INSTITUTO POLITECNICO NACIONAL

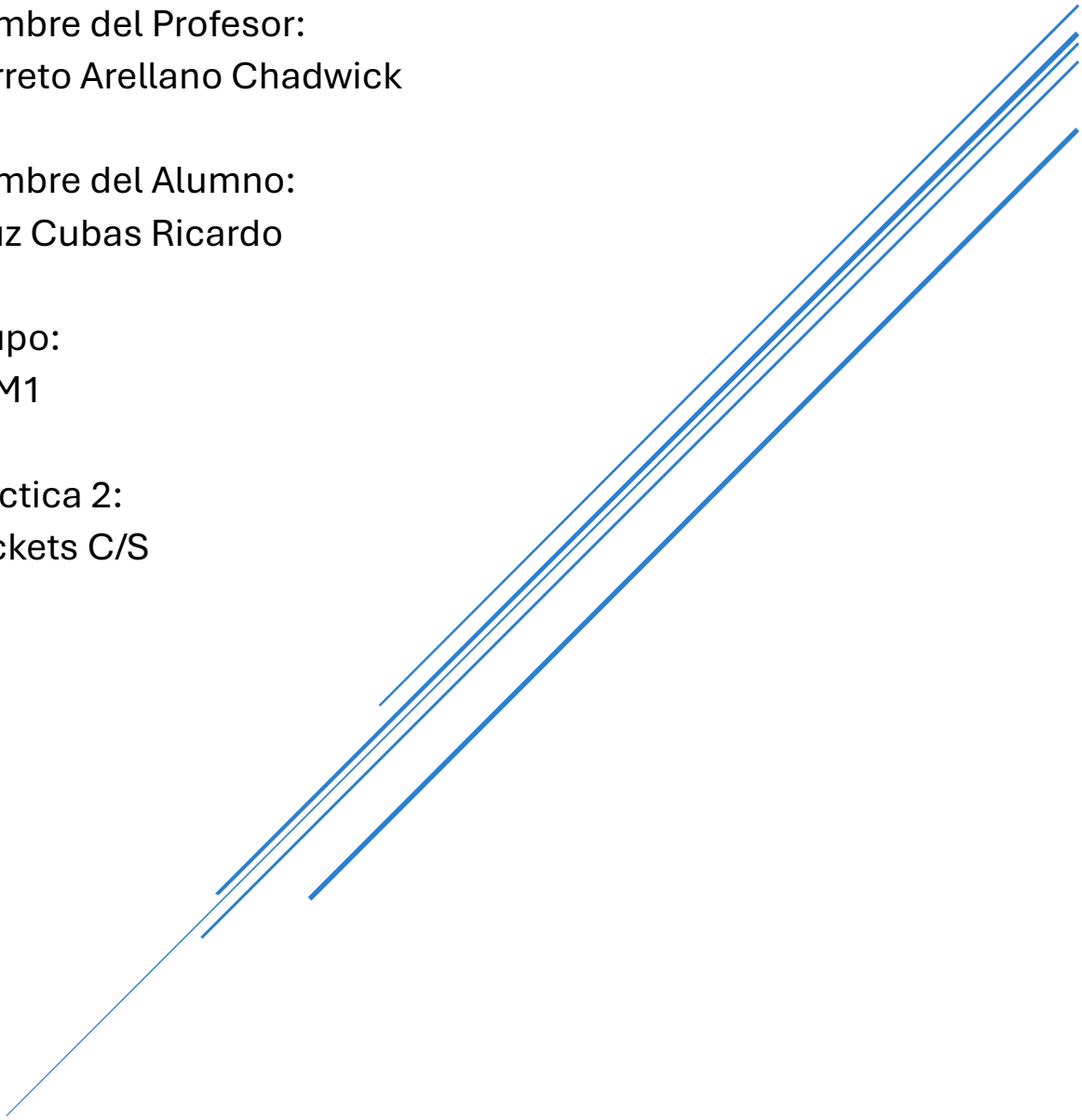
Unidad de Aprendizaje:
Sistemas Distribuidos

Nombre del Profesor:
Carreto Arellano Chadwick

Nombre del Alumno:
Cruz Cubas Ricardo

Grupo:
7CM1

Practica 2:
Sockets C/S



Antecedentes

La comunicación en redes de computadoras ha sido una necesidad desde los primeros días de la informática. Inicialmente, los sistemas informáticos eran independientes y no podían intercambiar información. Con el avance de la tecnología, se desarrollaron mecanismos para permitir la interconexión de computadoras, lo que dio lugar a la aparición de protocolos de comunicación y redes de computadoras.

