pueda acceder a un recurso compartido a la vez. Se puede implementar a través de:
    - **Bloqueo de hilos:** Se emplean estructuras como synchronized en Java para evitar accesos simultáneos.
    - **Semáforos:** Permiten regular el número de clientes que pueden acceder a un recurso de manera simultánea, utilizando valores enteros para controlar el acceso.
- ✓ **Monitores:** Son estructuras de alto nivel que combinan exclusión mutua y sincronización condicional. En Java, los monitores se implementan de forma nativa mediante synchronized y wait/notify.
- ✓ **Algoritmos de Exclusión Mutua Distribuida:** Se utilizan cuando los recursos compartidos están distribuidos en múltiples servidores. Algunos enfoques incluyen:
    - **Algoritmo de Ricart-Agrawala:** Basado en el intercambio de mensajes para coordinar el acceso a un recurso crítico.
    - **Algoritmo de Token Ring:** Utiliza un token que circula entre los procesos, otorgando permiso de acceso a quien lo posee.
- ✓ **Control de Transacciones con Locks de Base de Datos:** En sistemas que manejan persistencia de datos, se emplean bloqueos a nivel de registros o tablas para garantizar la consistencia de la información.
- ✓ **Sincronización basada en Wait-Notify:** Un proceso espera hasta que otro proceso complete una operación antes de continuar. Esto es útil para garantizar que las actualizaciones en los recursos sean visibles para todos los clientes antes de que realicen nuevas operaciones.
- ✓ **Optimistic Concurrency Control (OCC):** En lugar de bloquear el acceso, permite múltiples lecturas y verifica posibles conflictos antes de confirmar cambios. Es eficiente en escenarios con baja probabilidad de colisiones.

**Importancia de la Sincronización en Sistemas Distribuidos**

La correcta implementación de algoritmos de sincronización es fundamental para evitar problemas como:

- ✓ **Condiciones de carrera:** Cuando múltiples clientes intentan modificar el mismo recurso simultáneamente sin una coordinación adecuada.
- ✓ **Inconsistencias en los datos:** Cuando diferentes servidores tienen copias desactualizadas de la información debido a accesos descoordinados.

- ✓ **Deadlocks:** Cuando dos o más procesos quedan bloqueados esperando la liberación de un recurso que otro proceso posee.

El modelo multi-cliente y multi-servidor, combinado con una correcta estrategia de sincronización, permite diseñar sistemas robustos y eficientes, garantizando la integridad de los datos y optimizando la experiencia de los usuarios finales.

# PLANTEAMIENTO DEL PROBLEMA

En los sistemas distribuidos modernos, el acceso simultáneo a recursos compartidos plantea desafíos significativos en términos de coherencia y sincronización de datos. A medida que el número de clientes que interactúan con el sistema aumenta, también lo hace la posibilidad de colisiones en el acceso a recursos críticos, lo que puede generar inconsistencias en la información y afectar la integridad del sistema.

Uno de los principales problemas en los entornos de múltiples clientes y servidores es la concurrencia en la ejecución de operaciones críticas, tales como consultas, modificaciones y transacciones sobre los mismos datos. Sin un mecanismo adecuado de sincronización, podrían ocurrir situaciones en las que dos o más clientes intenten modificar el mismo recurso simultáneamente, provocando resultados inesperados o pérdidas de datos.

Este problema se ve agravado cuando se implementa una arquitectura con múltiples servidores, ya que la coordinación entre ellos debe garantizar la consistencia de los datos sin generar bloqueos innecesarios o pérdida de rendimiento. Para ello, es necesario implementar mecanismos de sincronización eficientes que permitan gestionar el acceso concurrente a los recursos de manera ordenada.

En un sistema bancario, donde múltiples clientes acceden simultáneamente a diferentes operaciones, estos problemas de concurrencia y sincronización se vuelven aún más críticos. Un sistema bancario distribuido permite a los usuarios realizar consultas de saldo, depósitos, retiros y transferencias de manera remota, interactuando con servidores específicos encargados de procesar cada una de estas solicitudes.

Sin una adecuada sincronización, podrían surgir problemas como:

- ✓ **Condiciones de carrera**, donde múltiples clientes intentan modificar el saldo de una cuenta al mismo tiempo, generando inconsistencias.
- ✓ **Pérdida de transacciones**, cuando las operaciones no se reflejan correctamente debido a accesos simultáneos.
- ✓ **Desactualización de datos**, donde un cliente consulta un saldo antes de que otra transacción haya sido procesada completamente.
- ✓ **Bloqueos innecesarios**, que pueden afectar el rendimiento general del sistema.

Al pasar de un modelo de un solo servidor y múltiples clientes a un modelo multi-cliente y multi-servidor, la gestión de concurrencia se vuelve aún más compleja, requiriendo estrategias avanzadas de sincronización para evitar fallos en la integridad de los datos y garantizar una experiencia bancaria segura y eficiente.

# PROPUESTA DE SOLUCIÓN

El código desarrollado responde a esta problemática mediante la implementación de un modelo multi-cliente y multi-servidor distribuido, donde diferentes clientes pueden acceder simultáneamente a múltiples servidores que cumplen funciones específicas. Además, se incorpora un servidor de sincronización, cuya función principal es garantizar que solo un cliente pueda acceder a un recurso crítico a la vez. Este servidor coordina los accesos y utiliza un mecanismo de espera y notificación para evitar colisiones en las operaciones concurrentes.

