



INSTITUTO POLITECNICO NACIONAL

ESCUELA SUPERIOR DE COMPUTO



Asignatura:
Sistemas Distribuidos.

Profesor:
Carreto Arellano Chadwick

Grupo:
7CM1.

Alumno:
Cruz Cubas Ricardo

Practica 3:
Múltiples Usuarios – Servidor.

Contenido

Antecedentes	2
Planteamiento del Problema	4
Propuesta de solución	4
Materiales y Métodos Empleados	4
Desarrollo	5

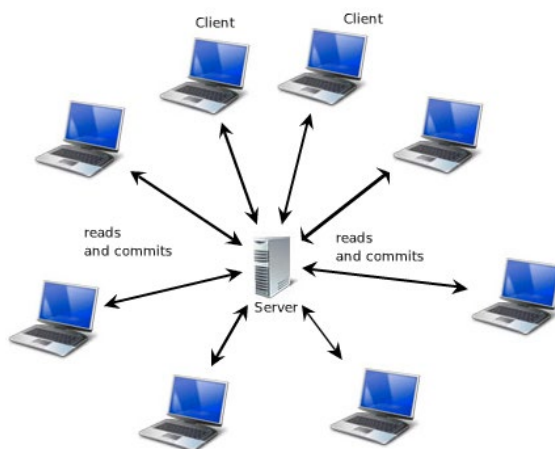
Antecedentes

El modelo de comunicación multcliente-servidor ha sido fundamental en el desarrollo de aplicaciones y servicios en redes de computadoras. En este modelo, un servidor centralizado proporciona recursos o servicios, y varios clientes se conectan a él para solicitar estos recursos. Este enfoque se ha vuelto clave en la arquitectura de Internet, con servidores web, bases de datos y otros servicios distribuidos funcionando bajo este esquema.

Históricamente, las redes de computadoras empezaron a funcionar con modelos de cliente-servidor simples, pero con el tiempo, la necesidad de manejar múltiples clientes simultáneamente llevó a la evolución de sistemas más complejos. El protocolo TCP/IP (Transmission Control Protocol/Internet Protocol), utilizado para la transmisión de datos en redes, ha sido esencial para que este modelo funcione eficientemente en diversas aplicaciones.

Los primeros sistemas multicliente-servidor surgieron en ambientes locales, como redes LAN (Local Area Network), pero con el auge de Internet en los años 90 y 2000, este modelo se expandió a nivel global, permitiendo que servidores hospedaran miles o incluso millones de conexiones simultáneas. A medida que las capacidades de los servidores y las redes aumentaron, la necesidad de manejar estas conexiones de manera eficiente y escalable se convirtió en una prioridad.

El modelo "multicliente - un servidor" es uno de los pilares fundamentales en el diseño de arquitecturas distribuidas modernas. En este enfoque, un único servidor centralizado interactúa con múltiples clientes que pueden estar ubicados en diferentes ubicaciones físicas. Estos clientes pueden ser dispositivos, aplicaciones o incluso otros servidores que requieren



acceso a los servicios proporcionados por el servidor central. El servidor, a su vez, gestiona todas las solicitudes que los clientes realizan, ofreciendo recursos, datos o servicios específicos según las necesidades de cada uno.

Este tipo de arquitectura se ha convertido en la base de muchas tecnologías que usamos a diario. Las aplicaciones web, servicios en la nube, bases de datos distribuidas y sistemas de mensajería instantánea son solo algunos ejemplos donde se emplea el modelo

multicliente-servidor. En este contexto, el servidor se encarga no solo de procesar y responder a las solicitudes de los clientes, sino también de coordinar tareas, administrar el acceso a datos y garantizar que todos los clientes reciban la información correcta en el momento adecuado.

La principal ventaja de este modelo es su capacidad para centralizar la administración de recursos y servicios. Esto significa que los recursos pueden ser gestionados de forma más eficiente, y la actualización o el mantenimiento del sistema puede realizarse en el servidor sin

necesidad de modificar el entorno de cada cliente. Además, el modelo permite la escalabilidad, es decir, la capacidad de manejar un número creciente de clientes sin afectar el rendimiento del sistema.