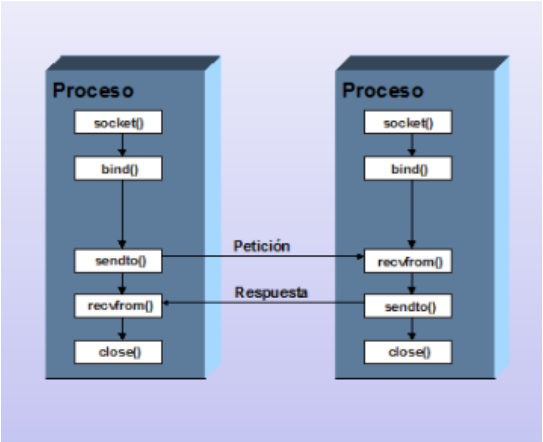
En los años 70 y 80, con el desarrollo del modelo TCP/IP (Transmission Control Protocol/Internet Protocol), se estableció una base sólida para la comunicación entre dispositivos en redes distribuidas. Como parte de este avance, surgió el concepto de sockets, introducido en Unix en 1983 como una interfaz de programación de aplicaciones (API) en la Berkeley Software Distribution (BSD). Los sockets proporcionaron una forma estándar y flexible de intercambiar datos entre procesos en diferentes dispositivos conectados a una red.

Desde entonces, los sockets han evolucionado y se han convertido en una tecnología clave en la programación de redes, siendo utilizados en una amplia gama de aplicaciones, como navegadores web, clientes de correo electrónico, sistemas de mensajería instantánea, videojuegos en línea y más.

Un socket es un mecanismo de comunicación que permite la interacción entre aplicaciones a través de una red. Se puede entender como un punto final en una conexión, donde dos dispositivos, o incluso procesos dentro de la misma máquina, pueden intercambiar datos de manera estructurada. Los sockets han sido fundamentales en el desarrollo de la comunicación en redes, permitiendo que aplicaciones como navegadores web, clientes de correo electrónico, servidores de juegos en línea y sistemas de mensajería funcionen de manera eficiente.

El concepto de sockets surgió en la década de 1980 con la introducción de la API de sockets en Berkeley Software Distribution (BSD) Unix, proporcionando un modelo estandarizado para la comunicación entre procesos en diferentes dispositivos. Antes de su aparición, las computadoras utilizaban métodos menos eficientes y específicos de cada sistema operativo para la transmisión de datos. La implementación de los sockets permitió un enfoque más universal, facilitando la interoperabilidad entre sistemas con distintos entornos y arquitecturas.

Los sockets funcionan sobre distintos protocolos de comunicación, siendo los más utilizados TCP (Transmission Control Protocol) y UDP (User Datagram Protocol). TCP es un protocolo orientado a la conexión que garantiza la entrega fiable y ordenada de los datos, lo que lo hace ideal para aplicaciones donde la precisión en la transmisión es crucial, como en la navegación web, correos electrónicos o transferencias de archivos. En cambio, UDP es un protocolo sin conexión, más rápido pero sin garantía de

