



**Universidad Veracruzana**  
**Lic. En Ingeniería de ciberseguridad e  
infraestructura de computo**

**Sistema de mensajería LAN  
sin hilos en C++**

**Raúl Nava Soler**  
**Juan Elías Antonio Ramírez**  
**Kevin Marzuk Jerónimo Rojano**

9 de junio de 2025

---

## INDICE

1. Introducción y Planteamiento del Problema	1
2. Marco Teórico (Estructuras y Componentes Usados)	1
2.1. Tabla Hash para Gestión de Usuarios	1
Archivos de Implementación:	1
Componentes y Funciones Clave:	2
2.2. Lista Simplemente Enlazada para Clientes Conectados	2
Archivos de Implementación:	3
Componentes y Funciones Clave:	3
2.3. Sockets y Concurrencia con select()	3
3. Diseño de la Solución	5
3.1. Diagramas de clases UML	5
3.2. Diagramas de flujo del servidor	5
4. Manual de Uso	6
1. Modificar “server.cpp”	6
2. Compilación de servidor	6
4. Modificar “cliente.cpp”	7
5. Compilación de cliente	7
8. Interacción con el programa cliente	8
6. Análisis de complejidad	9
7. Conclusiones y Trabajo Futuro	10
8. Referencias	11
9. Anexos	12
10. Resultados	15

## 1. Introducción y Planteamiento del Problema

En el mundo de la computación distribuida, la comunicación entre procesos es fundamental. Los sistemas de mensajería o chat son un ejemplo clásico de esta necesidad, donde múltiples usuarios deben poder intercambiar información en tiempo real de manera eficiente y ordenada. El reto computacional principal consiste en gestionar múltiples conexiones simultáneas, autenticar usuarios de forma segura y distribuir los mensajes de manera fiable a todos los participantes.

Este proyecto, "Sistema de mensajería local basado en sockets", aborda este problema implementando un servidor de chat multicliente y su correspondiente cliente en C++. El objetivo es aplicar estructuras de datos fundamentales para resolver los desafíos de la gestión de usuarios y la comunicación en red. La solución desarrollada simula un entorno de chat real donde los usuarios pueden registrarse, iniciar sesión, y enviar mensajes que son transmitidos (broadcast) a todos los demás usuarios conectados, demostrando una solución práctica y funcional a un problema común del contexto computacional.

## 2. Marco Teórico (Estructuras y Componentes Usados)

Para construir la solución, se seleccionaron dos estructuras de datos principales: una **Tabla Hash** para la gestión y autenticación de credenciales de usuario y una **Lista Simplemente Enlazada** para administrar el estado de los clientes conectados en tiempo real. Adicionalmente, el núcleo del sistema se basa en la API de **Sockets** para la comunicación en red.

### 2.1. Tabla Hash para Gestión de Usuarios

Se implementó una TablaHash para almacenar y validar las credenciales de los usuarios (usuario y contraseña). Esta estructura fue elegida por su alta eficiencia en operaciones de búsqueda e inserción, que en promedio tienen una complejidad de  $O(1)$ .

#### *Archivos de Implementación:*

- hashheader.hpp: Define la estructura y la interfaz de la clase.

- `hashhead.cpp`: Contiene la lógica de los métodos.

### **Componentes y Funciones Clave:**

- **struct Celda**: Representa cada una de las ranuras de la tabla. Contiene la clave (clave, que es la contraseña), el nombre de usuario (usuario), el estado de la celda (VACIO, OCUPADO), y un booleano (online) para verificar si el usuario ya tiene una sesión activa.
- **funcionHash(const Clave& clave)**: Es el corazón de la tabla. Genera un índice numérico a partir de una clave (la contraseña del usuario). La implementación actual suma los valores ASCII de los caracteres de la clave y aplica la operación módulo (%) con la capacidad de la tabla (CAPACIDAD) para asegurar que el índice esté dentro de los límites del arreglo.
- **Registrar(const string& usuario, const string& clave)**: Inserta un nuevo par usuario-contraseña en la tabla. Utiliza **sondeo lineal** para manejar colisiones: si la celda calculada por la `funcionHash` ya está ocupada, avanza a la siguiente celda disponible de manera secuencial. Antes de insertar, verifica si el usuario ya existe para evitar duplicados.
- **iniciarsesion(const string& usuario, const Clave& clave)**: Autentica a un usuario. Busca en la tabla una coincidencia de usuario y contraseña. Retorna 0 si las credenciales son correctas y el usuario no está en línea, 1 si el usuario ya está conectado, y 2 si las credenciales son incorrectas o el usuario no existe. También usa sondeo lineal para encontrar la celda correcta en caso de colisiones.
- **Desconectar(const string& usuario, const Clave& clave)**: Cambia el estado online de un usuario a offline. Esto es crucial para permitir que un usuario que se ha desconectado pueda volver a iniciar sesión.