Sin embargo, también existen desafíos asociados con este enfoque. Uno de los principales problemas es la gestión de múltiples conexiones simultáneas. A medida que el número de clientes crece, el servidor debe ser capaz de manejar una cantidad significativa de solicitudes concurrentes sin comprometer la eficiencia o estabilidad. Para lograr esto, es necesario implementar técnicas de optimización como la multiplexación, el balanceo de carga y la gestión de hilos o procesos.

En términos de redes, los servidores deben tener una infraestructura robusta que les permita comunicarse eficientemente con los clientes a través de redes locales o incluso a través de Internet. Los protocolos de comunicación como TCP/IP, HTTP, WebSockets y otros, son fundamentales para garantizar la correcta transmisión de datos entre el servidor y los clientes.

Otro aspecto importante de este modelo es la seguridad. Dado que el servidor centraliza el acceso a los recursos, cualquier vulnerabilidad en el servidor podría poner en riesgo la seguridad de todos los clientes. Por lo tanto, es esencial que los servidores implementen mecanismos de autenticación, autorización y cifrado para proteger tanto la integridad de los datos como la privacidad de los usuarios.

El modelo "multicliente - un servidor" es esencial en el mundo moderno de las tecnologías de la información. Su capacidad para permitir la interacción eficiente entre numerosos clientes y un único servidor ha sido clave para el desarrollo de la infraestructura que sustenta servicios en línea, aplicaciones móviles, y sistemas distribuidos. A lo largo de esta introducción al tema, se explorarán con mayor detalle los principios fundamentales de este modelo, sus aplicaciones más comunes, y los desafíos técnicos que enfrenta al implementarse en sistemas reales.

Planteamiento del Problema

En la actualidad, muchas aplicaciones y sistemas informáticos requieren la comunicación entre múltiples clientes y un único servidor para el intercambio de datos y servicios. Sin embargo, uno de los principales desafíos en estos entornos es la gestión eficiente de múltiples conexiones simultáneas sin afectar el rendimiento del sistema.

En esta práctica, se busca desarrollar un sistema cliente-servidor en el que un único servidor sea capaz de atender múltiples clientes al mismo tiempo, permitiendo la comunicación y el intercambio de información de manera fluida. El problema radica en garantizar que el servidor pueda manejar múltiples solicitudes sin bloquearse, colapsar o generar tiempos de respuesta elevados.

Para ello, se deben considerar aspectos como el uso de hilos (threads) o procesos para gestionar cada conexión de manera independiente, la correcta sincronización de los datos compartidos y la optimización del uso de los recursos del servidor. Además, es importante implementar mecanismos que permitan la estabilidad del sistema frente a fallos inesperados o desconexiones de clientes.

Esta práctica permitirá comprender los principios básicos del modelo multicliente-servidor, así como aplicar conceptos clave en la programación de redes y la gestión de concurrencia.

Propuesta de solución

Descripción del problema:

Se necesita implementar un servidor de chat que permita la comunicación entre múltiples clientes de manera concurrente. Cada cliente podrá enviar mensajes al servidor, y este los retransmitirá a todos los clientes conectados en tiempo real. Además, debe haber un mecanismo para que los clientes puedan salir del chat sin afectar la comunicación entre los demás usuarios.

Objetivo de la solución:

Desarrollar un sistema basado en sockets en Java que permita a múltiples clientes conectarse a un servidor de chat, enviar y recibir mensajes en tiempo real. La solución debe garantizar la gestión eficiente de las conexiones concurrentes, evitar bloqueos o fallos inesperados y proporcionar una experiencia fluida a los usuarios.

Materiales y Métodos Empleados

Materiales:

Entorno de Desarrollo:

- Eclipse IDE (versión recomendada: Eclipse IDE for Java Developers).
- JDK 8 o superior instalado en el sistema.

Lenguaje de Programación:

- Java (versión 8 o superior), debido a su capacidad para manejar programación en red y concurrencia de manera eficiente.

Librerías Utilizadas:

- java.net → Para la creación de sockets y la comunicación en red.
- java.io → Para la lectura y escritura de datos entre el cliente y el servidor.
- java.util.concurrent → Para la gestión de múltiples clientes mediante un ExecutorService.
- java.util.concurrent.CopyOnWriteArrayList → Para almacenar conexiones activas de clientes y garantizar la seguridad en entornos multihilo.