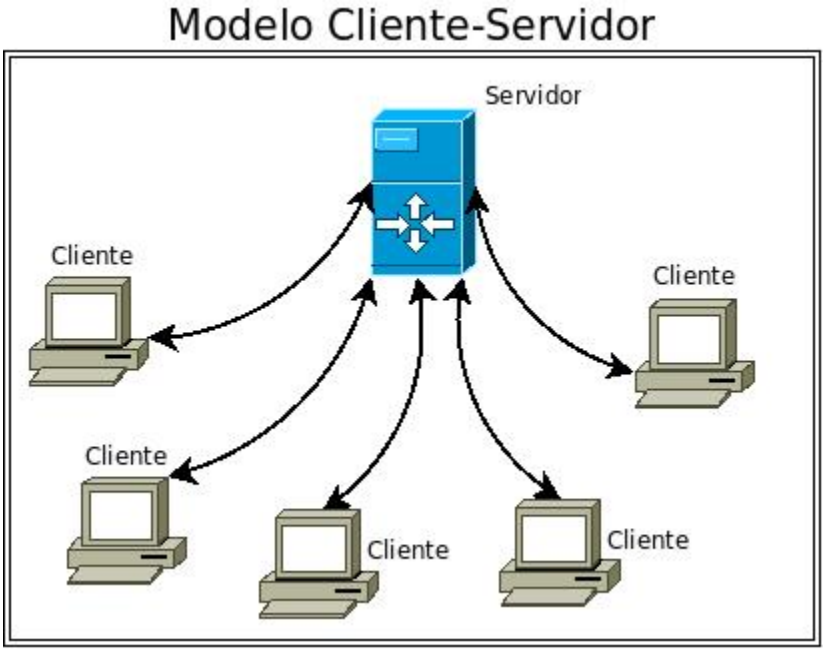


entrega, adecuado para aplicaciones en tiempo real como transmisiones de video en vivo, videojuegos en línea y videollamadas.

Además de la distinción entre TCP y UDP, los sockets pueden clasificarse en sockets de dominio Unix y sockets de red. Los primeros se utilizan para la comunicación entre procesos dentro de la misma máquina, mientras que los segundos permiten la interacción entre dispositivos en una red local o en internet. Esta versatilidad hace que los sockets sean una herramienta clave en la infraestructura de comunicación moderna.

La implementación de sockets es compatible con múltiples lenguajes de programación, incluidos C, Python, Java y JavaScript, lo que ha permitido su adopción en una amplia gama de aplicaciones. En el desarrollo de software, los sockets son esenciales para construir arquitecturas de cliente-servidor, donde un servidor escucha las solicitudes de múltiples clientes y responde de manera eficiente.

Gracias a su flexibilidad y eficiencia, los sockets han evolucionado junto con la tecnología de redes, adaptándose a nuevos modelos como WebSockets, que permiten la comunicación en tiempo real entre navegadores y servidores, optimizando aplicaciones como chats en vivo y sistemas de colaboración en línea. Esta evolución demuestra la importancia de los sockets en el mundo digital actual, facilitando la comunicación entre millones de dispositivos y servicios conectados.



Planteamiento del Problema

En esta práctica, se busca desarrollar una aplicación que implemente el uso de sockets para comprender su funcionamiento, administración y beneficios dentro de un sistema de comunicación en red.

Los sockets permiten el intercambio de datos entre procesos en diferentes dispositivos o dentro de una misma máquina, facilitando la comunicación en arquitecturas cliente-servidor. Su correcto uso es fundamental para garantizar una transmisión eficiente y segura, lo que implica manejar aspectos clave como la selección del protocolo adecuado (TCP o UDP), la gestión de conexiones y la sincronización de datos.

A través de esta práctica, se explorará cómo crear, configurar y gestionar sockets, además de analizar los principales desafíos asociados, como la latencia en la comunicación, la seguridad en la transferencia de datos y la gestión de múltiples clientes. Esto permitirá comprender mejor la importancia de los sockets en el desarrollo de aplicaciones de red eficientes y robustas.

Propuesta de Solución

Para abordar el problema planteado, la solución consistirá en el desarrollo de una aplicación cliente-servidor utilizando sockets para facilitar la comunicación entre un servidor y múltiples clientes.

Definición del entorno de comunicación:

La aplicación se estructurará como un modelo cliente-servidor, donde el servidor escuchará las solicitudes de conexión de los clientes. Para la comunicación, se utilizarán sockets de flujo (TCP), que garantizan la transmisión confiable de datos entre las partes involucradas.

Implementación del servidor:

El servidor se encargará de escuchar en un puerto específico por conexiones entrantes de los clientes. Al recibir una solicitud de conexión, el servidor aceptará la conexión y procederá a la comunicación con el cliente. Se implementará la capacidad de manejar múltiples clientes simultáneamente utilizando hilos de ejecución. Cada cliente conectado será gestionado por un hilo independiente, lo que permitirá que el servidor responda de manera eficiente a múltiples peticiones sin bloquearse.

Desarrollo del cliente:

El cliente iniciará la conexión al servidor utilizando un socket. Una vez conectado, podrá enviar solicitudes de datos o comandos al servidor y recibir respuestas. Los clientes podrán estar distribuidos en diferentes dispositivos dentro de la misma red local o incluso a través de internet, facilitando así la comunicación remota.

