Memoria Final – Proyecto de Servicios y Procesos

Título: Sistema de Chat en Grupo con Arquitectura Cliente-Servidor
Alumno: Rocío Fernández Besoy
Curso: 2º DAM, Programación de Servicios y Procesos
Fecha: 25/01/2026

1. Contexto

Descripción del proyecto
Este proyecto consiste en el desarrollo de una aplicación de chat en grupo basada en sockets TCP, con arquitectura cliente-servidor. Los usuarios pueden registrarse, iniciar sesión y unirse a grupos de chat predefinidos mediante una clave secreta. Cada grupo cuenta con un servidor independiente (un proceso Java ejecutándose en un puerto distinto), y permite que los mensajes se envíen en tiempo real a todos los usuarios conectados al mismo grupo, independientemente de si se encuentran en el mismo ordenador o en dispositivos diferentes.

El sistema permite múltiples sesiones simultáneas, incluso desde el mismo PC, donde varios clientes diferentes con usuarios distintos pueden conectarse al mismo grupo y chatear entre sí. Todos los mensajes se almacenan en una base de datos MySQL para mantener el historial, y al conectar, el usuario decide si cargar el historial completo o comenzar con un chat vacío. La interfaz gráfica de escritorio utiliza Swing, incluyendo un área de texto con scroll para ver mensajes, un campo de texto para enviar (clicando el botón Enter), opción para cargar historial, campo de búsqueda para filtrar mensajes por nombre de usuario, y un botón "Limpiar chat" que solo afecta la vista local.

La comunicación se realiza exclusivamente mediante sockets TCP (java.net.Socket y java.net.ServerSocket), y se gestiona concurrencia mediante hilos. Se implementa el patrón Observer para actualizar la interfaz gráfica en tiempo real. La persistencia se maneja mediante JDBC, y el código se organiza en paquetes con separación clara de responsabilidades (modelo, persistencia, servidor, cliente, gui, controller).

Este desarrollo me ha permitido aplicar conceptos avanzados de programación de servicios y procesos, como la gestión de procesos independientes, la sincronización en entornos concurrentes y la integración de una base de datos relacional. Además, refuerza el entendimiento de cómo se distribuye la lógica en un sistema cliente-servidor, donde el servidor se encarga de la persistencia y transmisión de mensajes, mientras el cliente se enfoca en la interfaz y la interacción del usuario.

2. Caso práctico

Objetivo

El objetivo de esta práctica es crear un sistema de chat en grupo que demuestre el uso de servicios distribuidos, con servidores independientes por grupo, persistencia de datos y actualización en tiempo real de la interfaz gráfica. Esto permite a los usuarios colaborar en conversaciones grupales de forma segura y eficiente, permitiendo múltiples participantes desde diferentes dispositivos y almacenando los datos en una base de datos central.

El proyecto resuelve un escenario real de comunicación colaborativa simplificado para enfocarse en los conceptos de la asignatura: sockets, hilos, patrones de diseño y JDBC.

Funcionalidades principales

- Registro e inicio de sesión: Los usuarios se registran con alias y contraseña (almacenada en texto plano, como indica el enunciado). El inicio de sesión valida las credenciales en la base de datos.
- Grupos precreados: Existen tres grupos definidos en la BD, cada uno con nombre, clave y puerto asignado. El cliente selecciona un grupo mediante su clave.
- Unirse a un grupo: El cliente envía alias, contraseña y clave del grupo al servidor correspondiente. Si es válido, se une y puede decidir cargar el historial.
- Envío de mensajes en tiempo real: Los mensajes se envían al servidor, que los almacena en MySQL y los reenvía a todos los conectados en el grupo, incluyendo a otros participante conectados desde el mismo PC.
- Persistencia e historial: Todos los mensajes se guardan en la BD. Al unirte, puedes optar por cargar el historial completo o empezar vacío.
- Búsqueda y filtro: Campo para filtrar mensajes por nombre de usuario en la vista local.
- Limpieza local: Botón para limpiar la vista del chat sin afectar la BD ni otros clientes.

Estas funcionalidades cumplen con los requisitos funcionales obligatorios del enunciado.

3. Arquitectura del sistema

Arquitectura cliente-servidor

El sistema sigue una arquitectura cliente-servidor distribuida, con servidores independientes por grupo, que permiten escalabilidad y aislamiento.

