



# INSTITUTO POLITECNICO NACIONAL

## ESCUELA SUPERIOR DE COMPUTO

Asignatura:

Sistemas Distribuido

Profesor:

Carreto Arellano Chadwick

Alumno:

Cruz Cubas Ricardo

Grupo:

7CM1

Practica 4:

Múltiples Clientes – Múltiples Servidores

## Contenido

Introducción.....	2
Planteamiento del Problema .....	3
Propuesta de Solución .....	4
Materiales y Métodos Utilizados .....	5
Desarrollo de la Solución .....	6
Resultados .....	8
Conclusión.....	8

# Introducción

En una arquitectura de múltiples clientes y múltiples servidores, varios clientes pueden conectarse simultáneamente a diferentes servidores para intercambiar información de manera eficiente. Este modelo se usa en aplicaciones distribuidas, tales como servicios en la nube, sistemas de mensajería en tiempo real, plataformas de juegos en línea y procesamiento paralelo de datos. La principal ventaja de esta arquitectura es su capacidad para distribuir la carga entre múltiples servidores, mejorando la escalabilidad y el rendimiento del sistema.

Cada servidor actúa como un nodo independiente que puede manejar múltiples clientes de manera concurrente. Para lograrlo, los servidores utilizan un modelo basado en hilos, donde cada cliente recibe un hilo exclusivo que le permite enviar y recibir datos sin interferir con otras conexiones. Esto significa que un servidor puede atender múltiples clientes al mismo tiempo, evitando bloqueos y asegurando una respuesta fluida.

Los clientes pueden conectarse dinámicamente a distintos servidores, ya sea de forma manual o mediante un balanceador de carga, optimizando el uso de los recursos del sistema. Este enfoque es especialmente útil en entornos donde la demanda de usuarios varía constantemente, permitiendo agregar o eliminar servidores según sea necesario para mantener el rendimiento óptimo.

Para garantizar una comunicación sincronizada y evitar problemas de acceso a recursos compartidos, se emplean diferentes mecanismos de sincronización en la gestión de hilos. Entre las técnicas más utilizadas se encuentran:

- **Bloques synchronized:** Permiten restringir el acceso a secciones críticas del código, asegurando que solo un hilo pueda ejecutarlas a la vez.
- **ReentrantLock:** Proporciona un control más flexible sobre la sincronización, permitiendo bloqueos explícitos y condiciones avanzadas de concurrencia.
- **BlockingQueue:** Facilita la comunicación segura entre hilos al gestionar colas de mensajes o tareas sin riesgos de condiciones de carrera.
- **Semáforos y Monitores:** Limitan el acceso simultáneo a ciertos recursos, controlando el número de hilos que pueden interactuar con ellos al mismo tiempo.

El uso de estos métodos asegura que múltiples clientes puedan interactuar con los servidores sin generar conflictos ni afectar el rendimiento del sistema. Esta arquitectura es ampliamente utilizada en sistemas de alta disponibilidad, garantizando que las aplicaciones puedan soportar grandes volúmenes de usuarios sin comprometer la estabilidad ni la velocidad de respuesta.

En escenarios más avanzados, se pueden implementar técnicas como balanceo de carga, replicación de servidores y sistemas de mensajería distribuida para mejorar aún más la eficiencia del sistema. Gracias a estas estrategias, los sistemas con múltiples clientes y múltiples servidores pueden ofrecer una experiencia fluida y confiable, adaptándose a las necesidades cambiantes de los usuarios y optimizando el uso de los recursos disponibles.

## Planteamiento del Problema

En la actualidad, muchas aplicaciones y servicios requieren la capacidad de manejar múltiples clientes de manera simultánea sin afectar el rendimiento ni la estabilidad del sistema. En entornos distribuidos, como servicios en la nube, sistemas de mensajería en tiempo real y procesamiento de datos, la demanda de usuarios puede aumentar de forma impredecible, lo que hace necesario implementar una arquitectura eficiente que permita distribuir la carga de trabajo entre varios servidores.

El problema surge cuando un único servidor debe gestionar numerosas conexiones simultáneamente, lo que puede generar congestión, tiempos de respuesta elevados y fallos en la comunicación. A medida que el número de clientes crece, un solo servidor se vuelve un cuello de botella, afectando la experiencia del usuario y limitando la escalabilidad del sistema.