## **2.2. Lista Simplemente Enlazada para Clientes Conectados**

Para gestionar los clientes activamente conectados al servidor, se utiliza una Lista simplemente enlazada. Esta estructura es ideal para manejar un número dinámico de elementos, ya que su tamaño puede crecer o decrecer en tiempo de ejecución sin necesidad de pre-asignar una cantidad fija de memoria.

### **Archivos de Implementación:**

- ListaEnlazada.hpp: Define la estructura del Nodo, del Cliente y la interfaz de la clase Lista.
- ListaEnlazada.cpp: Proporciona la lógica de las operaciones sobre la lista.

### **Componentes y Funciones Clave:**

- **struct Cliente:** Almacena toda la información relevante de un cliente conectado: Su descriptor de archivo de socket, el último mensaje recibido, su nombre de usuario, contraseña y su estado de autenticación.
- **class Nodo:** Es el bloque de construcción de la lista. Contiene un dato de tipo Cliente y un puntero (siguiente) al próximo nodo en la secuencia.
- **insertarAlFinal(Cliente valor):** Añade un nuevo cliente al final de la lista. Esta operación es muy eficiente ( $O(1)$ ) porque la clase Lista mantiene un puntero (cola) directamente al último nodo, evitando la necesidad de recorrer toda la lista.
- **eliminarNodo(Cliente valor):** Elimina un cliente de la lista, por ejemplo, cuando se desconecta. La operación requiere buscar el nodo a eliminar, lo que resulta en una complejidad de  $O(n)$ , donde  $n$  es el número de clientes conectados.
- **obtenerCabeza():** Devuelve un puntero al primer nodo de la lista (cabeza), lo que permite recorrerla para, por ejemplo, retransmitir un mensaje a todos los clientes.
- **ObtenerCantidad():** Retorna el número de clientes conectados, útil para verificar si el servidor ha alcanzado su capacidad máxima.

### **2.3. Sockets y Concurrencia con select()**

La comunicación en red se maneja a través de la API de Sockets, un estándar para la comunicación entre procesos en redes TCP/IP.

- **Cliente (cliente.cpp):** Crea un socket y se conecta al servidor. La comunicación con el usuario y el servidor se maneja de forma asíncrona usando la función `select()`. Esto permite al cliente esperar simultáneamente por entrada del teclado (`STDIN_FILENO`) y por mensajes entrantes del servidor sin bloquearse en una sola operación.
- **Servidor (server.cpp):**

- o **crearSocketServidor**: Crea el socket principal, lo enlaza a un puerto (25565) y lo pone en modo de escucha para aceptar nuevas conexiones.
- o **aceptarCliente**: Acepta una conexión entrante, realiza el proceso de autenticación interactuando con la TablaHash, y si es exitoso, agrega el nuevo cliente a la Lista de clientes conectados.
- o **manejarMensajes**: Itera sobre la lista de clientes para procesar los mensajes entrantes.
- o **Bucle Principal y select()**: El corazón del servidor es un bucle while(true) que utiliza select(). Esta función monitoriza un conjunto de descriptores de archivo (el socket del servidor para nuevas conexiones, y los sockets de todos los clientes conectados para nuevos mensajes). Esto permite al servidor manejar múltiples clientes de manera concurrente en un único hilo de ejecución, evitando la complejidad de la programación multihilo.

### **3. Diseño de la Solución**

El diseño combina las estructuras de datos y los componentes de red en un sistema único.

#### **3.1. Diagramas de clases UML**

El siguiente diagrama muestra las relaciones entre las clases principales del sistema. (véase figura 9.1 Diagrama de Clases UML).

#### **3.2. Diagramas de flujo del servidor**

Este diagrama ilustra la lógica principal del bucle del servidor. (véase figura 9.2.1 Diagrama de Flujo del Servidor & 9.2.2 Diagrama de Flujo del Servidor).

## 4. Manual de Uso

### 1. Modificar "server.cpp"

Abrir el archivo server.cpp, dirigirse a la línea 23 y a partir de ahí registrar los usuarios y contraseñas autorizadas para conectar al servidor con el siguiente formato:

```
Modificar usuarios y contraseñas C++  
23 void registrarUsuarios(TablaHash& tabla){  
24     tabla.Registrar("USUARIO", "CONTRASEÑA");
```

Para compilar y ejecutar el sistema, se necesita un compilador de C++ (como g++) y un entorno tipo UNIX (Linux, macOS, o WSL en Windows).