Servidor:

- Un proceso principal (ServidorApp) lanza servidores independientes para cada grupo, llamando a GestorGrupos, que utiliza ProcessBuilder. Los almacena como un HashMap que almacena el IdGrupo y el Proceso.
- Cada ServidorGrupo escucha en su propio puerto, acepta conexiones y crea un ManejadorCliente por usuario (hilo para concurrencia).
- ManejadorCliente Gestiona la conexión de un cliente al servidor de grupo y se encarga de autenticación, registro en el servicio de chat, recepción de mensajes y envío de mensajes al cliente.
- ServicioChat gestiona la sala de chat: transmisión de mensajes, persistencia con MensajeDAO y uso del patrón Observer (ChatObservable) para notificar a clientes conectados.
- Acceso a BD mediante JDBC, con DAOs para usuarios, grupos y mensajes.

Cliente:

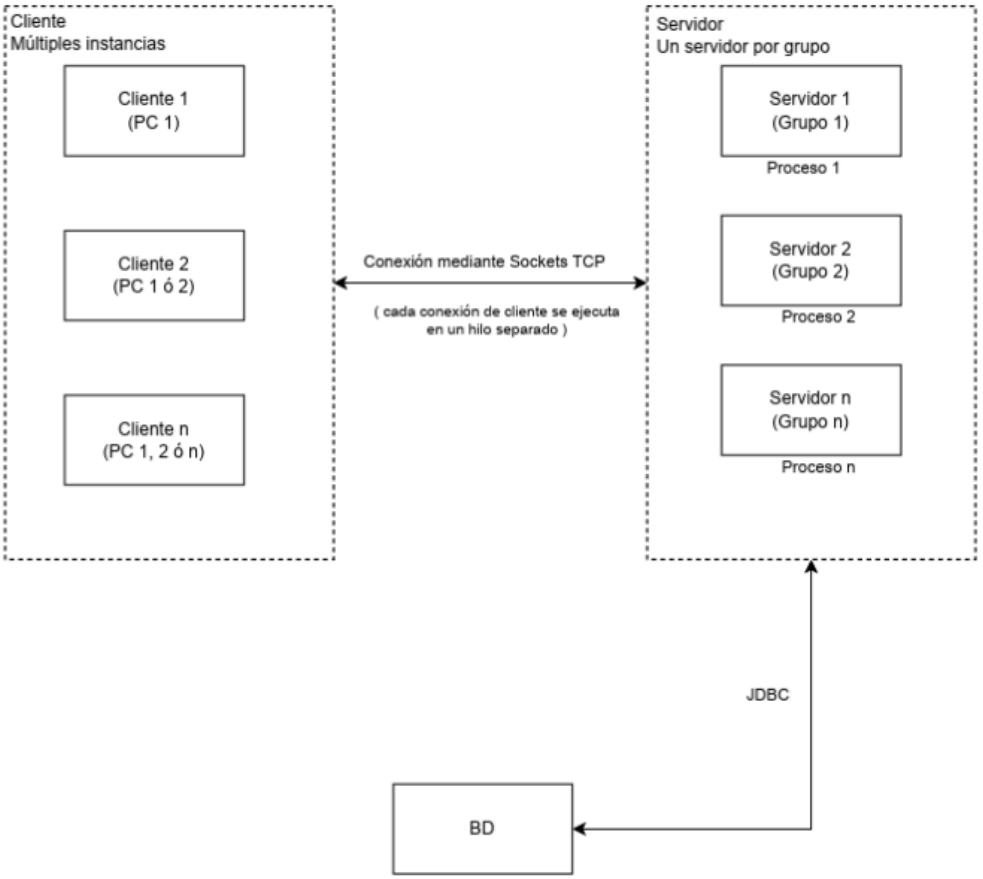
- Interfaz gráfica (Swing) con ventanas para login, registro, selección de grupo y chat.
- Controladores (LoginController, ChatController) separan la lógica de la vista.
- Conexión TCP mediante Cliente, con hilo MessageListener para recibir mensajes y notificar a la GUI mediante MessageObserver.

Gestión de concurrencia: Hilos en el servidor para múltiples clientes; en el cliente, hilo para escuchar mensajes.