Desarrollo

Cliente

El cliente del chat es básicamente el programa que usamos para conectarnos con otras personas a través del servidor. Imaginemos que es como abrir una aplicación de mensajería, solo que en este caso se comunica directamente con el servidor en lugar de depender de una plataforma como WhatsApp o Telegram.

```
public class ClienteChat {  
    private static final String DIRECCION_SERVIDOR = "localhost";  
    private static final int PUERTO_SERVIDOR = 12345;  
  
    public static void main(String[] args) {  
        try (Socket socket = new Socket(DIRECCION_SERVIDOR, PUERTO_SERVIDOR));
```

Esta parte establece la conexión con el servidor. Este bloque crea un Socket, que es como un canal de comunicación entre el cliente y el servidor. Usamos la dirección "localhost" (que indica que el servidor está en la misma computadora) y el puerto 12345, que es el canal específico que el servidor está escuchando. Si el servidor está en otro lugar, en lugar de "localhost", se pondría la dirección IP del servidor. Si la conexión es exitosa, el cliente se conecta; si no, el programa lanzará un error.

```
        BufferedReader entrada = new BufferedReader(new InputStreamReader(socket.getInputStream()));  
        PrintWriter salida = new PrintWriter(socket.getOutputStream(), true);  
        BufferedReader entradaUsuario = new BufferedReader(new InputStreamReader(System.in)) {  
  
        System.out.println("Conectado al servidor de chat. Escribe tu mensaje:");
```

Se configuran tres objetos para manejar la entrada y salida de datos. El primero, entrada, se usa para leer los mensajes que el servidor envía al cliente. salida es el objeto que se utiliza para enviar mensajes del cliente al servidor. Y finalmente, entradaUsuario lee lo que el usuario escribe en la consola, es decir, los mensajes que va a enviar al servidor.

```

Thread hiloRecepcion = new Thread(() -> {
    try {
        String mensajeServidor;
        while ((mensajeServidor = entrada.readLine()) != null) {
            System.out.println(mensajeServidor);
        }
    } catch (IOException e) {
        System.out.println("Conexión cerrada.");
    }
});
hiloRecepcion.start();

```

El siguiente bloque de código es muy importante porque crea un hilo en segundo plano que está constantemente escuchando los mensajes del servidor. Usamos un hilo porque de esta manera el programa puede seguir funcionando mientras espera recibir mensajes. Dentro de este hilo, el programa está leyendo constantemente los mensajes del servidor usando `entrada.readLine()` y los imprime en la pantalla cada vez que recibe algo. Si en algún momento la conexión se cierra o el servidor deja de enviar mensajes, se captura una excepción y el programa muestra "Conexión cerrada" para informar al usuario.

```

String mensajeUsuario;
while ((mensajeUsuario = entradaUsuario.readLine()) != null) {
    salida.println(mensajeUsuario);
    if (mensajeUsuario.equalsIgnoreCase("salir")) {
        break;
    }
}

```

Después de configurar el hilo para recibir mensajes, el programa entra en un ciclo donde espera que el usuario escriba algo en la consola. Lo que el usuario escribe se manda al servidor mediante `salida.println(mensajeUsuario)`. Si el mensaje es "salir", el ciclo se interrumpe y el programa termina la conexión. Esto permite que el usuario envíe mensajes mientras el chat está activo.

```

    }
} catch (IOException e) {
    e.printStackTrace();
}

```

Por último, si ocurre algún error, como que el servidor se apague o haya problemas con la red, el programa captura cualquier excepción de entrada y salida con el bloque `catch` y muestra el error en la consola

Servidor.

El servidor comienza escuchando en el puerto 12345 para aceptar conexiones entrantes. Esto se realiza mediante el uso de un `ServerSocket`, que se encarga de esperar a que los clientes se conecten. El servidor se queda en un bucle esperando que alguien se conecte. Cuando un cliente intenta conectarse, el servidor acepta la conexión mediante el método `accept()`, lo que devuelve un `Socket` que representa esa conexión con el cliente.

```
public static void main(String[] args) {
    System.out.println("Servidor de chat iniciado en el puerto " + PORT);
    try (ServerSocket serverSocket = new ServerSocket(PORT)) {
        while (true) {
            Socket clientSocket = serverSocket.accept();
            System.out.println("Nuevo cliente conectado: " + clientSocket.getInetAddress());
            pool.execute(new ClientHandler(clientSocket));
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Una vez que el servidor ha aceptado una conexión, se crea un nuevo hilo (`Thread`) para manejar la comunicación con ese cliente. En lugar de atender a cada cliente en el hilo principal (lo que haría que el servidor se bloqueara esperando a que uno de los clientes termine de interactuar), se utiliza un `ExecutorService` que maneja un pool de hilos para gestionar múltiples conexiones de manera eficiente. Esto asegura que el servidor pueda manejar a varios clientes al mismo tiempo sin problemas de bloqueo o sobrecarga.