Manejo de errores y desconexiones:

Para garantizar la robustez de la aplicación, se implementarán mecanismos de manejo de errores, como la detección de desconexiones inesperadas y la gestión adecuada de los errores de comunicación, tales como problemas de red o tiempos de espera. Esto permitirá que tanto el servidor como los clientes sigan operando sin interrupciones en situaciones adversas.

Sincronización y control de flujo:

Aunque los sockets de TCP proporcionan una entrega ordenada de los paquetes, se implementarán estrategias adicionales para gestionar la sincronización y el control del flujo de datos. El servidor podrá gestionar múltiples clientes de manera eficiente, utilizando un sistema de cola o buffer para almacenar los mensajes entrantes y salientes. Esto ayudará a mantener la fluidez en la comunicación y evitará posibles bloqueos o caídas del sistema.

Optimización de la comunicación:

Con el objetivo de mejorar la eficiencia de la transmisión de datos, se podrá implementar una técnica de compresión de datos antes de enviarlos a través del socket, lo que reducirá el tiempo de transferencia, especialmente en redes con limitaciones de ancho de banda. Además, se explorarán opciones para reducir la latencia, como el ajuste de los parámetros del socket (por ejemplo, tamaño del buffer, tiempos de espera) para adaptarse a las condiciones de red.

Escalabilidad y seguridad:

Para garantizar que la solución sea escalable y segura, se podrá implementar cifrado SSL/TLS para proteger los datos enviados entre el cliente y el servidor, evitando posibles ataques o espionaje de la información. Además, el servidor podrá gestionarse para aceptar un número dinámico de conexiones, lo que permitirá que la aplicación crezca conforme se necesiten más clientes sin afectar el rendimiento del servidor.

Con esta solución, se logrará una implementación eficiente de la comunicación entre clientes y servidores mediante sockets, abordando los desafíos comunes como la gestión de múltiples conexiones, la sincronización de datos y la seguridad de la información. Esto proporcionará una base sólida para el desarrollo de aplicaciones distribuidas y sistemas de comunicación en red.

Materiales y Métodos Empleados

Software:

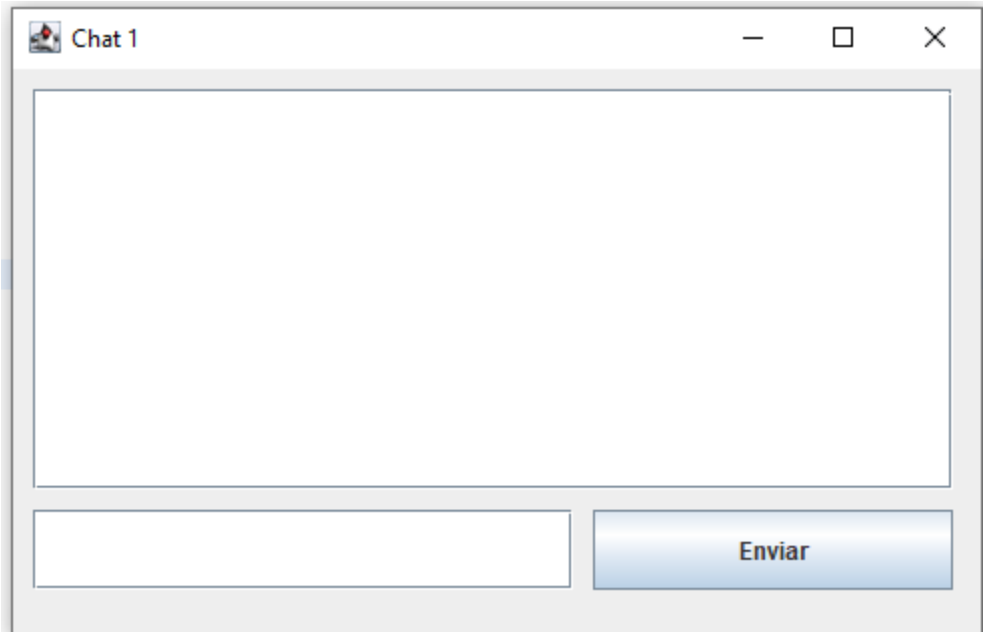
- Java Development Kit (JDK) 8 o superior – Para compilar y ejecutar programas en Java.
- Entorno de desarrollo integrado (IDE) de Eclipse
- Bibliotecas estándar de Java – ServerSocket: Esta clase se utiliza para crear un servidor que escucha las solicitudes de conexión entrantes de los clientes. ServerSocket acepta conexiones en un puerto específico y crea un socket de cliente cada vez que se establece una conexión.
    - Métodos principales:
        - accept(): Acepta una conexión entrante y devuelve un objeto de tipo Socket.
        - bind(SocketAddress endpoint): Asocia el socket a un puerto específico.
- Socket: La clase Socket se utiliza para crear un socket de cliente que se conecta a un servidor. Esta clase permite la comunicación bidireccional a través del socket, enviando y recibiendo datos entre el cliente y el servidor.
    - Métodos principales:
        - connect(SocketAddress endpoint): Conecta el socket al servidor especificado por su dirección y puerto.
        - getInputStream(): Devuelve un flujo de entrada para recibir datos.
        - getOutputStream(): Devuelve un flujo de salida para enviar datos.
        - close(): Cierra el socket.
- Otras Bibliotecas fundamentales:
    - java.net.ServerSocket
    - java.net.Socket
    - java.io.InputStream
    - java.io.OutputStream
    - java.io.BufferedReader
    - java.io.PrintWriter