Diagrama de clases

Se adjunta diagrama de clases completo como anexo al final de la memoria.

Diagrama de arquitectura general:



El diseño asegura separación de responsabilidades: modelo para entidades, persistencia para BD, servidor para lógica de chat, cliente para UI, controllers para intermediación.

4. Funcionamiento y comunicación

Funcionamiento del sistema

El flujo comienza con el servidor: ServidorApp consulta grupos en la BD y lanza un proceso ServidorGrupo por cada uno.

Al conectar un cliente:

- Login: Valida credenciales con LoginController.
- Selección de grupo: Obtiene puerto por clave con GrupoDAO, envía credenciales y opción de historial.
- Chat: ChatController abre socket, lanza listener, envía mensajes.

Mensajes en tiempo real: El cliente envía texto, el servidor lo guarda y notifica a los observadores.

Historial: Si se solicita, el servidor envía lista de mensajes con alias (JOIN en DAO).

Búsqueda: Filtro local en GUI por alias.

Limpieza: Borra JTextArea local.

Protocolo de comunicación

Uso de sockets TCP con texto plano delimitado por "|":

- Autenticación: alias\n password\n clave\n /historial true/false
- Mensaje: texto puro
- Broadcast: alias|fecha|texto
- Salida: /salir

El servidor interpreta y responde "OK" o "ERROR".

Explicación detallada del patrón Observer

El patrón Observer se implementa de forma explícita en el servidor y en el cliente para actualizar vistas en tiempo real.

En el servidor:

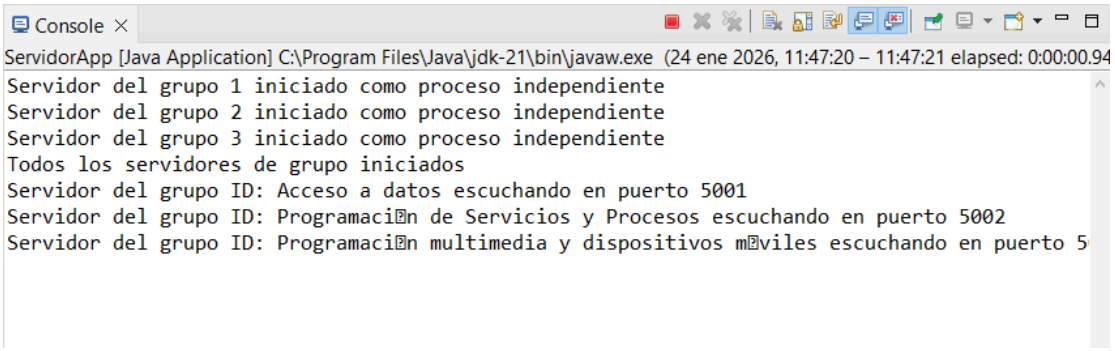
- ChatObservable es el sujeto que se observa (observable): mantiene una lista de observadores (ChatObserver).
- ManejadorCliente implementa ChatObserver (cada cliente es observador).
- Cuando se envía mensaje, ServicioChat llama a notificar() en observable, que invoca recibirMensaje() en cada manejador (difusión).

En cliente:

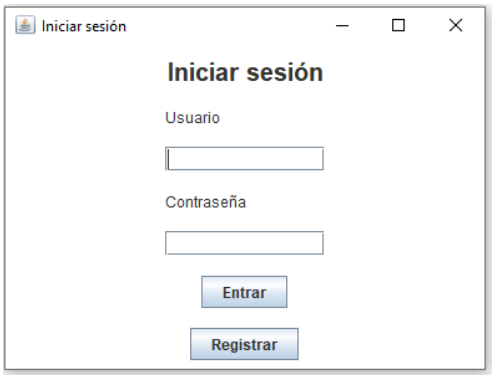
- MessageListener (hilo) es el sujeto que se observa (observable): recibe del socket y notifica a observadores (MessageObserver).
- GroupChat implementa MessageObserver: en onMessageReceived().