El sistema propuesto divide las responsabilidades en múltiples servidores:

- ✓ **Servidor de Autenticación:** Se encarga de validar la identidad de los clientes antes de permitirles realizar operaciones.
- ✓ **Servidor de Saldo:** Administra la consulta y actualización de saldos de los clientes.
- ✓ **Servidor de Transacciones:** Procesa las operaciones de depósito, retiro y transferencia, asegurando que las modificaciones en los datos sean consistentes.
- ✓ **Servidor de Sincronización:** Garantiza que solo un cliente pueda realizar una operación crítica sobre un mismo recurso a la vez, utilizando un mecanismo de exclusión mutua con espera y notificación.

Mediante la implementación de este sistema, se soluciona el problema de la concurrencia y se asegura que las operaciones sean ejecutadas en el orden correcto, evitando inconsistencias en los datos. Este enfoque permite mejorar la escalabilidad del sistema sin comprometer la integridad de la información, proporcionando una solución eficiente para entornos donde múltiples clientes requieren acceso simultáneo a recursos compartidos.

# MATERIALES Y METODOS

Para la realización de modelo, requieran los siguientes recursos:

- ✓ Lenguaje de programación: Java
- ✓ Entorno de desarrollo: NetBeans IDE 21
- ✓ Librerías utilizadas:
  - Java.net para el manejo de sockets TCP
  - Java.io para la lectura y escritura de datos entre el cliente y servidor
  - Java.util.concurrent para la gestión de hilos y concurrencia
- ✓ Sistema operativo: Windows 11
- ✓ Protocolo de comunicación: TCP (Transmission Control Protocol), que garantiza una conexión estable entre el cliente y servidor

# DESARROLLO DE SOLUCIÓN

Para abordar los problemas identificados, se implementó un sistema basado en el modelo multi-cliente y multi-servidor con sincronización de acceso a los recursos. Este sistema está compuesto por los siguientes componentes

### Clase Servidor de Autenticación

El Servidor de Autenticación es el encargado de validar las credenciales de los clientes antes de permitirles acceder al sistema bancario. Se ejecuta en el puerto 3000 y utiliza un ServerSocket para recibir conexiones entrantes. Para almacenar las cuentas de los usuarios, se utiliza un HashMap, donde las claves son los números de cuenta y los valores son los NIPs asociados.

Cuando el servidor inicia, carga un conjunto de cuentas con sus respectivos NIPs en la estructura HashMap. Luego, entra en un bucle infinito donde espera conexiones de clientes. Cuando un cliente se conecta, se crea un Socket y se imprime un mensaje en la consola con la dirección IP y el puerto del cliente. Para manejar múltiples clientes simultáneamente sin bloquear el servidor principal, se crea un nuevo hilo que ejecuta la autenticación de manera independiente mediante la clase MAutenticacion.

```
public static void main(String[] args){
    cuentas.put("1234567890", 1234);
    cuentas.put("2468024680", 2468);
    cuentas.put("3692581470", 3692);
    cuentas.put("4826048260", 4826);
    cuentas.put("6284062840", 6284);

    try{
        ServerSocket ss = new ServerSocket(3000);
        System.out.println("Servidor de autenticacion activo en el puerto 3000");

        while(true){
            Socket cl = ss.accept();
            System.out.println("Cliente conectado desde: " + cl.getInetAddress() + ":" + cl.getPort());
            new Thread(new MAutenticacion(cl)).start();
        }

    }catch(IOException e){
        System.out.println("Error en el Servidor de Autenticación "+ e.getMessage());
    }
}
```

La clase interna MAutenticacion implementa Runnable, lo que permite ejecutar el proceso en un hilo separado. Su función es recibir los datos del cliente, validarlos y enviar una respuesta indicando si la autenticación fue exitosa o no. Para la comunicación con el cliente, se crean flujos de entrada y salida. El servidor espera que el cliente envíe primero el número de cuenta y luego el NIP, que es leído como una cadena de texto y convertido a un valor entero.

El proceso de validación se realiza consultando el HashMap. Si la cuenta existe y el NIP ingresado coincide con el almacenado, el servidor responde con el mensaje "Autenticación

Exitosa", permitiendo que el cliente continúe con sus operaciones bancarias. Si la cuenta no existe o el NIP es incorrecto, responde con "Error en autenticación", denegando el acceso al cliente.