### 2. Compilación de servidor

Abre una terminal y ejecuta los siguientes comandos para compilar el servidor

```
Compilar Server Bash  
g++ hashhead.cpp ListaEnlazada.cpp server.cpp -o server
```

Abrir una terminal, sabiendo que estamos en un entorno Linux y en una red local, la persona que está corriendo el programa.

### 3. Compartir IP, usuario y contraseña.

La persona la cual correrá el programa server debe abrir una terminal y ejecutar el siguiente comando:

```
Obtener mi IP Bash  
$ ip a  
  
3: wlp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000  
    link/ether b4:6b:fc:xx:xx:xx brd ff:ff:ff:ff:ff:ff  
    inet 192.168.0.25/24 brd 192.168.0.255 scope global dynamic wlp3s0  
        valid_lft 86378sec preferred_lft 86378sec  
    inet6 fe80::b66b:fcff:feXX:XXXX/64 scope link  
        valid_lft forever preferred_lft forever  
  
→ Tu IP local por Wi-Fi es 192.168.0.25  
  
(Nota: los nombres como wlp3s0 pueden variar; podrían ser wlan0, wlan1, etc.)
```

Una vez conociendo la IP de tu maquina debes compartirla a tus usuarios, además de su usuario y contraseña.



#### 4. Modificar “cliente.cpp”

Debes abrir el archivo cliente.cpp, dirigirte a la línea 25 y colocar la dirección IP proporcionada por tu administrador de server.

```
Modificar la IP a la cual conectarse C++  
25  inet_pton(AF_INET, "IP DEL SERVIDOR", &servidor.sin_addr);
```

#### 5. Compilación de cliente

Abre terminal y ejecuta el siguiente comando para compilar el cliente

```
Compilar Cliente Bash  
g++ cliente.cpp -o cliente
```

#### 6. Ejecución de server

Situado en el directorio en el cual previamente compilaste el servidor debes ejecutar el siguiente comando:

```
Ejecutar Server Bash  
./server
```

El servidor comenzará a escuchar en el puerto 25565 y creará un archivo logs.txt para registrar eventos. Para detener el servidor, presiona Enter en la terminal donde se está ejecutando, al hacer esto modificará el nombre del log agregando fecha y hora de cierre del servidor.

#### 7. Ejecución de cliente

Ahora, sabiendo que el servidor se encuentra prendido, debes ejecutar el código cliente con el siguiente comando:

```
Ejecutar Cliente Bash  
./cliente
```

## **8. Interacción con el programa cliente**

1. **Autenticación:** Al conectarse, el servidor solicitará Usuario: y Password:. Ingresa una de las credenciales pre-registradas (ej. Usuario: magic, Password: tira1d20).
2. **Envío de Mensajes:** Una vez autenticado, todo lo que escribas en la terminal del cliente y presiones Enter será enviado al servidor, que lo retransmitirá a todos los demás clientes conectados.
3. **Desconexión:** Para desconectar un cliente, simplemente cierra su terminal o presiona Ctrl+C o CTRL+Z. El servidor detectará la desconexión y actualizará el estado del usuario.

## **9. Interacción con el programa server**

1. **Apagar servidor:** Para apagar el servidor basta con presionar "Enter", lo que provocará la salida del bucle principal, desconectando a todos los usuarios y cambiando el nombre del log con la fecha y hora del cierre del server.

## 6. Análisis de complejidad

El análisis de complejidad es crucial para entender el rendimiento del sistema.

- **Tabla Hash (TablaHash):**
  - **Inserción (Registrar) y Búsqueda (iniciarsesion):** En el caso promedio, la complejidad es  $O(1)$ . En el peor de los casos (cuando hay muchas colisiones y el sondeo lineal debe recorrer gran parte del arreglo), la complejidad es  $O(N)$ , donde  $N$  es la capacidad de la tabla.
- **Lista Enlazada (Lista):**
  - **Inserción (insertarAlFinal):** Gracias al puntero a la cola, la complejidad es constante:  $O(1)$ .
  - **Eliminación (eliminarNodo):** Requiere una búsqueda previa, por lo que la complejidad es lineal:  $O(k)$ , donde  $k$  es el número de clientes conectados.
  - **Recorrido (para broadcast en manejarMensajes):** Se debe visitar cada nodo, resultando en una complejidad de  $O(k)$ .

El rendimiento general del servidor por cada ciclo del bucle principal está dominado por el recorrido de la lista de clientes para el broadcast de mensajes, lo que le da una complejidad de  $O(k)$ . Dado que el número de clientes (MAX\_CLIENTES) está limitado a 10, el rendimiento es excelente para la escala del problema.