Capturas de pantalla

1. Al iniciar ServidorApp

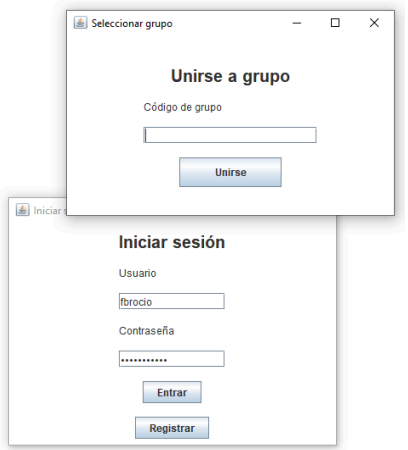


2. Al iniciar ClienteApp - Pantalla de login

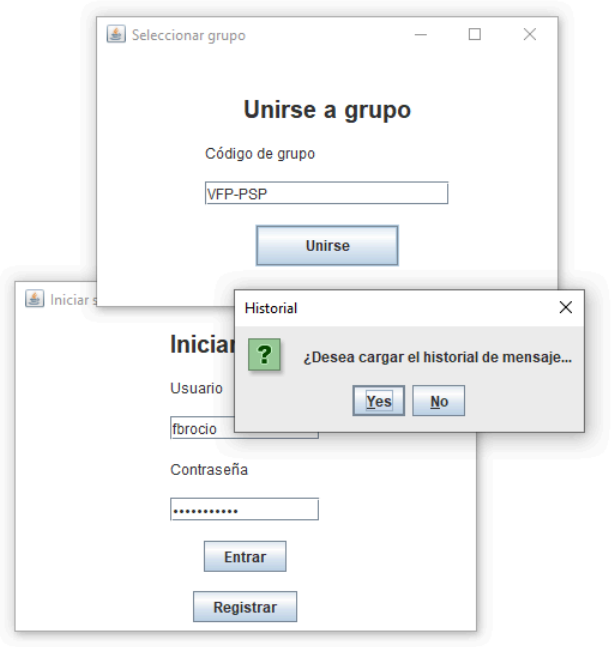


3. Selección de grupo

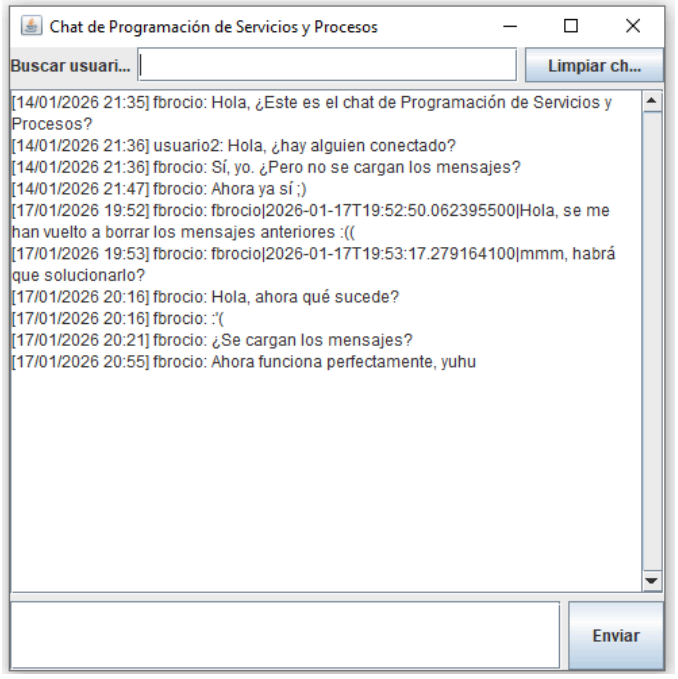
   Una vez introducidas las credenciales, si son correctas, se abre la ventana 'Seleccionar grupo'