Si ocurre un error en la comunicación, se muestra un mensaje de advertencia en la consola indicando la dirección IP y el puerto del cliente afectado. Independientemente del resultado de la autenticación, el servidor cierra la conexión con el cliente en el bloque finally para liberar recursos y asegurarse de que no queden conexiones abiertas innecesariamente.

```
private static class MAutenticacion implements Runnable{
    private Socket cl;
    public MAutenticacion(Socket cl){
        this.cl = cl;
    }
    @Override
    public void run() {
        try{
            BufferedReader ent = new BufferedReader(new InputStreamReader(cl.getInputStream()));
            PrintWriter sal = new PrintWriter(cl.getOutputStream(),true);
            String cuenta = ent.readLine();
            String nipStr = ent.readLine();
            int nip = Integer.parseInt(nipStr);
            if(cuentas.containsKey(cuenta) && cuentas.get(cuenta).equals(nip)){
                sal.println("Autenticacion Exitosa");
            }else{
                sal.println("Error en autenticación");
            }
        }catch(IOException e){
            System.err.println("Error con cliente: " + cl.getInetAddress() + ":" + cl.getPort());
        } finally {
            try {
                cl.close();
                System.out.println("Cliente desconectado.");
            } catch (IOException e) {
                System.err.println("Error al cerrar la conexión con el cliente.");
            }
        }
    }
}
```

### Clase Servidor Saldo

El Servidor de Saldo es responsable de gestionar la consulta y actualización de los saldos de las cuentas en el sistema bancario distribuido. Se ejecuta en el puerto 4000 y utiliza un ServerSocket para aceptar conexiones de clientes. A diferencia del servidor de autenticación, este servidor maneja operaciones dinámicas sobre los saldos de las cuentas, permitiendo a los clientes consultar su saldo actual o actualizarlo después de realizar una transacción. Para garantizar el acceso concurrente seguro a los datos, se utiliza una estructura ConcurrentHashMap, que permite múltiples accesos simultáneos sin generar problemas de concurrencia.

Cuando el servidor inicia, carga un conjunto de cuentas con sus respectivos saldos en el ConcurrentHashMap. Luego, entra en un bucle infinito donde espera conexiones de clientes. Cuando un cliente se conecta, se crea un Socket, y el servidor imprime en la consola la dirección IP y el puerto del cliente. Para procesar cada solicitud sin bloquear el servidor principal, se crea un nuevo hilo (Thread), que ejecuta la clase MSaldo, permitiendo manejar múltiples clientes simultáneamente.

```
public static void main(String[] args){
    cuentas.put("1234567890", 9500.0);
    cuentas.put("2468024680", 10000.0);
    cuentas.put("3692581470", 6348.3);
    cuentas.put("4826048260", 100.80);
    cuentas.put("6284062840", 1150000.0);

    try{
        ServerSocket ss = new ServerSocket(4000);
        System.out.println("Servidor de saldo activo en el puerto 4000");

        while(true){
            Socket cl = ss.accept();
            System.out.println("Cliente conectado desde: " + cl.getInetAddress() + ":" + cl.getPort());
            new Thread(new MSaldo(cl)).start();
        }
    }catch(IOException e){
        e.printStackTrace();
    }
}
```

La clase interna MSaldo implementa Runnable, lo que permite ejecutar el proceso en un hilo separado. Su función principal es recibir las solicitudes de los clientes, procesarlas y enviar la respuesta correspondiente. Para la comunicación con el cliente, se utilizan flujos de entrada (BufferedReader) y salida (PrintWriter). El servidor espera que el cliente envíe un código de operación ("C" para consultar saldo o "A" para actualizarlo) seguido del número de cuenta.

Si la operación es una consulta ("C"), el servidor verifica si la cuenta existe en el ConcurrentHashMap. Si la cuenta es válida, envía el saldo actual al cliente. En caso contrario, responde con "Cuenta no encontrada". Si la operación es una actualización de saldo ("A"), el servidor recibe el nuevo saldo desde el cliente, verifica que la cuenta exista y actualiza el valor en el ConcurrentHashMap. Si la actualización es exitosa, responde con "Saldo actualizado correctamente", de lo contrario, indica que la cuenta no existe y no se puede actualizar el saldo.

Si ocurre un error en la comunicación con el cliente, el servidor imprime un mensaje de advertencia en la consola, indicando la dirección IP y el puerto del cliente afectado. Finalmente, la conexión con el cliente se cierra en el bloque finally para liberar recursos y evitar bloqueos.

```
private static class MSaldo implements Runnable{
    private Socket cl;
    public MSaldo(Socket cl){
        this.cl = cl;
    }
    @Override
    public void run() {
        try{
            BufferedReader ent = new BufferedReader(new InputStreamReader(cl.getInputStream()));
            PrintWriter sal = new PrintWriter(cl.getOutputStream(),true);
            String opc = ent.readLine();
            String cuenta = ent.readLine();

            switch(opc){
                case "C":
                        if(cuentas.containsKey(cuenta)){
                            sal.println(cuentas.get(cuenta));
                        }else{
                            sal.println("Cuenta no encontrada");
                        }
                        break;
                case "A":
                        double nuevoSaldo = Double.parseDouble(ent.readLine());
                        if (cuentas.containsKey(cuenta)) {
                            cuentas.put(cuenta, nuevoSaldo);
                            sal.println("Saldo actualizado correctamente.");
                        } else {
                            sal.println("Cuenta no encontrada, no se puede actualizar saldo.");
                        }
                        break;
                default:sal.println("Operacion no valida");
                        break;
            }
        }catch(IOException e){
            System.err.println("Error con cliente: " + cl.getInetAddress() + ":" + cl.getPort());
        } finally {
            try {
                cl.close();
                System.out.println("Cliente desconectado.");
            } catch (IOException e) {
                System.err.println("Error al cerrar la conexión con el cliente.");
            }
        }
    }
}
```

### Clase Servidor Transacciones

El Servidor de Transacciones es el componente encargado de gestionar las operaciones bancarias de los clientes, como depósitos, retiros y transferencias. Se ejecuta en el puerto 5000 y permite que múltiples clientes realicen transacciones simultáneamente sin comprometer la coherencia de los datos. Para garantizar que solo un cliente modifique el saldo de una cuenta a la vez, este

servidor implementa un mecanismo de sincronización basado en comunicación con un Servidor de Sincronización.