## 7. Conclusiones y Trabajo Futuro

El proyecto cumplió exitosamente con su objetivo de desarrollar un sistema de mensajería funcional aplicando estructuras de datos vistas en el curso. La **Tabla Hash** probó ser una elección eficiente para la autenticación de usuarios, mientras que la **Lista Enlazada** ofreció la flexibilidad necesaria para gestionar un número variable de clientes conectados. La utilización de `select()` permitió un manejo concurrente de clientes de forma simple y efectiva en un solo hilo.

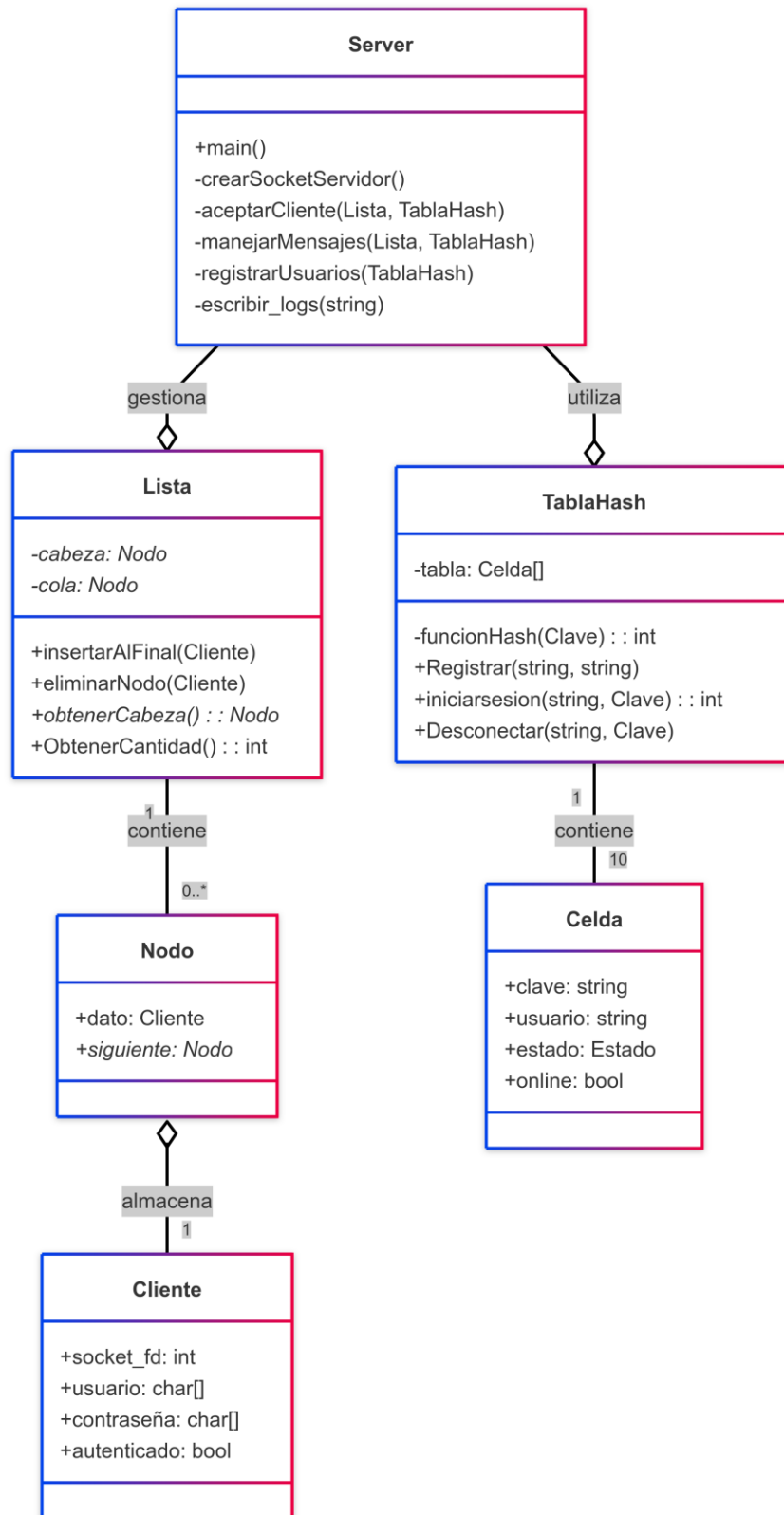
Como **trabajo futuro**, se proponen las siguientes mejoras:

1. **Seguridad Mejorada:** Implementar el hashing de contraseñas con un "salt" antes de almacenarlas, en lugar de guardarlas en texto plano.
2. **Persistencia de Datos:** Conectar el servidor a una base de datos para que los usuarios registrados no se pierdan al reiniciar el servidor.
3. **Funcionalidades Adicionales:**
  - a. Implementar **mensajes privados** entre usuarios.
  - b. Crear **canales o salas de chat** temáticas.
  - c. Añadir un historial de mensajes que se envíe a los nuevos clientes al conectarse.
4. **Escalabilidad:** Para soportar un número mucho mayor de usuarios, se podría migrar de `select()` a `epoll` (en Linux) y considerar una arquitectura multihilo o basada en eventos para distribuir la carga de trabajo.