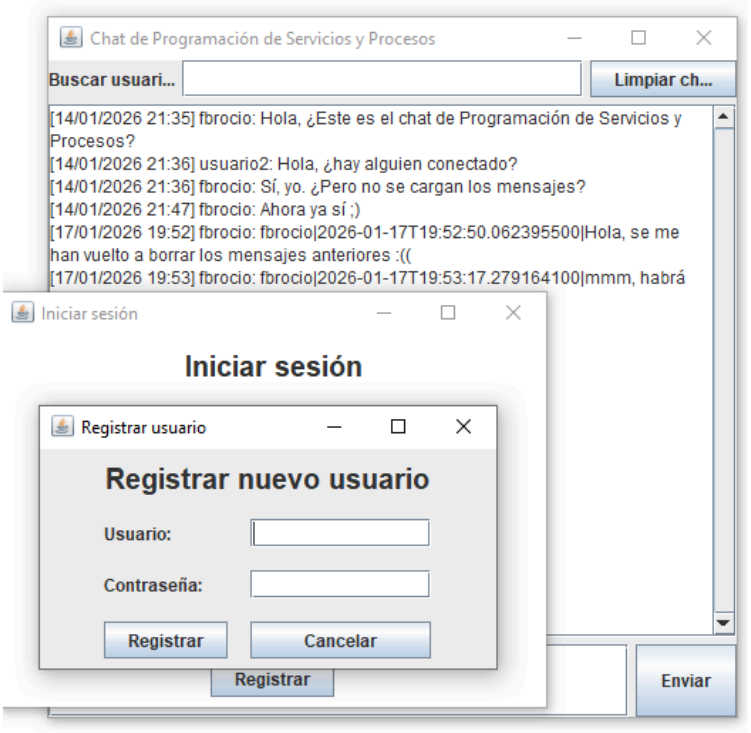
4. Una vez se introduce un código de grupo y se valida en la BD, se pregunta al usuario si desea carga el historial de mensajes