Desarrollo de la Solución

En esta práctica, se ha creado una aplicación de chat utilizando sockets TCP para la comunicación entre clientes y un servidor. A través de la interfaz gráfica, los usuarios pueden enviar mensajes y recibir los de otros usuarios de manera simultánea. Este sistema de chat se ha implementado usando Java con el patrón de diseño Observer para actualizar las interfaces de usuario (GUI) en tiempo real cuando se recibe un nuevo mensaje. Este se divide en 4 archivos .java, los cuales se explicarán a continuación.

Cliente 1

Esta clase se representa la interfaz gráfica de un usuario en el chat, y permite que el usuario envíe mensajes al servidor.



Componentes de la GUI:

JTextArea (Area1): Es un área de texto donde se muestran los mensajes recibidos.

JTextField (text1): Es un campo de texto donde el usuario ingresa su mensaje.

JButton (btn1): Es un botón que el usuario presiona para enviar su mensaje.

```
getContentPane().setLayout(null);

Area1 = new JTextArea();

panel1 = new JScrollPane(Area1);
panel1.setBounds(10, 10, 460, 200);
getContentPane().add(panel1);

text1 = new JTextField();
text1.setBounds(10, 220, 270, 40);
getContentPane().add(text1);
```

Acción del botón "Enviar":

Cuando el usuario hace clic en el botón "Enviar", el mensaje que se encuentra en text1 se toma, se le antepone el prefijo "1.- " para indicar que es el mensaje del Chat1 y se añade a la JTextArea (Area1).

```
btn1 = new JButton("Enviar");
btn1.setBounds(290, 220, 180, 40);
getContentPane().add(btn1);

ActionListener act1 = new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        String mensaje = "1.- " + text1.getText() + "\n";
        Area1.append(mensaje);

        Cliente clien = new Cliente(6000, mensaje);
        Thread hilo2 = new Thread(clien);
        hilo2.start();
    }
};
```

Se crea un objeto Cliente, que se encarga de enviar el mensaje al servidor, y se ejecuta en un hilo (Thread) para no bloquear la interfaz gráfica del cliente.

Conexión con el servidor:

Se crea una instancia de la clase Servidor que se ejecuta en el puerto 5000 y se añade como observador de la clase Chat1. Esto permite que Chat1 reciba notificaciones cuando el servidor envíe mensajes.

```
Servidor server = new Servidor(5000);
server.addObserver(this);

Thread hilo1 = new Thread(server);
hilo1.start();
```

Método update:

Implementa el método de la interfaz Observer para recibir los mensajes del servidor y actualiza el área de texto con los nuevos mensajes.

```
public void update(Observable o, Object arg) {
    this.Area1.append((String) arg);
}
```