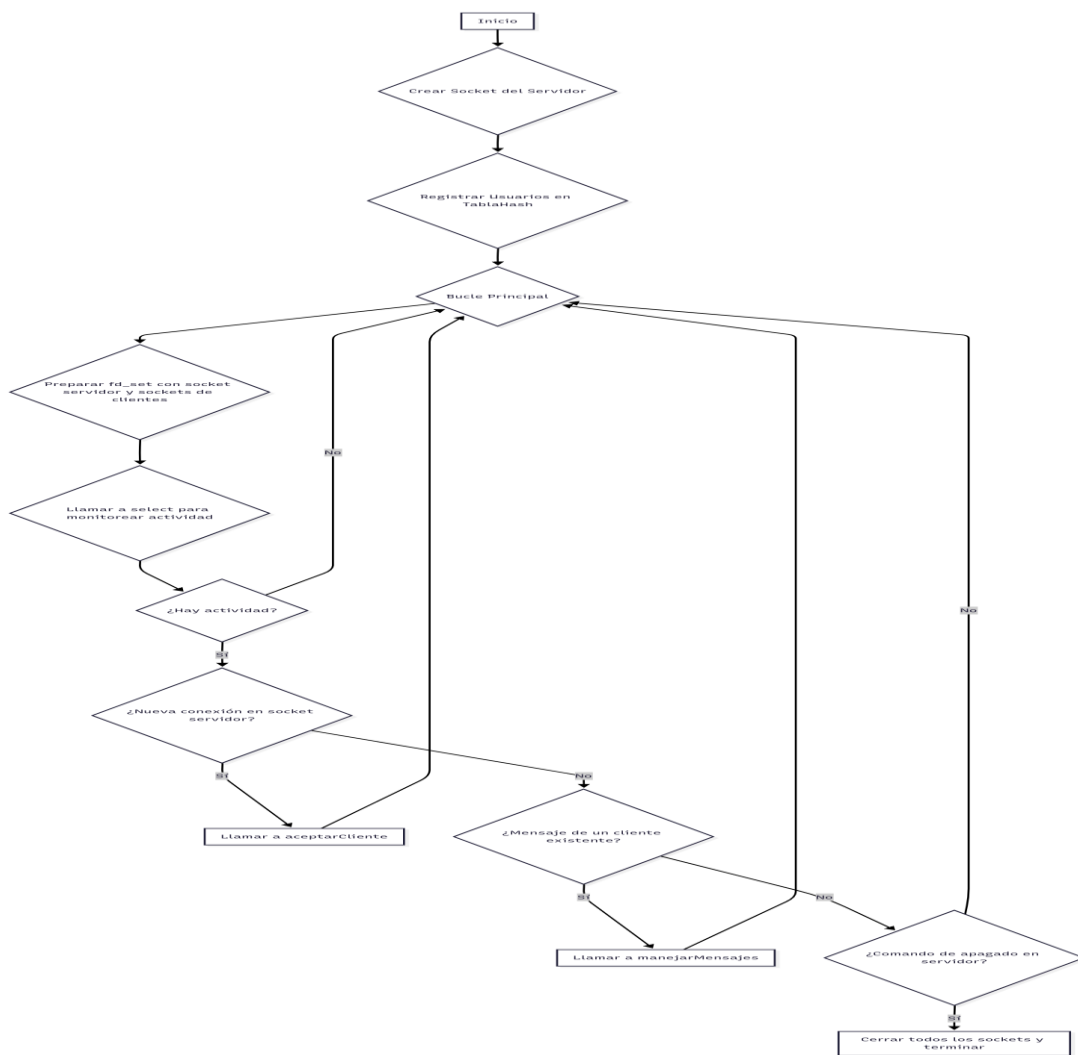
## 8. Referencias

1. W. R. Stevens, B. Fenner, and A. M. Rudoff, UNIX Network Programming, Volume 1: The Sockets Networking API, 3rd ed. Addison-Wesley Professional, 2003.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed. MIT Press, 2009.
3. M. A. Weiss, Data Structures and Algorithm Analysis in C++, 4th ed. Pearson, 2013.
4. "select(2) - Linux man page," linux.die.net. [Online]. Available: <https://linux.die.net/man/2/select>. [Accessed: Jun. 08, 2025].
5. Beej's Guide to Network Programming. [Online]. Available: <https://beej.us/guide/bgnet/>. [Accessed: Jun. 08, 2025].
6. LinuxHowtos, «www.LinuxHowtos.org howtos, tips&tricks and tutorials for linux,» S. F.. [En línea]. Available: [https://www.linuxhowtos.org/C\\_C++/socket.htm](https://www.linuxhowtos.org/C_C++/socket.htm). [Último acceso: 28 Mayo 2025].
7. LinuxHowtos, «www.LinuxHowtos.org howtos, tips&tricks and tutorials for linux,» S. F.. [En línea]. Available: <https://www.linuxhowtos.org/manpages/2/socket.htm>. [Último acceso: 30 Mayo 2025].
8. IBM, «fileno() — Get the file descriptor from an open stream» IBM, 25 Junio 2021. [En línea]. Available: <https://www.ibm.com/docs/en/zos/2.4.0?topic=functions-fileno-get-file-descriptor-from-open-stream>. [Último acceso: 2025 Junio 1].