```
Socket clientSocket = serverSocket.accept();
System.out.println("Nuevo cliente conectado: " + clientSocket.getInetAddress());
pool.execute(new ClientHandler(clientSocket));
```

Dentro del hilo, el servidor configura dos flujos de datos: uno para recibir datos del cliente (`BufferedReader in`) y otro para enviar datos de vuelta al cliente (`PrintWriter out`). Estos flujos permiten que el servidor lea los mensajes del cliente y también envíe respuestas.

```
try (BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
    PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true)) {
```

El servidor también mantiene una lista de todos los clientes conectados. Usando un `CopyOnWriteArrayList`, se guarda un objeto `PrintWriter` para cada cliente. Este tipo de lista es útil en entornos multihilo porque permite agregar y quitar elementos sin problemas, lo que es esencial para gestionar las conexiones de manera concurrente.

```
private Socket clientSocket;
private static CopyOnWriteArrayList<PrintWriter> clients = new CopyOnWriteArrayList<>();

public ClientHandler(Socket socket) {
    this.clientSocket = socket;
}
```


Cuando un cliente envía un mensaje, el servidor lo recibe, lo imprime en la consola y luego lo retransmite a todos los demás clientes conectados. Esto se hace mediante el método `broadcast()`, que recorre todos los `PrintWriter` de los clientes y les envía el mensaje. Así, todos los usuarios conectados pueden ver el mensaje que un cliente ha enviado.

El servidor también permite que los clientes se desconecten de manera ordenada. Si un cliente envía el mensaje "salir", el servidor le informa que se está desconectando, cierra su socket y elimina su `PrintWriter` de la lista de clientes. Si ocurre algún problema durante la comunicación (como que un cliente se desconecte inesperadamente), se maneja mediante excepciones, y el servidor simplemente cierra la conexión de ese cliente y lo elimina de la lista de clientes activos.

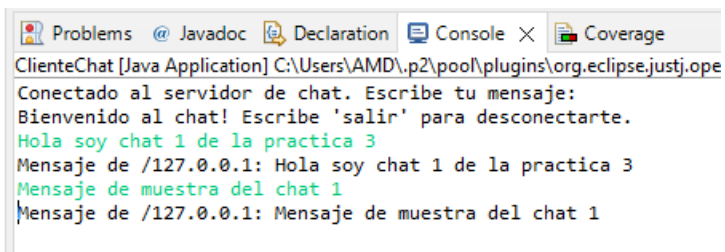
```
clients.add(out);
out.println("Bienvenido al chat! Escribe 'salir' para desconectarte.");
String message;
while ((message = in.readLine()) != null) {
    if (message.equalsIgnoreCase("salir")) {
        out.println("Desconectando...");
        break;
    }
    System.out.println("Mensaje recibido: " + message);
    broadcast("Mensaje de " + clientSocket.getInetAddress() + ": " + message);
}
```

Finalmente, el servidor sigue en ejecución esperando más conexiones de nuevos clientes. Este proceso de escuchar nuevas conexiones y manejar a los clientes concurrentemente se repite indefinidamente hasta que el servidor se detenga.