Para abordar esta problemática, es fundamental diseñar una arquitectura de múltiples clientes y múltiples servidores, en la cual varias instancias de servidores puedan operar en paralelo, distribuyendo las solicitudes de los clientes de manera equitativa. Sin embargo, este modelo introduce nuevos desafíos relacionados con la sincronización de conexiones, la gestión eficiente de los recursos y la comunicación entre los distintos nodos del sistema.

Uno de los principales retos es garantizar que cada cliente pueda conectarse a un servidor disponible sin generar conflictos en la gestión de datos compartidos. La falta de un mecanismo adecuado de sincronización puede llevar a problemas como condiciones de carrera, accesos simultáneos a recursos críticos y pérdida de datos. Además, se requiere una estrategia eficiente para permitir que los clientes seleccionen el servidor más adecuado y aseguren una conexión estable y confiable.

En este contexto, es necesario implementar un modelo basado en programación concurrente que permita gestionar múltiples conexiones de manera eficiente. Utilizando técnicas de sincronización como bloques `synchronized`, `ReentrantLock` y colas concurrentes, se puede garantizar que los clientes interactúen con los servidores sin generar problemas de concurrencia. Además, se pueden emplear algoritmos de balanceo de carga para distribuir las solicitudes de manera uniforme entre los servidores disponibles, optimizando el uso de los recursos del sistema.

Por lo tanto, el desarrollo de una arquitectura de múltiples clientes y múltiples servidores representa una solución efectiva para mejorar la escalabilidad, la eficiencia y la estabilidad de sistemas que manejan grandes volúmenes de tráfico. Implementar correctamente esta solución permitirá evitar problemas de sobrecarga, reducir los tiempos de respuesta y garantizar una comunicación fluida entre los clientes y los servidores.

# Propuesta de Solución

Para abordar la problemática de la sobrecarga en servidores y la gestión eficiente de múltiples clientes, se propone implementar una arquitectura de múltiples clientes y múltiples servidores basada en programación concurrente en Java. Esta solución permitirá distribuir la carga de trabajo entre varios servidores, optimizando el rendimiento del sistema y garantizando una comunicación estable y fluida entre los clientes y los servidores.

## Implementación de Servidores Concurrentes

Cada servidor estará diseñado para manejar múltiples clientes simultáneamente mediante el uso de hilos (threads). Al recibir una conexión entrante, el servidor creará un nuevo hilo para gestionar la comunicación con ese cliente sin bloquear la ejecución de otros procesos. De esta manera, cada servidor podrá atender múltiples solicitudes de manera eficiente.

Para garantizar la sincronización y evitar problemas de concurrencia, se emplearán técnicas como:

- Bloques synchronized para restringir el acceso a secciones críticas del código.
- ReentrantLock para gestionar bloqueos de manera más flexible.
- BlockingQueue para la gestión de mensajes entre clientes y servidores sin riesgo de condiciones de carrera.

## Distribución de Carga entre Servidores

Para evitar la sobrecarga de un solo servidor, los clientes podrán conectarse a diferentes servidores de manera dinámica. Se pueden utilizar diferentes estrategias para distribuir las conexiones, tales como:

- Selección manual: El cliente elige el servidor al cual conectarse.
- Balanceador de carga: Se implementa un servicio central que redirige a los clientes hacia el servidor con menor carga.
- Round-robin: Cada nuevo cliente es asignado secuencialmente a un servidor distinto.

## Comunicación Cliente-Servidor

Los clientes se conectarán a los servidores utilizando sockets TCP/IP, garantizando una comunicación confiable. Cada cliente enviará solicitudes al servidor, el cual procesará los datos y devolverá una respuesta en tiempo real. Este modelo es ideal para aplicaciones como sistemas de mensajería, procesamiento de datos distribuidos y servicios en la nube.

## Gestión de Conexiones y Sincronización

Para evitar conflictos en el acceso a recursos compartidos, se implementarán mecanismos de sincronización que garanticen la integridad de los datos. Se utilizarán:

- Semáforos y monitores para limitar el acceso simultáneo a ciertos recursos.
- Colas de mensajes concurrentes para procesar múltiples solicitudes sin interferencias.