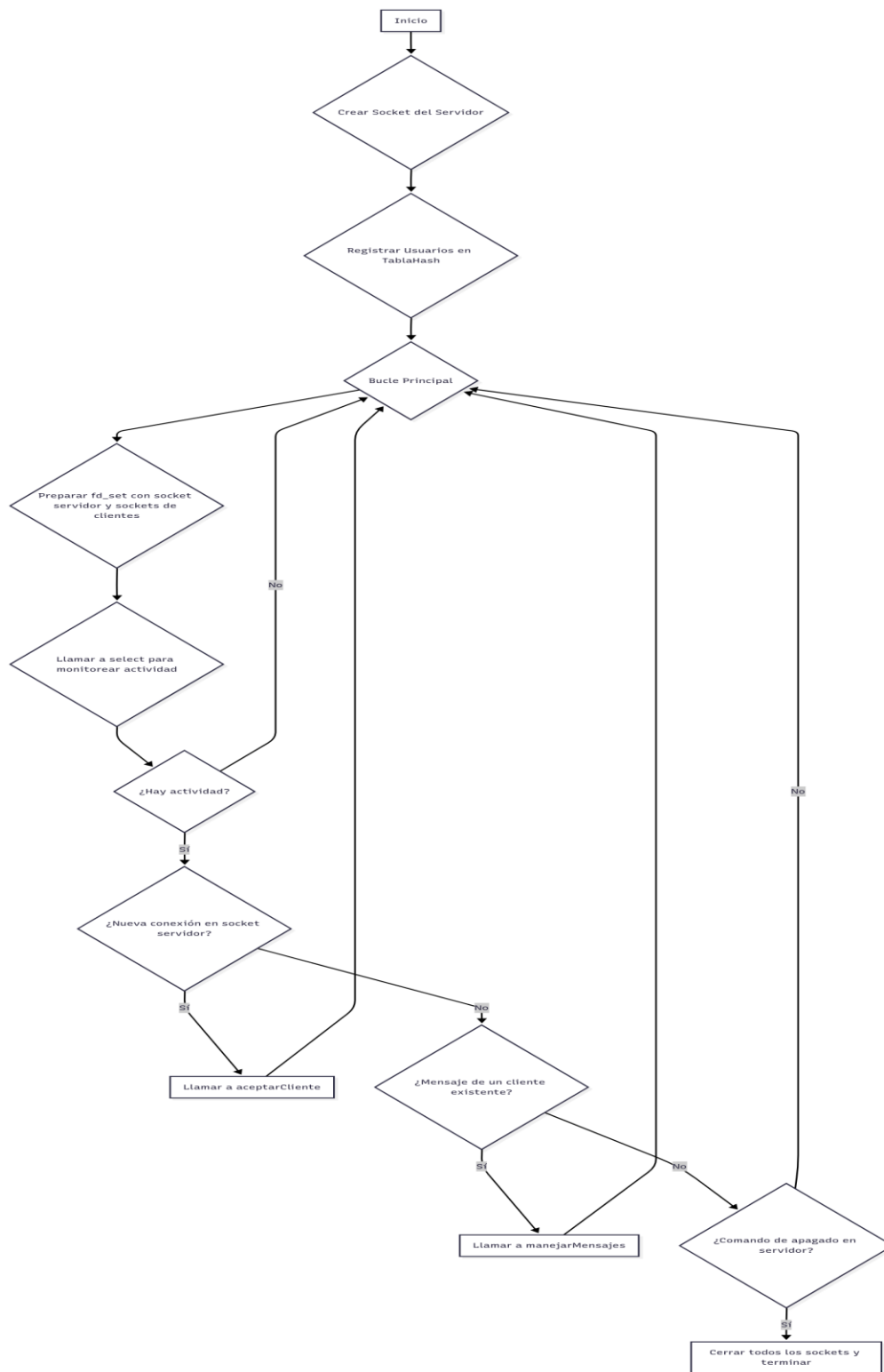
## 9. Anexos



9.1 Diagrama de Clases UML



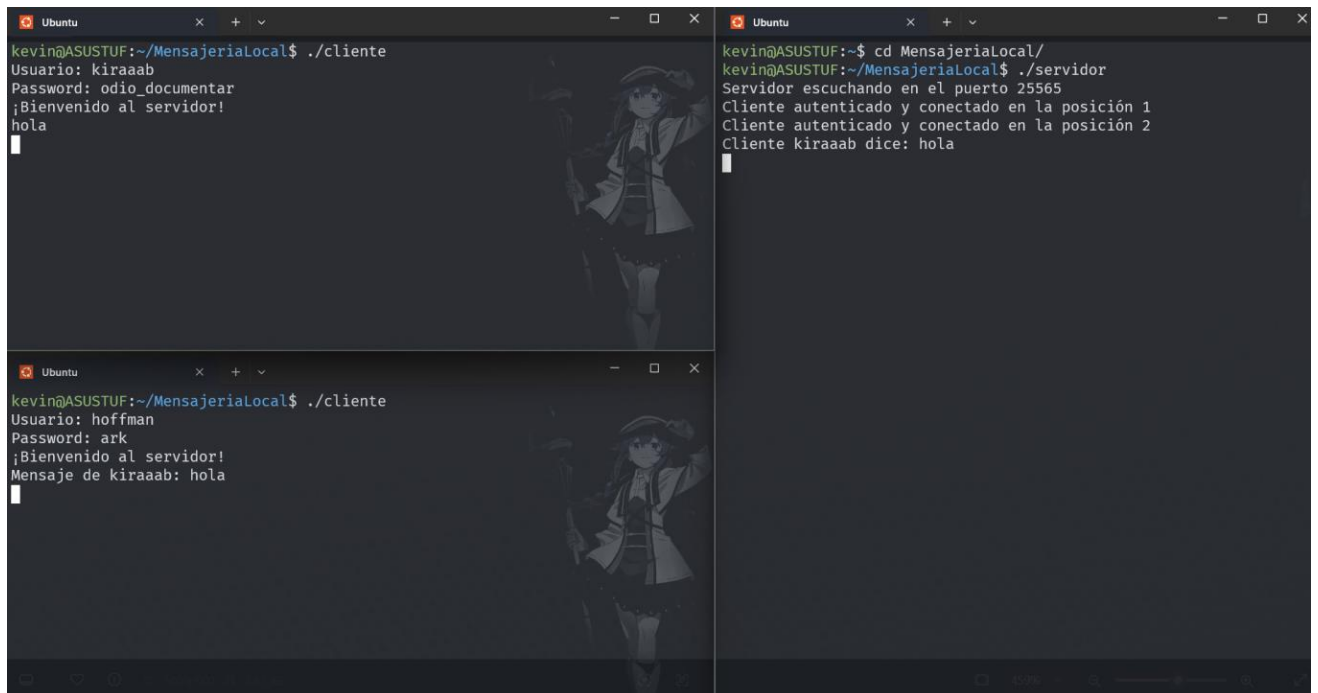
9.2.1 Diagrama de Flujo del Servidor



9.2.2 Diagrama de Flujo del Servidor



## 10. Resultados



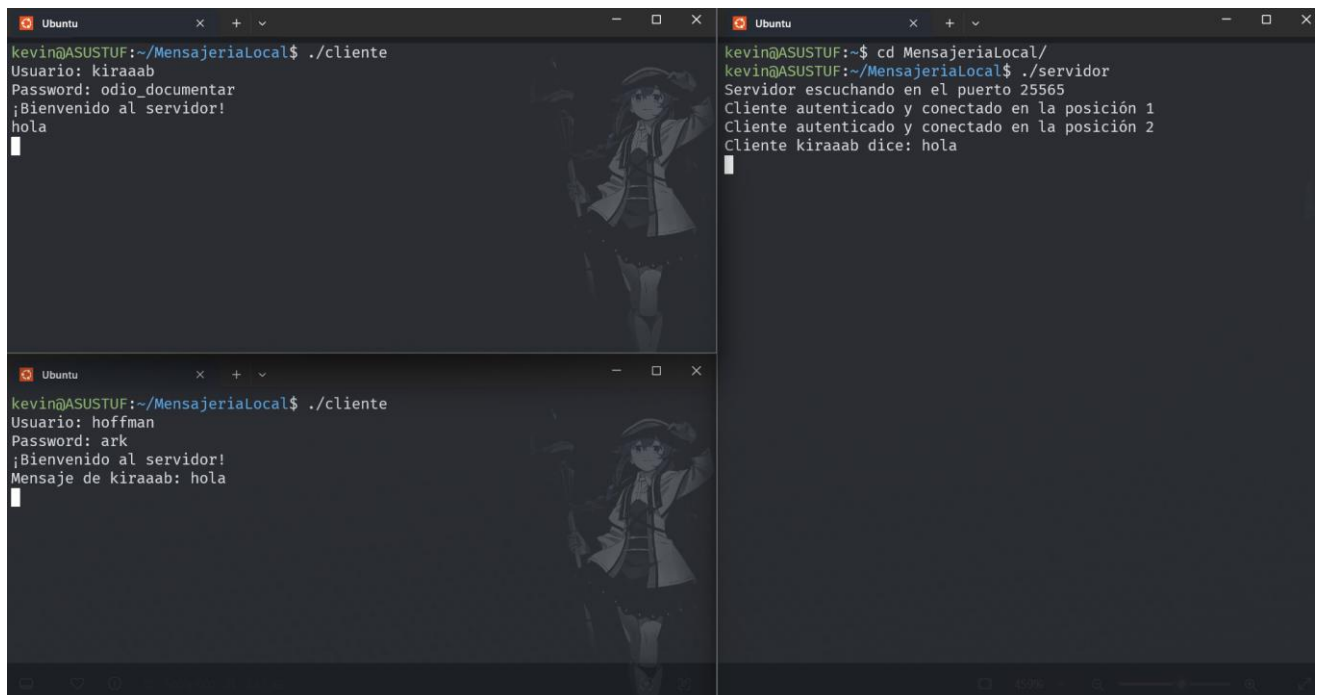
```
kevin@ASUSTUF:~/MensajeriaLocal$ ./cliente
Usuario: kiraab
Password: odio_documentar
¡Bienvenido al servidor!
hola

```

```
kevin@ASUSTUF:~/MensajeriaLocal$ ./servidor
Servidor escuchando en el puerto 25565
Cliente autenticado y conectado en la posición 1
Cliente autenticado y conectado en la posición 2
Cliente kiraaab dice: hola

```

Resultados 1



```
kevin@ASUSTUF:~/MensajeriaLocal$ ./cliente