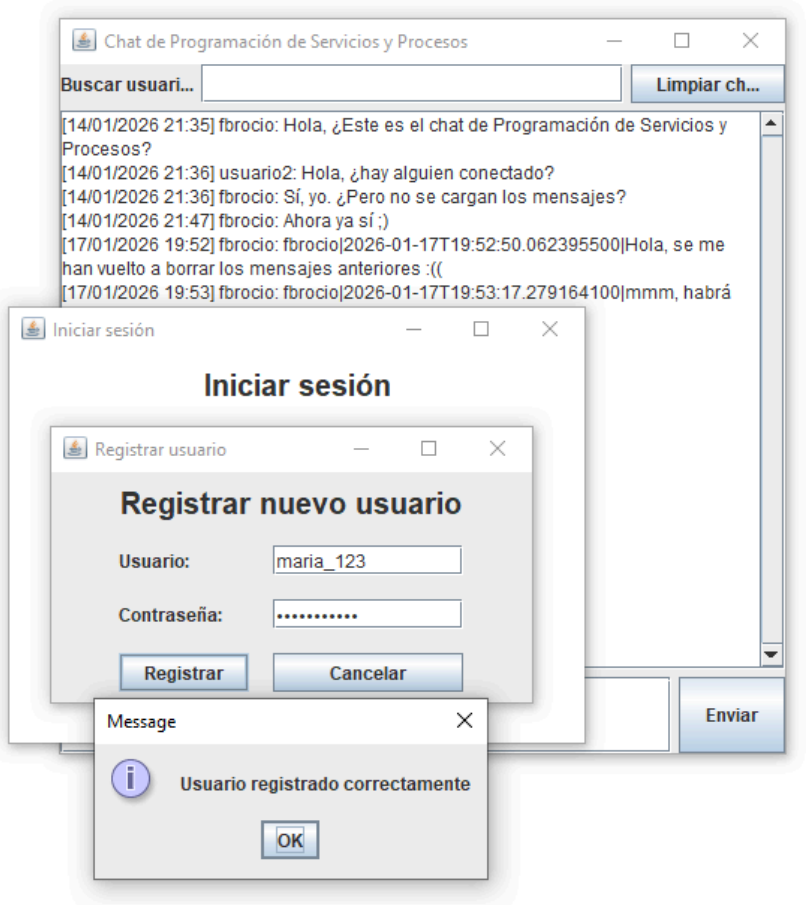


5. Si se selecciona que sí, se accede al chat indicado con los mensajes cargados

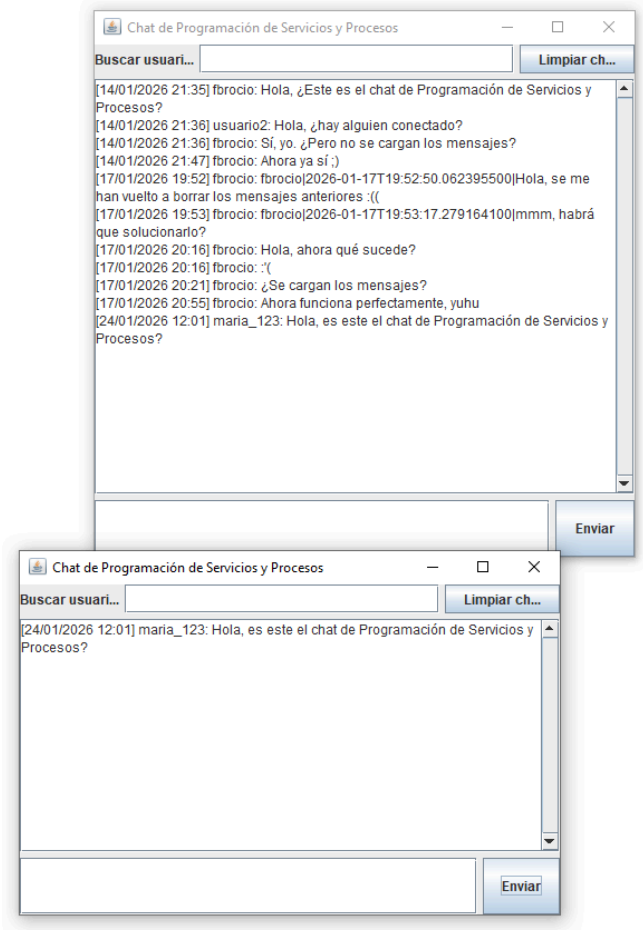


6. Si volvemos a iniciar ClienteApp, podemos iniciar sesión con otro usuario. En este caso, vamos a registrar un nuevo usuario