Cuando el servidor inicia, crea un ServerSocket en el puerto 5000 y entra en un bucle infinito donde espera conexiones de clientes. Cada vez que un cliente se conecta, se imprime un mensaje con su dirección IP y puerto, y se crea un hilo independiente mediante la clase MTransacciones, lo que permite manejar múltiples transacciones en paralelo sin bloquear el servidor principal.

```
public static void main(String[] args){
    try{
        ServerSocket ss = new ServerSocket(5000);
        System.out.println("Servidor de transacciones activo en el puerto 5000");

        while(true){
            Socket cl = ss.accept();
            System.out.println("Cliente conectado desde: " + cl.getInetAddress() + ":" + cl.getPort());
            new Thread(new MTransacciones(cl)).start();
        }
    }catch(IOException e){
        System.out.println("Error al iniciar el Servidor de Transacciones: "+e.getMessage());
    }
}
```

La clase MTransacciones implementa Runnable y gestiona cada solicitud de transacción. Primero, recibe el tipo de operación (D para depósito, R para retiro o T para transferencia) junto con el número de cuenta. Luego, se consulta el saldo actual de la cuenta comunicándose con el Servidor de Saldo, que opera en el puerto 4000. Antes de proceder con la modificación del saldo, se solicita acceso al Servidor de Sincronización, que actúa como un control de concurrencia asegurando que solo una transacción se realice sobre la misma cuenta en un momento dado.

Si la operación es un depósito (D), el monto enviado por el cliente se suma al saldo actual y se actualiza en el Servidor de Saldo. Si la operación es un retiro (R), el servidor verifica que la cuenta tenga fondos suficientes antes de descontar el monto y actualizar el saldo. Para una transferencia (T), primero se consulta el saldo de la cuenta de origen y la de destino. Si la cuenta de origen tiene fondos suficientes, se descuentan del saldo de esa cuenta y se suman al saldo de la cuenta de destino. Si la actualización de ambos saldos es exitosa, la transacción se confirma; de lo contrario, se informa al cliente que hubo un error.

```
private static class MTransacciones implements Runnable{
    private Socket cl;
    public MTransacciones(Socket cl){
        this.cl = cl;
    }
    @Override
    public void run() {
        try{
            BufferedReader ent = new BufferedReader(new InputStreamReader(cl.getInputStream()));
            PrintWriter sal = new PrintWriter(cl.getOutputStream(),true);

            String opc = ent.readLine();
            String cuenta = ent.readLine();
            double saldoActual = consultarSaldo(cuenta);
            if (!solicitarAccesoSincronizacion()) {
                sal.println("Error: No se pudo obtener acceso a la transacción.");
                return;
            }

        switch(opc){
            case "D": // Depósito
                double montoD = Double.parseDouble(ent.readLine());
                if (montoD > 0) {
                    saldoActual += montoD;
                    if (actualizarSaldo(cuenta, saldoActual)) {
                        sal.println("Deposito exitoso. Nuevo saldo: $" + saldoActual);
                    } else {
                        sal.println("Error al actualizar saldo en el servidor de saldo.");
                    }
                } else {
                    sal.println("Error: El monto debe ser mayor a 0.");
                }
                break;

            case "R": // Retiro
                double montoR = Double.parseDouble(ent.readLine());
                if (saldoActual >= montoR && montoR > 0) {
                    saldoActual -= montoR;
                    if (actualizarSaldo(cuenta, saldoActual)) {
                        sal.println("Retiro exitoso. Nuevo saldo: $" + saldoActual);
                    } else {
                        sal.println("Error al actualizar saldo en el servidor de saldo.");
                    }
                } else {
                    sal.println("Error: Saldo insuficiente o monto inválido.");
                }
                break;
```

```
            case "T": // Transferencia
                sal.println(saldoActual);
                String cuentaDestino = ent.readLine();
                double montoT = Double.parseDouble(ent.readLine());
                double saldoDestino = consultarSaldo(cuentaDestino);

                if (saldoDestino < 0) {
                    sal.println("Error: Cuenta de destino no encontrada.");
                } else if (saldoActual >= montoT && montoT > 0) {
                    saldoActual -= montoT;
                    saldoDestino += montoT;

                    if (actualizarSaldo(cuenta, saldoActual) && actualizarSaldo(cuentaDestino, saldoDestino)) {
                        sal.println("Transferencia exitosa. Nuevo saldo: $" + saldoActual);
                    } else {
                        sal.println("Error al actualizar los saldos en el servidor de saldo.");
                    }
                } else {
                    sal.println("Error: Saldo insuficiente, cuenta inexistente o monto inválido.");
                }
                break;

            default:
                sal.println("Operación no válida.");
                break;
        }
        liberarAccesoSincronizacion();

        }catch(IOException e){
            System.err.println("Error en transacción: " + e.getMessage());
        } catch (InterruptedException ex) {
            Logger.getLogger(ServidorTransacciones.class.getName()).log(Level.SEVERE, null, ex);
        } finally {
            try {
                cl.close();
                System.out.println("Cliente desconectado del servidor de transacciones.");
            } catch (IOException e) {
                System.err.println("Error al cerrar la conexión con el cliente.");
            }
        }
    }
```

```java
private static double consultarSaldo(String cuenta) {
    try (Socket sSaldo = new Socket("localhost", 4000);
         BufferedReader entSaldo = new BufferedReader(new InputStreamReader(sSaldo.getInputStream()));
         PrintWriter salSaldo = new PrintWriter(sSaldo.getOutputStream(), true)) {

        salSaldo.println("C");
        salSaldo.println(cuenta);
        salSaldo.flush();

        String respuesta = entSaldo.readLine();
        if (respuesta == null || respuesta.equals("Cuenta no encontrada")) {
            return -1; // Indicar que la cuenta no existe
        }
        return Double.parseDouble(respuesta);

    } catch (IOException | NumberFormatException e) {
        System.err.println("Error al consultar el saldo en el Servidor de Saldo: " + e.getMessage());
        return -1; // Indicar que hubo un error
    }
}
```

```java
private static boolean actualizarSaldo(String cuenta, double nuevoSaldo) {
    try (Socket sSaldo = new Socket("localhost", 4000);
         PrintWriter salSaldo = new PrintWriter(sSaldo.getOutputStream(), true);
         BufferedReader entSaldo = new BufferedReader(new InputStreamReader(sSaldo.getInputStream()))) {

        salSaldo.println("A");
        salSaldo.println(cuenta);
        salSaldo.println(nuevoSaldo);
        salSaldo.flush();

        String respuesta = entSaldo.readLine();
        return respuesta != null && respuesta.equals("Saldo actualizado correctamente.");

    } catch (IOException e) {
        System.err.println("Error al actualizar saldo en el Servidor de Saldo: " + e.getMessage());
        return false;
    }
}
```