- Manejo de excepciones y reconexión automática para mejorar la estabilidad del sistema en caso de fallos de conexión.

#### Escalabilidad y Expansión del Sistema

El diseño modular de la arquitectura permitirá agregar más servidores sin necesidad de modificar la estructura del sistema. Además, se podrán incorporar nuevas funcionalidades, como autenticación de usuarios, cifrado de datos y monitoreo del rendimiento, para mejorar la seguridad y la eficiencia de la aplicación.

La implementación de una arquitectura de múltiples clientes y múltiples servidores en Java proporcionará un sistema escalable, eficiente y robusto. Mediante el uso de técnicas de programación concurrente y balanceo de carga, se garantizará una distribución equitativa del trabajo, evitando la sobrecarga de los servidores y asegurando tiempos de respuesta óptimos para los clientes.

## Materiales y Métodos Utilizados

- Java Development Kit (JDK 8 o superior) para la programación en Java.
- IDE de desarrollo (Eclipse, IntelliJ IDEA o NetBeans) para la escritura y depuración del código.
- Bibliotecas estándar de Java (java.net y java.io) para el manejo de sockets y flujo de datos.

Para el desarrollo de la practica múltiples clientes y múltiples servidores se utilizarán tanto hardware como software adecuados para garantizar una comunicación eficiente. Se requiere una computadora o servidor con capacidad para manejar múltiples procesos concurrentes y una red de comunicación estable, ya sea LAN o Internet. En cuanto al software, se utilizará Java Development Kit (JDK 8 o superior) junto con un entorno de desarrollo como Eclipse, IntelliJ IDEA o NetBeans. Además, se emplearán bibliotecas estándar de Java como java.net y java.io para el manejo de sockets y flujo de datos en un sistema operativo compatible como Windows.

El programa se basa en una arquitectura cliente-servidor utilizando sockets TCP/IP para permitir la interacción en tiempo real. Cada servidor gestionará múltiples conexiones a través de hilos independientes, garantizando la concurrencia y evitando bloqueos en la comunicación. Para esto se emplearán mecanismos de sincronización como synchronized, ReentrantLock y BlockingQueue.

Los clientes enviarán mensajes al servidor, el cual procesará la información y la retransmitirá a todos los clientes conectados. Para manejar la comunicación se utilizarán flujos de entrada y salida mediante InputStream y OutputStream en Java. También se implementará un sistema de gestión de conexiones que permitirá manejar la desconexión de clientes sin afectar la estabilidad del servidor.

# Desarrollo de la Solución

La practica tiene tres partes principales:

- El servidor: Es como un operador de llamadas que recibe los mensajes de los clientes y los reenvía a todos los demás.
- Los clientes: Son las personas que llaman al operador para hablar con otras personas.
- Los mensajes: Son las palabras que los clientes envían y que el operador retransmite a los demás.

Este es el codigo que se encarga de recibir los mensajes de los clientes y enviarlos a todos los demás.

```
public class Servidor {  
    private int puerto;  
  
    public Servidor(int puerto) {  
        this.puerto = puerto;  
    }  
  
    public void iniciar() {  
        try (ServerSocket serverSocket = new ServerSocket(puerto)) {  
            System.out.println("Servidor en el puerto " + puerto + " esperando conexiones...");  
            while (true) {  
                Socket cliente = serverSocket.accept();  
                System.out.println("Cliente conectado en el servidor " + puerto);  
                new ManejadorCliente(cliente).start();  
            }  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

El servidor es como una oficina de un operador de llamadas. Primero, crea una lista para almacenar a todos los clientes conectados. Luego, abre una "línea telefónica" (llamada ServerSocket) en el puerto 12345 y espera a que alguien llame. Cuando un cliente se conecta, el servidor le asigna un nuevo operador (ManejadorCliente) y empieza a escuchar su conversación.

```
private static class ManejadorCliente extends Thread {  
    private Socket socket;  
  
    public ManejadorCliente(Socket socket) {  
        this.socket = socket;  
    }  
  
    public void run() {  
        try (BufferedReader entrada = new BufferedReader(new InputStreamReader(socket.getInputStream()));  
            PrintWriter salida = new PrintWriter(socket.getOutputStream(), true)) {  
  
            String mensaje;  
            while ((mensaje = entrada.readLine()) != null) {  
                System.out.println("Recibido: " + mensaje);  
                salida.println("Echo: " + mensaje);  
            }  
        } catch (IOException e) {  
            e.printStackTrace();  
        } finally {  
            try {  
                socket.close();  
            } catch (IOException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

Cada vez que un cliente se conecta, el servidor crea un operador específico para él (ManejadorCliente). Este operador se encargará de escuchar lo que dice el cliente y transmitirlo a los demás.