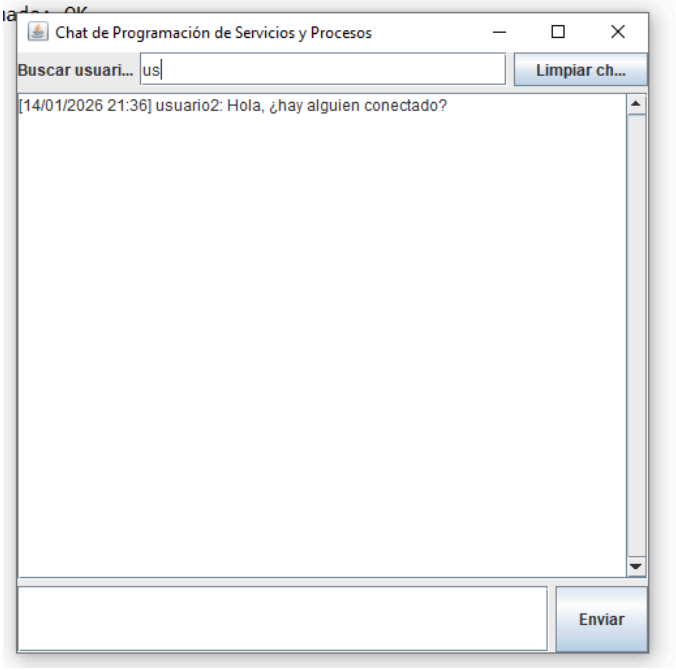




7. En este caso, con este nuevo usuario vamos a acceder al mismo chat pero sin cargar el historial



8. Búsqueda por filtro:



5. Dificultades y conclusiones

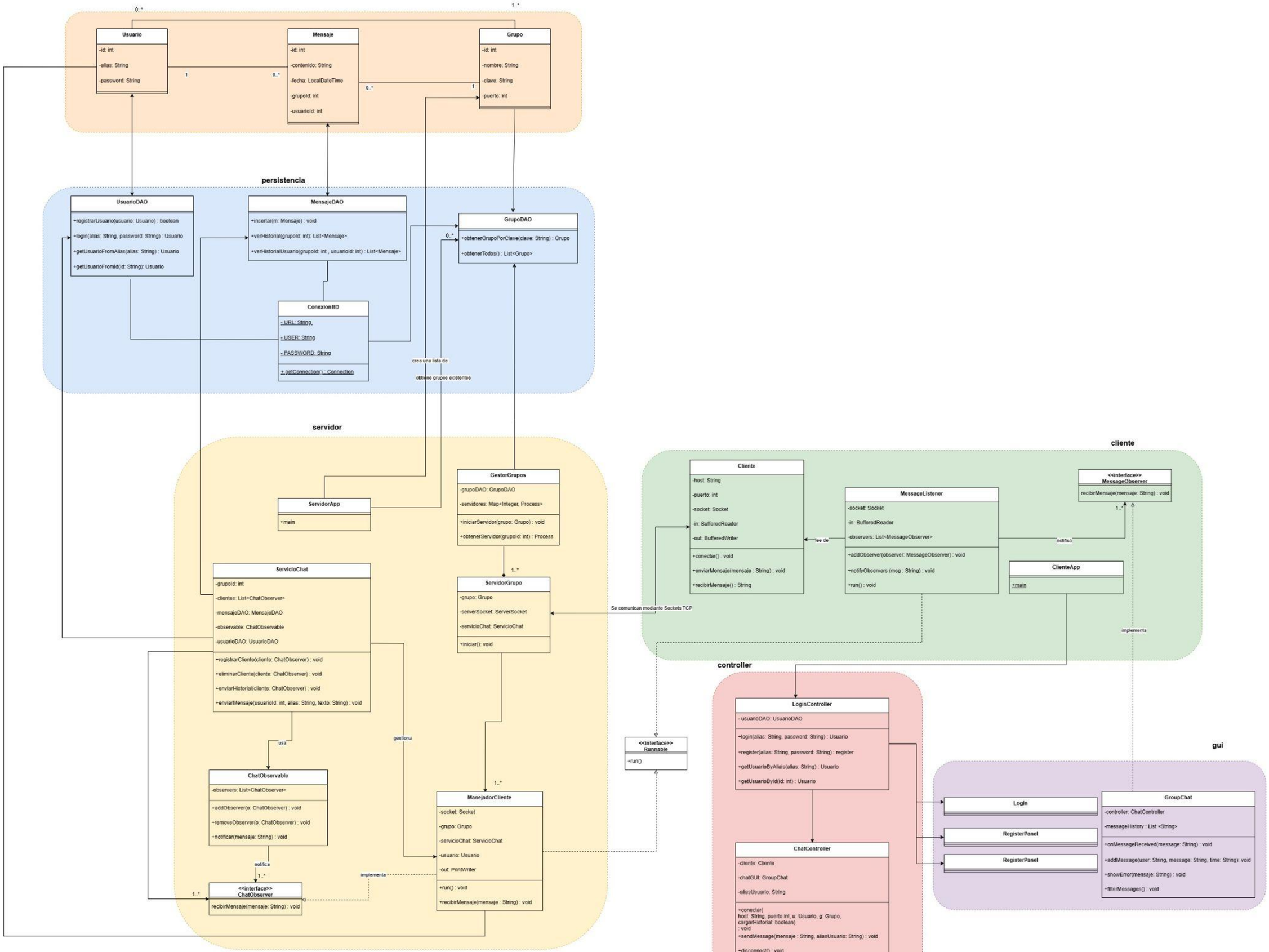
Dificultades encontradas

Durante el desarrollo, me encontré con varios retos que me ayudaron a entender y profundizar en los conceptos de la asignatura:

- Gestión de procesos independientes por grupo: Crear servidores en procesos separados con ProcessBuilder fue complejo, ya que tenía que manejar rutas de classpath y argumentos correctamente para que cada proceso accediera a la BD.
- Concurrencia con hilos: Asegurar que ManejadorCliente maneje múltiples usuarios sin condiciones de carrera, usando CopyOnWriteArrayList para clientes conectados.
- Implementación del patrón Observer: Entender cómo aplicarlo en el servidor para hacer 'broadcast' y en cliente para la GUI, diferenciando observadores en cada lado.
- UI gráfica con Swing: Añadir scroll automático, filtro por usuario y botón limpiar, mientras se integra con el hilo de escucha para actualizaciones en tiempo real.
- Validaciones y errores: Manejar desconexiones inesperadas y errores de BD sin crashear la app.