Cliente 2

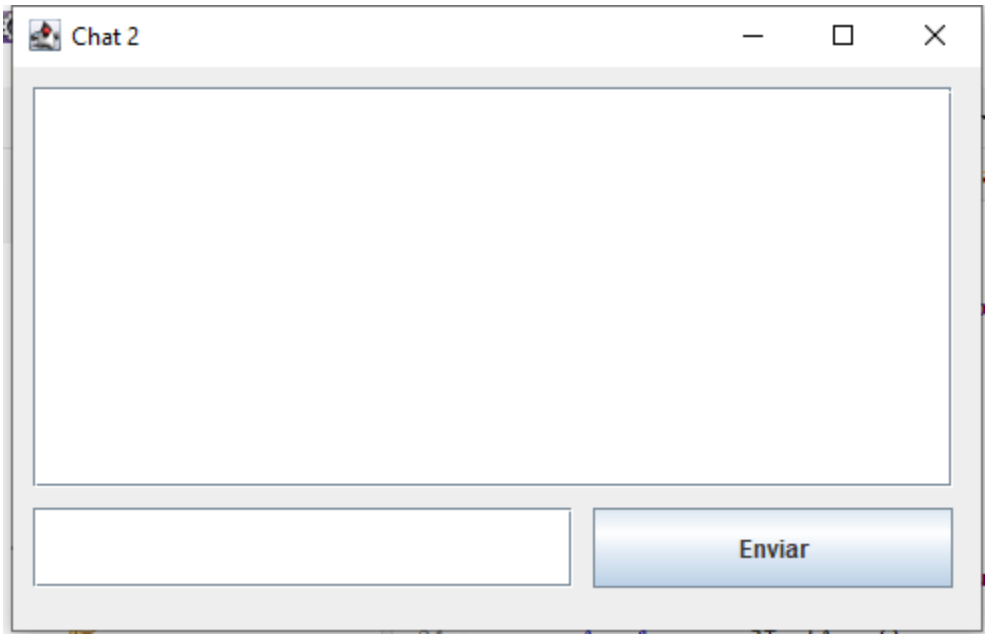
La clase Chat2 funciona de manera similar a Chat1, pero está configurada para conectarse al servidor en el puerto 6000. La estructura y funcionalidad son iguales, con la única diferencia de que el mensaje enviado se marca con el prefijo "2.- " para indicar que proviene del Chat2.

```
ActionListener act1 = new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        String mensaje = "2.- " + text1.getText() + "\n";
        Area1.append(mensaje);

        Cliente clien = new Cliente(5000, mensaje);
        Thread hilo2 = new Thread(clien);
        hilo2.start();
    }
};
btn1.addActionListener(act1);
```

Al igual que en Chat1, el cliente envía el mensaje al servidor y se ejecuta en un hilo para mantener la interfaz gráfica activa.

Al igual que Chat1, Chat2 implementa el patrón Observer y se actualiza cada vez que el servidor envía un mensaje.



Cliente

La clase Cliente es responsable de enviar mensajes al servidor. Cada vez que se crea una nueva instancia de Cliente, se le pasa el puerto al que debe conectarse y el mensaje que se enviará.

La clase utiliza un Socket TCP para conectarse al servidor en la dirección 127.0.0.1 (localhost) y en el puerto especificado (5000 o 6000).

```
private int puerto;
private String mensaje;

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

public void run() {

    Socket sc = null;
    DataOutputStream out;
    String host = "127.0.0.1";
```

El mensaje se envía utilizando un DataOutputStream, lo que garantiza que los datos se envíen de manera ordenada y adecuada.

```
public void run() {

    Socket sc = null;
    DataOutputStream out;
    String host = "127.0.0.1";

    try {
        sc = new Socket(host, puerto);

        out = new DataOutputStream(sc.getOutputStream());

        out.writeUTF(mensaje);

        sc.close();
    }catch(Exception e) {
        e.printStackTrace();
    }
}
```

Después de enviar el mensaje, el socket se cierra para liberar los recursos.

Servidor

El servidor se encarga de recibir los mensajes enviados por los clientes y luego notificar a todos los observadores (los clientes) para que actualicen sus interfaces gráficas.