El acceso al Servidor de Sincronización se maneja mediante el método solicitarAccesoSincronizacion(), que intenta obtener permiso antes de modificar los saldos. Si el servidor de sincronización responde con "P", la transacción puede proceder; si responde con "E", significa que otro cliente está operando sobre la misma cuenta, por lo que el cliente debe esperar antes de intentarlo nuevamente. Una vez que la transacción se completa, se libera el acceso llamando al método liberarAccesoSincronizacion(), permitiendo que otras operaciones puedan continuar.

```java
private static boolean solicitarAccesoSincronizacion() throws IOException, InterruptedException {
    try {
        Socket sSync = new Socket("localhost", 6000);
        BufferedReader entSync = new BufferedReader(new InputStreamReader(sSync.getInputStream()));
        PrintWriter salSync = new PrintWriter(sSync.getOutputStream(), true);

        salSync.println("S");

        String respuesta;
        while (true) {
            respuesta = entSync.readLine();
            if ("P".equals(respuesta)) {
                System.out.println("Entro a servidor de sincronizacion");
                return true;
            } else if ("E".equals(respuesta)) {
                System.out.println("En espera");
                Thread.sleep(500); // Espera un momento antes de reintentar
            }
        }

    } catch (IOException e) {
        System.out.println("Error al solicitar acceso al Servidor de Sincronización.");
    }
    return false;
}
```

```java
private static void liberarAccesoSincronizacion() {
    try (Socket sSync = new Socket("localhost", 6000);
         PrintWriter salSync = new PrintWriter(sSync.getOutputStream(), true)) {

        salSync.println("L");

    } catch (IOException e) {
        System.out.println("Error al liberar acceso en el Servidor de Sincronización.");
    }
}
```

El servidor maneja errores de conexión y garantiza que los clientes sean desconectados correctamente después de completar sus transacciones. Al distribuir las tareas entre diferentes servidores y aplicar un control de concurrencia, el Servidor de Transacciones permite que múltiples clientes operen sobre el sistema bancario sin generar inconsistencias en los datos. Su integración con el Servidor de Sincronización y el Servidor de Saldo asegura que las modificaciones de los saldos sean seguras y reflejen el estado correcto de las cuentas en todo momento.

### Clase Servidor Sincronización

El Servidor de Sincronización es el componente clave del sistema bancario distribuido que gestiona el acceso concurrente a los recursos críticos, garantizando que solo una transacción se ejecute a la vez sobre una cuenta bancaria específica. Se ejecuta en el puerto 6000 y utiliza un ServerSocket para recibir conexiones de los Servidores de Transacciones, quienes solicitan

permiso antes de modificar los saldos de las cuentas. Su propósito principal es evitar condiciones de carrera y asegurar la integridad de los datos en un entorno con múltiples clientes y servidores operando simultáneamente.