Usuario: kiraab
Password: odio_documentar
¡Bienvenido al servidor!
hola

```

```
kevin@ASUSTUF:~/MensajeriaLocal$ ./servidor
Servidor escuchando en el puerto 25565
Cliente autenticado y conectado en la posición 1
Cliente autenticado y conectado en la posición 2
Cliente kiraaab dice: hola

```

Resultados 2

```
Kevin@ASUSTUF:~/MensajeriaLocal$ ./cliente
Usuario: kiraab
Password: odio_documentar
;Bienvenido al servidor!
hola
█

Kevin@ASUSTUF:~/MensajeriaLocal$ ./cliente
Usuario: hoffman
Password: ark
;Bienvenido al servidor!
Mensaje de kiraab: hola
█

Kevin@ASUSTUF:~$ cd MensajeriaLocal/
Kevin@ASUSTUF:~/MensajeriaLocal$ ./servidor
Servidor escuchando en el puerto 25565
Cliente autenticado y conectado en la posición 1
Cliente autenticado y conectado en la posición 2
Cliente kiraab dice: hola
█
```

Resultados 3

```
Kevin@ASUSTUF:~/MensajeriaLocal$ ./cliente
Usuario: kiraab
Password: odio_documentar
;Bienvenido al servidor!
hola
█

Kevin@ASUSTUF:~/MensajeriaLocal$ ./cliente
Usuario: hoffman
Password: ark
;Bienvenido al servidor!
Mensaje de kiraab: hola
█

Kevin@ASUSTUF:~$ cd MensajeriaLocal/
Kevin@ASUSTUF:~/MensajeriaLocal$ ./servidor
Servidor escuchando en el puerto 25565
Cliente autenticado y conectado en la posición 1
Cliente autenticado y conectado en la posición 2
Cliente kiraab dice: hola
█
```

Resultados 4