Escucha en un puerto: El servidor crea un ServerSocket y escucha las conexiones entrantes en un puerto específico (5000 o 6000).

```
private int puerto;

public Servidor(int puerto) {
    this.puerto = puerto;
}

public void run() {

    ServerSocket server = null;
    Socket sc = null;
    DataInputStream in;

    try {
        server = new ServerSocket(puerto);
        System.out.println("Servidor Iniciado");
```

Cuando un cliente se conecta al servidor, el método accept() es bloqueante y espera hasta que una conexión sea establecida.

```
try {
    server = new ServerSocket(puerto);
    System.out.println("Servidor Iniciado");
```

Una vez que se establece la conexión con el cliente, se utiliza un DataInputStream para leer el mensaje enviado por el cliente.

```
in = new DataInputStream(sc.getInputStream());

String mensaje = in.readUTF();
System.out.println(mensaje);
```

Tras recibir el mensaje, el servidor llama a setChanged() y notifyObservers(mensaje) para notificar a todos los clientes conectados que un nuevo mensaje ha llegado. El mensaje es pasado como parámetro en el método notifyObservers() para que los clientes lo reciban.

```
this.setChanged();
this.notifyObservers(mensaje);
this.clearChanged();

sc.close();
```

Después de procesar el mensaje, el servidor cierra la conexión con el cliente mediante sc.close().

Resultados

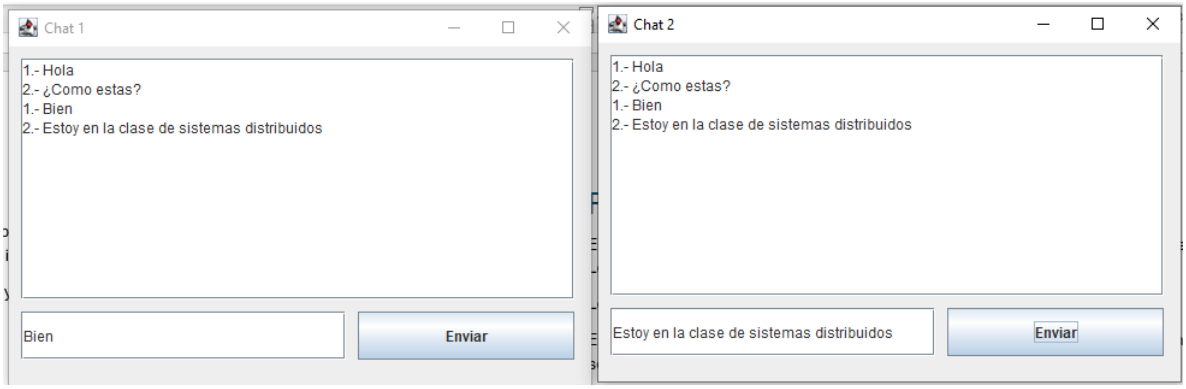
En esta práctica, hemos implementado una aplicación de chat utilizando sockets TCP en Java. Los principales componentes del sistema son:

Los clientes (Chat1 y Chat2) que se encargan de enviar y recibir mensajes.

El servidor que recibe los mensajes de los clientes y los distribuye a los observadores para que se actualicen en tiempo real.

El uso del patrón Observer ha sido fundamental para que los clientes reciban los mensajes en tiempo real sin necesidad de hacer consultas constantes al servidor. Además, el uso de hilos garantiza que la interfaz gráfica de los clientes no se bloquee al enviar o recibir mensajes.

Este enfoque básico puede ser ampliado para gestionar más usuarios, mejorar la seguridad, manejar múltiples conexiones simultáneas, y muchas otras funcionalidades.



Conclusión

Esta practica nos muestra cómo funciona la comunicación entre cliente y servidor utilizando sockets TCP en Java. A diferencia de un programa tradicional donde los procesos se ejecutan de manera secuencial, en este caso, tenemos varios componentes trabajando al mismo tiempo, como los clientes enviando mensajes y el servidor gestionando las conexiones. La implementación de hilos permite que el envío y la recepción de mensajes no interfieran con la interacción del usuario en las interfaces gráficas, lo que hace que el programa sea más eficiente y responsivo. Además, el uso del patrón de diseño Observer asegura que los mensajes se actualicen en tiempo real en todas las interfaces conectadas, permitiendo una experiencia de chat dinámica. La manera en que el servidor y los clientes interactúan refleja cómo los sistemas distribuidos manejan la comunicación en redes, donde el orden de los mensajes y las conexiones no siempre es predecible, ya que depende de varios factores como el tiempo de ejecución y el manejo de los recursos por parte del sistema operativo.