Recibe los mensajes: El operador espera a que el cliente escriba algo.

Guarda la conexión: Se añade a la lista de clientes conectados.

Envía los mensajes a todos: Cuando el cliente dice algo, el servidor reenvía el mensaje a todos los demás clientes.

Elimina al cliente si se desconecta: Si el cliente cierra la aplicación, el servidor lo elimina de la lista.

```
public class Cliente {
    private String servidorIP;
    private int puerto;

    public Cliente(String servidorIP, int puerto) {
        this.servidorIP = servidorIP;
        this.puerto = puerto;
    }

    public void iniciar() {
        try (Socket socket = new Socket(servidorIP, puerto);
            BufferedReader entrada = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter salida = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader teclado = new BufferedReader(new InputStreamReader(System.in))) {

            System.out.println("Conectado al servidor en " + servidorIP + ":" + puerto);
            String mensaje;
            while (true) {
                System.out.print("Mensaje: ");
                mensaje = teclado.readLine();
                if ("salir".equalsIgnoreCase(mensaje)) break;
                salida.println(mensaje);
                System.out.println("Servidor responde: " + entrada.readLine());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        String servidorIP = args[0];
        int puerto = Integer.parseInt(args[1]);
        new Cliente(servidorIP, puerto).iniciar();
    }
}
```

El cliente es como una persona que llama al operador para unirse al chat.

Se conecta al servidor: Abre una conexión con el operador en el puerto 12345.

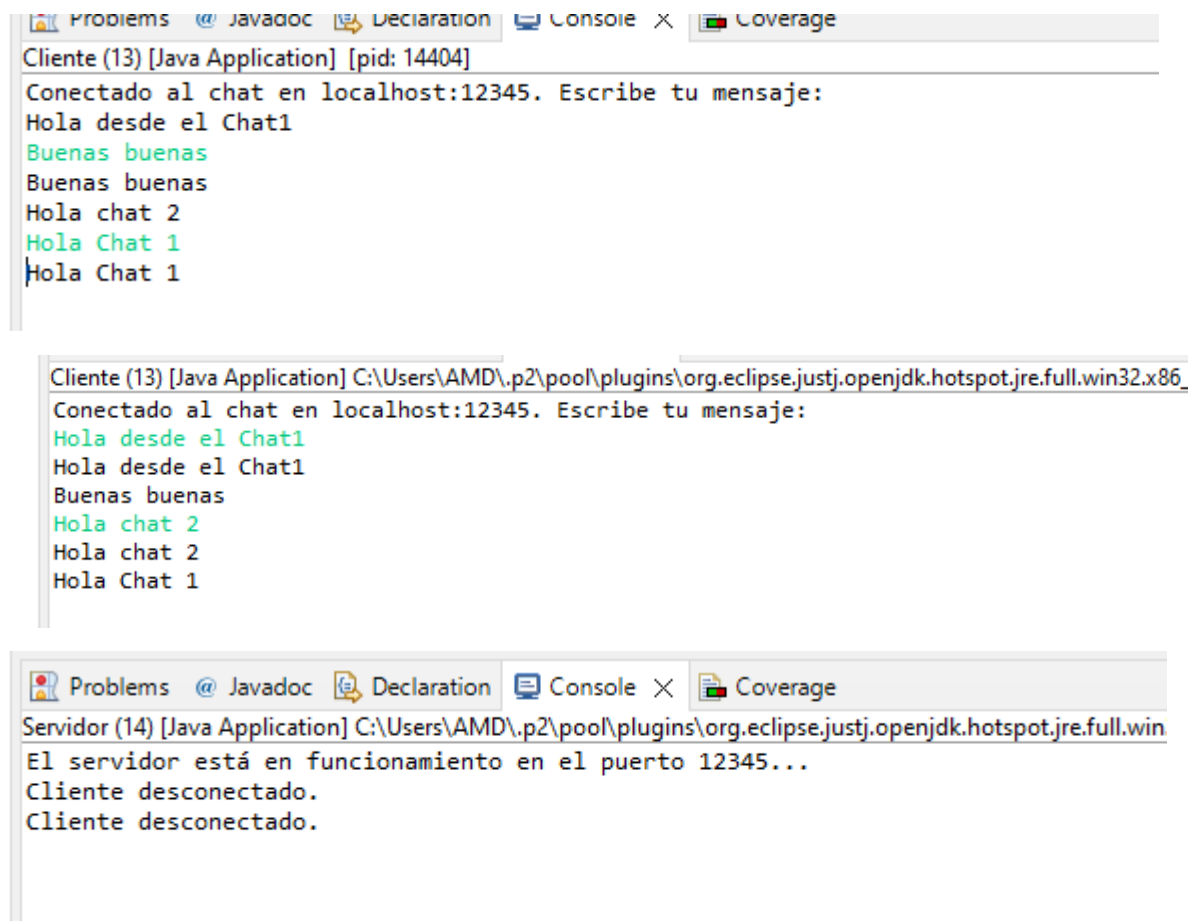
Escucha mensajes: Hay un "receptor de radio" que siempre está escuchando lo que dicen los demás.