Cuando el servidor inicia, crea un ServerSocket en el puerto 6000 y entra en un bucle infinito donde espera conexiones. Cada vez que un servidor de transacciones solicita acceso, el Servidor de Sincronización acepta la conexión y crea un nuevo hilo mediante la clase GestorAcceso, lo que permite manejar múltiples solicitudes sin bloquear el servidor principal. Este diseño asegura que todas las solicitudes sean procesadas sin afectar el rendimiento del sistema.

```
public static void main(String[] args) {
    try (ServerSocket ss = new ServerSocket(6000)) {
        System.out.println("Servidor de Sincronizacion activo en el puerto 6000");

        while (true) {
            Socket cl = ss.accept();
            System.out.println("Cliente conectado desde: " + cl.getInetAddress() + ":" + cl.getPort());
            new Thread(new GestorAcceso(cl)).start();
        }
    } catch (IOException e) {
        System.err.println("Error al iniciar el Servidor de Sincronización: " + e.getMessage());
    }
}
```

La clase interna GestorAcceso implementa Runnable y maneja cada solicitud de sincronización. Cuando un servidor de transacciones se conecta, este envía un mensaje indicando que desea acceder a una cuenta ("S"). Si el mensaje recibido no es válido, el servidor responde con un mensaje de error y termina la conexión. Si la solicitud es válida, se ejecuta un bloque sincronizado (synchronized (lock)) sobre un objeto lock, el cual controla el acceso concurrente. Si el recurso ya está en uso, el servidor responde con "E" (Espera) y el proceso entra en un estado de espera (wait()), lo que significa que debe esperar a que la transacción en curso finalice antes de proceder.

Cuando el recurso está disponible, el servidor cambia la variable enUso a true y responde con "P" (Permiso), permitiendo que la transacción se realice. Una vez que el servidor de transacciones ha completado la operación, envía un mensaje "L" (Liberar), lo que indica que el recurso puede volver a estar disponible para otros procesos. En este punto, la variable enUso se restablece a false, y se notifica (notifyAll()) a los procesos en espera, permitiendo que otro servidor de transacciones continúe con su operación.

```java
private static class GestorAcceso implements Runnable {
    private final Socket cl;
    public GestorAcceso(Socket cl) {
        this.cl = cl;
    }
    @Override
    public void run() {
        try {
            BufferedReader ent = new BufferedReader(new InputStreamReader(cl.getInputStream()));
            PrintWriter sal = new PrintWriter(cl.getOutputStream(), true);
            String solicitud = ent.readLine();
            if (!"S".equals(solicitud)) {
                sal.println("ERROR: Solicitud no válida.");
                return;
            }
            synchronized (lock) {
                while (enUso) {
                    sal.println("E");
                    lock.wait();
                }
                enUso = true;
                sal.println("P");
            }
            String fin = ent.readLine();
            if ("L".equals(fin)) {
                synchronized (lock) {
                    enUso = false;
                    lock.notifyAll();
                }
            }
        } catch (IOException | InterruptedException e) {
            System.err.println("Error en comunicación con cliente: " + e.getMessage());
        }
```

El Servidor de Sincronización es esencial en el sistema multi-cliente y multi-servidor, ya que evita que múltiples clientes modifiquen simultáneamente el saldo de una cuenta, previniendo errores e inconsistencias en los datos. Su diseño basado en bloqueo y espera garantiza que las transacciones se ejecuten en orden y sin interferencias. Al utilizar hilos independientes para procesar cada solicitud, este servidor logra mantener un alto nivel de concurrencia sin comprometer la seguridad de las operaciones bancarias.

**Clase Cliente**

El Cliente es el componente que permite a los usuarios interactuar con el sistema bancario distribuido. Su función principal es establecer conexiones con los servidores correspondientes para autenticar al usuario, consultar su saldo, realizar depósitos, retiros y transferencias. Se ejecuta en la máquina del usuario y utiliza sockets para la comunicación con los distintos servidores del sistema.

Cuando el cliente inicia, solicita al usuario su número de cuenta y NIP, y establece una conexión con el Servidor de Autenticación en el puerto 3000. Se envían las credenciales al servidor y se

espera una respuesta. Si la autenticación es exitosa, el cliente puede continuar con las operaciones; de lo contrario, se cierra la conexión y finaliza la ejecución.

```
public static void main(String[] args) {
    int nip,opc = 0,montoR;
    double montoD;
    String cuentaD;

    try{
        //Conectando al servidor de autenticacion
        Socket sAut = new Socket("localhost",3000);
        BufferedReader entAut = new BufferedReader(new InputStreamReader(sAut.getInputStream()));
        PrintWriter salAut = new PrintWriter(sAut.getOutputStream(), true);
        Scanner scan = new Scanner(System.in);
        //Solicitando numero de cuenta
        System.out.print("Ingrese su numero de cuenta: ");
        String numCuenta = scan.nextLine();
        //Solicitando NIP
        System.out.print("Ingrese su NIP: ");
        nip = scan.nextInt();
        //Enviar info al servidor
        salAut.println(numCuenta);
        salAut.println(nip);
        salAut.flush();
        //Recibiendo respueta de servidor
        String autenticacion = entAut.readLine();
        System.out.println("\n"+autenticacion);
        //Si se autentico correctamente el cliente
        if(!autenticacion.startsWith("Autenticacion Exitosa")){
            sAut.close();
            return;
        }
        sAut.close();
```

Después de autenticarse, el cliente muestra un menú donde el usuario puede elegir entre consultar su saldo, depositar dinero, retirar dinero, realizar una transferencia o salir del sistema. Para cada opción seleccionada, el cliente establece una conexión con el servidor correspondiente:

1. Consulta de saldo: Se conecta al Servidor de Saldo en el puerto 4000, envía la solicitud de consulta junto con el número de cuenta y recibe el saldo disponible.