Conclusiones

El proyecto demuestra una implementación de un sistema de chat distribuido en Java, cumpliendo los requisitos: servidores independientes, concurrencia, persistencia en MySQL, interfaz gráfica usable y patrón Observer demostrable. Me ha permitido consolidar conocimientos en sockets TCP, hilos, JDBC y patrones de diseño, comprendiendo cómo se integra una arquitectura cliente-servidor compleja con persistencia y UI reactiva. Aunque podría mejorarse con encriptación de contraseñas o soporte para más usuarios concurrentes, el sistema es estable y funcional, representando un caso práctico valioso para la asignatura.


Usuario
-id: int
-alias: String
-password: String
Mensaje
-id: int
-contenido: String
-fecha: LocalDateTime
-grupoId: int
-usuarioId: int
Grupo
-id: int
-nombre: String
-clave: String
-puerto: int
0..*
1
0..*
0..*
1
1..*
persistencia
UsuarioDAO
+registrarUsuario(usuario: Usuario) : boolean
+login(alias: String, password: String) : Usuario
+getUsuarioFromAlias(alias: String) : Usuario
+getUsuarioFromId(id: String): Usuario
MensajeDAO
+insertar(m: Mensaje) : void
+verHistorial(grupoId: int): List<Mensaje>
+verHistorialUsuario(grupoId: int , usuarioId: int) : List<Mensaje>
GrupoDAO
+obtenerGrupoPorClave(clave: String) : Grupo
+obtenerTodos() : List<Grupo>
0..*
ConexionBD
- URL: String
- USER: String
- PASSWORD: String
+ getConnection() : Connection
crea una lista de
obtiene grupos existentes
servidor
ServidorApp
+main
GestorGrupos
-grupoDAO: GrupoDAO
-servidores: Map<Integer, Process>
+iniciarServidor(grupo: Grupo) : void
+obtenerServidor(grupoId: int) : Process
1..*
ServidorGrupo
-grupo: Grupo
-serverSocket: ServerSocket
-servicioChat: ServicioChat
+iniciar(): void
ServicioChat
-grupoId: int
-clientes: List<ChatObserver>
-mensajeDAO: MensajeDAO
-observable: ChatObservable
-usuarioDAO: UsuarioDAO
+registrarCliente(cliente: ChatObserver) : void
+eliminarCliente(cliente: ChatObserver) : void
+enviarHistorial(cliente: ChatObserver) : void
+enviarMensaje(usuarioId: int, alias: String, texto: String) : void
usa
gestiona
ChatObservable
-observers: List<ChatObserver>
+addObserver(o: ChatObserver) : void
+removeObserver(o: ChatObserver) : void
+notificar(mensaje: String) : void
notifica
1..*
1..*
<<interface>>
ChatObserver
recibirMensaje(mensaje: String) : void
implementa
1..*
ManejadorCliente
-socket: Socket
-grupo: Grupo
-servicioChat: ServicioChat
-usuario: Usuario
-out: PrintWriter
+run() : void
+recibirMensaje(mensaje : String) : void
<<interface>>
Runnable
+run()
Se comunican mediante Sockets TCP
cliente
Cliente
-host: String
-puerto: int
-socket: Socket
-in: BufferedReader
-out: BufferedWriter
+conectar() : void
+enviarMensaje(mensaje : String) : void
+recibirMensaje() : String
lee de
MessageListener
-socket: Socket
-in: BufferedReader
-observers: List<MessageObserver>
+addObserver(observer: MessageObserver) : void
+notifyObservers (msg : String) : void
+run() : void
notifica
<<interface>>
MessageObserver
recibirMensaje(mensaje: String) : void
1..*
ClienteApp
+main
implementa
controller
LoginController
- usuarioDAO: UsuarioDAO
+login(alias: String, password: String) : Usuario
+register(alias: String, password: String) : register
+getUsuarioByAlias(alias: String) : Usuario
+getUsuarioById(id: int) : Usuario
ChatController
-cliente: Cliente
-chatGUI: GroupChat
-aliasUsuario: String
+conectar(
host: String, puerto int, u: Usuario, g: Grupo,
cargarHistorial: boolean)
: void
+sendMessage(mensaje : String, aliasUsuario: String) : void
+disconnect() : void
+getUsuarioAlias() : String
gui
Login
RegisterPanel
RegisterPanel
GroupChat
-controller: ChatController
-messageHistory : List <String>
+onMessageReceived(message: String) : void
+addMessage(user: String, message: String, time: String): void
+showError(mensaje: String) : void
+filterMessages() : void