Envía mensajes: Cuando el usuario escribe algo, se lo manda al operador, quien lo reenvía a todos.

Muestra los mensajes en la pantalla: Todo lo que dicen los demás clientes aparece en la consola del usuario.



## Resultados



The image shows two screenshots of the Eclipse IDE console. The top screenshot shows the output for 'Cliente (13) [Java Application] [pid: 14404]'. The bottom screenshot shows the output for 'Servidor (14) [Java Application] C:\Users\AMD\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86\_64.jre\bin\java.exe'.

```
Problems  @ Javadoc  Declaration  Console  X  Coverage
Cliente (13) [Java Application] [pid: 14404]
Conectado al chat en localhost:12345. Escribe tu mensaje:
Hola desde el Chat1
Buenas buenas
Buenas buenas
Hola chat 2
Hola Chat 1
Hola Chat 1

Cliente (13) [Java Application] C:\Users\AMD\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64.jre\bin\java.exe
Conectado al chat en localhost:12345. Escribe tu mensaje:
Hola desde el Chat1
Hola desde el Chat1
Buenas buenas
Hola chat 2
Hola chat 2
Hola Chat 1

Problems  @ Javadoc  Declaration  Console  X  Coverage
Servidor (14) [Java Application] C:\Users\AMD\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64.jre\bin\java.exe
El servidor está en funcionamiento en el puerto 12345...
Cliente desconectado.
Cliente desconectado.
```

## Conclusión

En conclusión, hemos construido un sistema de chat con múltiples clientes y servidores en Java, utilizando sockets TCP/IP para permitir la comunicación entre varios usuarios en tiempo real. Este sistema se compone de dos partes principales: el servidor y los clientes. El servidor actúa como un intermediario, recibiendo los mensajes de los clientes y retransmitiéndolos a todos los demás, mientras que cada cliente envía y recibe mensajes de manera independiente.

A lo largo del desarrollo, hemos abordado importantes aspectos como la programación concurrente mediante el uso de hilos para manejar múltiples clientes, asegurando que cada uno pueda comunicarse sin interferencias. También implementamos medidas para evitar errores comunes, como el `ArrayIndexOutOfBoundsException`, y facilitamos la configuración flexible del puerto y la dirección del servidor, lo que permite ejecutar el sistema en diferentes entornos.

Este sistema de chat no solo muestra cómo manejar la concurrencia en Java con hilos, sino que también pone en práctica conceptos clave como el uso de sockets para la comunicación

en red, sincronización de recursos compartidos (como la lista de clientes) y manejo de excepciones para mejorar la estabilidad del sistema.

En general, este enfoque es escalable, flexible y robusto, permitiendo futuras mejoras como la implementación de múltiples servidores para mayor disponibilidad y rendimiento, o incluso la integración de nuevas funcionalidades como salas de chat privadas o la gestión de usuarios. Este tipo de arquitectura es fundamental para aplicaciones de mensajería en tiempo real, videojuegos en línea y otros sistemas distribuidos.