2. Depósito de dinero: Se conecta al Servidor de Transacciones en el puerto 5000, envía la solicitud de depósito junto con el número de cuenta y el monto a depositar, y recibe la confirmación de la operación.

3. Retiro de dinero: Se conecta al Servidor de Transacciones, envía la solicitud de retiro con el número de cuenta y el monto deseado, y recibe una respuesta indicando si la operación fue exitosa o si hay saldo insuficiente.

4. Transferencia de dinero: Se conecta al Servidor de Transacciones, consulta el saldo disponible, solicita al usuario el número de cuenta de destino y el monto a transferir, y

envía la solicitud al servidor. Se recibe una respuesta confirmando o rechazando la transacción.

```
System.out.println("\nSeleccione la opcion deseada:");
System.out.println("1. Consultar saldo");
System.out.println("2. Depositar dinero");
System.out.println("3. Retirar dinero");
System.out.println("4. Trasferencia");
System.out.println("5. Salir");
opc = scan.nextInt();
switch(opc){
    case 1: //Conectando al servidor de Saldo
        try{
            Socket sSaldo = new Socket("localhost",4000);
            BufferedReader entSaldo = new BufferedReader(new InputStreamReader(sSaldo.getInputStream()));
            PrintWriter salSaldo = new PrintWriter(sSaldo.getOutputStream(), true);
            //Enviando numero de cuenta
            salSaldo.println("C");
            salSaldo.println(numCuenta);
            salSaldo.flush();
            System.out.print("\nSaldo Actual: $"+entSaldo.readLine()+"\n");
        }catch(IOException e){
            System.out.println("Error al consutar el saldo");
        }
        break;
```

```
case 2: //Conectando al servidor de transacciones (Deposito)
        try{
            Socket sDeposito = new Socket("localhost",5000);
            BufferedReader entDeposito = new BufferedReader(new InputStreamReader(sDeposito.getInputStream()));
            PrintWriter salDeposito = new PrintWriter(sDeposito.getOutputStream(), true);

            System.out.print("Ingrese el monto a depositar: $");
            montoD = scan.nextDouble();

            salDeposito.println("D");
            salDeposito.println(numCuenta);
            salDeposito.println(montoD);
            salDeposito.flush();

            System.out.println(entDeposito.readLine());
        }catch(IOException e){
            System.out.println("Error al depositar dinero");
        }
        break;
```

```
case 3: //Conectando a servidor de transacciones (Retiros)
        try{
            Socket sRetiro = new Socket("localhost",5000);
            BufferedReader entRetiro = new BufferedReader(new InputStreamReader(sRetiro.getInputStream()));
            PrintWriter salRetiro = new PrintWriter(sRetiro.getOutputStream(), true);

            System.out.print("Ingrese el monto a retirar: $");
            montoR = scan.nextInt();

            salRetiro.println("R");
            salRetiro.println(numCuenta);
            salRetiro.println(montoR);
            salRetiro.flush();

            System.out.println(entRetiro.readLine());
        }catch(IOException e){
            System.out.println("Error al retirar dinero");
        }
        break;
case 4: //Conectando a servidor de transacciones (Transferencias)
        try{
            Socket sTrans = new Socket("localhost",5000);
            BufferedReader entTrans = new BufferedReader(new InputStreamReader(sTrans.getInputStream()));
            PrintWriter salTrans = new PrintWriter(sTrans.getOutputStream(), true);

            salTrans.println("T");
            salTrans.println(numCuenta);
            salTrans.flush();

            System.out.print("\nSaldo disponible: $"+entTrans.readLine());
            System.out.print("\nIngrese el numero de cuenta destino: ");
            scan.nextLine();
            cuentaD = scan.nextLine();
            salTrans.println(cuentaD);
            salTrans.flush();

            System.out.print("Ingresa el monto a depositar: $");
            montoD = scan.nextInt();
            salTrans.println(montoD);
            salTrans.flush();

            System.out.println(entTrans.readLine());

        }catch(IOException e){
            System.out.println("Error al retirar dinero");
        }
        break;
case 5: System.out.println("Sesion finalizada");
        break;
```

## RESULTADOS

Al correr todos los servidores, estos se inicializan y se colocan en espera de conexiones de clientes a su respectivo puerto

```
Servidor de autenticacion activo en el puerto 3000
Servidor de saldo activo en el puerto 4000
Servidor de Sincronizacion activo en el puerto 6000
Servidor de transacciones activo en el puerto 5000
```

Al ejecutar los clientes, estos se conectan al servidor de autenticación para ingresar sus números de cuenta y el nip. Aquí ejecutamos 3 clientes, donde inicialmente se conectan al servidor antes mencionado para autenticarse

```
Servidor de autenticacion activo en el puerto 3000
Cliente conectado desde: /127.0.0.1:50524
Cliente conectado desde: /127.0.0.1:50525
Cliente conectado desde: /127.0.0.1:50526
```

El primer cliente no ingreso correctamente sus dados, por lo que obtuvo lo siguiente

```
Ingrese su numero de cuenta: 2468024680
Ingrese su NIP: 3690

Error en autenticaci□n
```