```
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        clientSocket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    clients.removeIf(out -> out.checkError());
    System.out.println("Cliente desconectado");
}
```

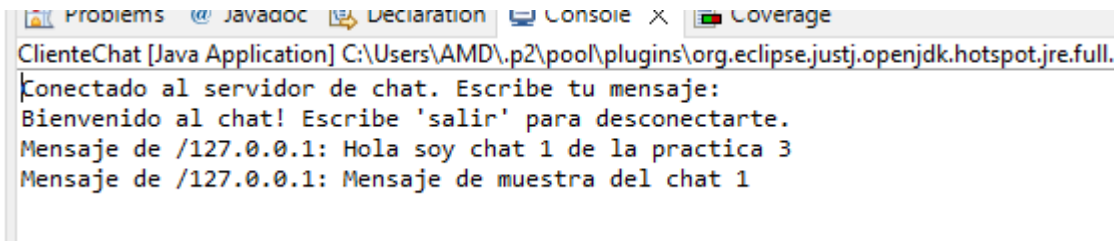
Resultados.

Chat 1



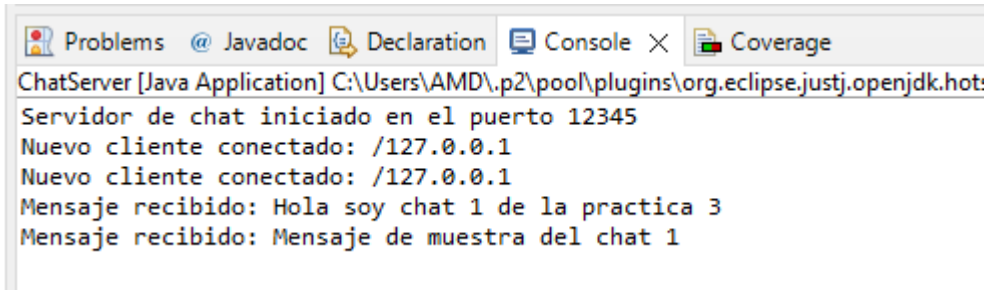
```
Problems @ Javadoc Declaration Console X Coverage
ClienteChat [Java Application] C:\Users\AMD\.p2\pool\plugins\org.eclipse.justj.ope
Conectado al servidor de chat. Escribe tu mensaje:
Bienvenido al chat! Escribe 'salir' para desconectarte.
Hola soy chat 1 de la practica 3
Mensaje de /127.0.0.1: Hola soy chat 1 de la practica 3
Mensaje de muestra del chat 1
Mensaje de /127.0.0.1: Mensaje de muestra del chat 1
```

Chat 2



```
Problems  @ Javadoc  Declaration  Console  X  Coverage
ClienteChat [Java Application] C:\Users\AMD\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.
Conectado al servidor de chat. Escribe tu mensaje:
Bienvenido al chat! Escribe 'salir' para desconectarte.
Mensaje de /127.0.0.1: Hola soy chat 1 de la practica 3
Mensaje de /127.0.0.1: Mensaje de muestra del chat 1
```

Servidor



```
Problems  @ Javadoc  Declaration  Console  X  Coverage
ChatServer [Java Application] C:\Users\AMD\.p2\pool\plugins\org.eclipse.justj.openjdk.hot
Servidor de chat iniciado en el puerto 12345
Nuevo cliente conectado: /127.0.0.1
Nuevo cliente conectado: /127.0.0.1
Mensaje recibido: Hola soy chat 1 de la practica 3
Mensaje recibido: Mensaje de muestra del chat 1
```

Conclusión

En conclusión, el sistema de chat cliente-servidor desarrollado utiliza un enfoque robusto y eficiente para permitir la comunicación en tiempo real entre múltiples usuarios. El servidor, que escucha de manera continua las conexiones entrantes, puede manejar múltiples clientes simultáneamente sin bloquear el flujo de trabajo gracias al uso de hilos y un `ExecutorService`. Por su parte, el cliente se encarga de interactuar con el servidor de manera fluida, permitiendo al usuario enviar y recibir mensajes sin interrupciones, gracias a la implementación de un hilo para recibir mensajes en segundo plano.

El uso de estructuras como `CopyOnWriteArrayList` asegura que la gestión de las conexiones de los clientes sea segura incluso en un entorno multihilo, lo que previene errores comunes relacionados con la manipulación de listas compartidas. Además, el sistema es flexible y permite desconectar a los usuarios de manera ordenada, lo que mejora la experiencia general.

Este sistema de chat básico puede ser expandido con más funcionalidades, como autenticación de usuarios, almacenamiento de mensajes históricos o incluso chat privado entre usuarios específicos. Sin embargo, el diseño actual proporciona una base sólida y escalable para cualquier tipo de aplicación de chat en red, demostrando la eficacia de Java para manejar aplicaciones concurrentes y en red.