El segundo y tercer cliente se autenticaron correctamente por lo que obtuvieron acceso al menú

```
Ingrese su numero de cuenta: 1234567890
Ingrese su NIP: 1234

Autenticacion Exitosa

Seleccione la opcion deseada:
1. Consultar saldo
2. Depositar dinero
3. Retirar dinero
4. Trasferencia
5. Salir
```

```
Ingrese su numero de cuenta: 2468024680
Ingrese su NIP: 2468

Autenticacion Exitosa

Seleccione la opcion deseada:
1. Consultar saldo
2. Depositar dinero
3. Retirar dinero
4. Trasferencia
5. Salir
```

Una vez autenticados, se desconectan de este servidor, para que puedan interactuar con el menú, según la solicitud que deseen realizar:

```
Servidor de autenticacion activo en el puerto 3000
Cliente conectado desde: /127.0.0.1:50524
Cliente conectado desde: /127.0.0.1:50525
Cliente conectado desde: /127.0.0.1:50526
Cliente desconectado.
Cliente desconectado.
Cliente desconectado.
```

Los 2 clientes conectados, solicitan ver su saldo, por lo que se conectan al servidor de Saldo para obtenerlo

```
Servidor de saldo activo en el puerto 4000
Cliente conectado desde: /127.0.0.1:50576
Cliente desconectado.
Cliente conectado desde: /127.0.0.1:50578
Cliente desconectado.
```

```
Saldo Actual: $9500.0
Seleccione la opcion deseada:
1. Consultar saldo
2. Depositar dinero
3. Retirar dinero
4. Trasferencia
5. Salir
```

```
Saldo Actual: $10000.0
Seleccione la opcion deseada:
1. Consultar saldo
2. Depositar dinero
3. Retirar dinero
4. Trasferencia
5. Salir
```

Para las transacciones, ya sea depositar, retirar o transferir, se realizar una sincronización a través del servidor de sincronización. El cliente 1 realiza un deposito, el cliente 2 realiza un retiro y el cliente 3 realiza una transferencia

```
Seleccione la opcion deseada:
1. Consultar saldo
2. Depositar dinero
3. Retirar dinero
4. Trasferencia
5. Salir
2
Ingrese el monto a depositar: $1500
```

```
Seleccione la opcion deseada:
1. Consultar saldo
2. Depositar dinero
3. Retirar dinero
4. Trasferencia
5. Salir
3
Ingrese el monto a retirar: $1000
Retiro exitoso. Nuevo saldo: $9000.0
```

```
Saldo disponible: $9000.0
Ingrese el numero de cuenta destino: 1234567890
Ingresa el monto a depositar: $6000
Transferencia exitosa. Nuevo saldo: $3000.0

Seleccione la opcion deseada:
1. Consultar saldo
2. Depositar dinero
3. Retirar dinero
4. Trasferencia
5. Salir
```

```
Saldo Actual: $11350.0

Seleccione la opcion deseada:
1. Consultar saldo
2. Depositar dinero
3. Retirar dinero
4. Trasferencia
5. Salir
1

Saldo Actual: $17350.0
```

Para estas transacciones, el servidor de sincronización nos ayudó a evitar problemas de sincronización entre los montos. En el caso de la transferencia, nos ayudó a que cuando el usuario que recibe la transferencia consulte su saldo, este se actualice correctamente y no muestren inconsistencias

# CONCLUSIONES

A lo largo de este proyecto, he adquirido conocimientos clave en el diseño e implementación de sistemas distribuidos basados en el modelo multi-cliente y multi-servidor. Me permitió comprender la importancia de dividir las responsabilidades en servidores especializados para mejorar el rendimiento y la escalabilidad del sistema. Además, profundicé en el manejo de concurrencia, aprendiendo a aplicar técnicas de sincronización como la exclusión mutua mediante synchronized y la coordinación de accesos mediante wait y notifyAll en entornos de múltiples clientes y servidores.

Uno de los aspectos más relevantes que experimenté fue la integración de diferentes componentes del sistema, desde la autenticación de usuarios hasta la ejecución segura de transacciones bancarias. La implementación de un Servidor de Autenticación me permitió garantizar que solo los clientes registrados pudieran interactuar con el sistema, mientras que el Servidor de Saldo y el Servidor de Transacciones aseguraron la correcta administración de los fondos y las operaciones financieras. El Servidor de Sincronización resultó ser una pieza fundamental para evitar condiciones de carrera y garantizar la integridad de los datos en escenarios de concurrencia elevada.

El uso de sockets en Java para la comunicación entre clientes y servidores me permitió afianzar mi conocimiento sobre protocolos de red y el manejo de flujos de datos en aplicaciones distribuidas. También reforcé mi capacidad de diseñar sistemas tolerantes a fallos, implementando estrategias para manejar excepciones y garantizar la continuidad del servicio incluso ante posibles errores de conexión.

En conclusión, este trabajo no solo me permitió comprender los principios teóricos de los sistemas distribuidos, sino que también me brindó experiencia práctica en la construcción de un sistema bancario seguro, eficiente y escalable. La aplicación de conceptos como el control de concurrencia, la comunicación en red y la gestión de múltiples servidores ha sido fundamental para mi desarrollo en el diseño de arquitecturas distribuidas. Estos conocimientos no solo son aplicables en el ámbito bancario, sino que pueden extenderse a otros dominios tecnológicos y proyectos de software en el futuro.