**Proyecto final**
**Documentación Técnica: Sistema de Chat Distribuido**

Jaider Andrés Melo Rodriguez
Juliana Andrea Bustamante Niño
Jhenier Alejandro Araujo Madroñero


Duvan Molina Marín

Universidad del
Quindío Facultad de
Ingeniería
Ingeniería de Sistemas y Computación
Sistemas de Información
Armenia, Quindío
2025

## Introducción

En este documento se presenta la descripción técnica de manera detallada del sistema de chat distribuido implementado en el lenguaje de programación **Elixir**, haciendo uso de la plataforma **Earlang/OTP.** Se propone como objetivo el facilitar la comunicación en tiempo real entre múltiples usuarios, para proporcionar diversas funcionalidades como salas de chat, un registro de los mensajes enviados, la autenticación de los usuarios y la tolerancia a los fallos.

El sistema se ha desarrollado como un proyecto académico en la Universidad del Quindío, sin embargo, su diseño funcional facilita su adaptación para ser productivo en ambientes corporativos.

## Arquitectura general

El sistema desarrollado se basa en una arquitectura **cliente-servidor** distribuida, que se encuentra compuesta por:

- **Nodo_servidor (servidor de chat):** se encarga de la centralización de la autenticación, la gestión y el almacenamiento de los historiales.
- **Nodo_cliente (clientes del chat):** estas son las aplicaciones que permiten que los usuarios se registren, realicen la autenticación, se unan a salas y puedan participar de charlas con otros usuarios en tiempo real de manera exitosa.
- **Procesos de la sala:** cada sala se rige por procesos separados que gestionan sus usuarios y mensajes.
- **Supervisor:** es el módulo encargado de vigilar al servidor y reiniciar los procesos fallidos, todo esto para garantizar disponibilidad.

La comunicación se lleva a cabo mediante el envío de mensajes entre nodos conectados y procesos.

## Descripción de módulos

### NodoServidor:

- **Archivo:** nodo_servidor.exs
- **Responsabilidades:**
    1. Crear salas y registrar los procesos que se lleven a cabo en estas.
    2. Autentificación de los usuarios enviados por clientes.
    3. Responder a los comandos ofrecidos como listar usuarios y búsqueda.
    4. Gestión del historial.

**Fragmento:**

```
{:crear_sala, pid, nombre_sala} - >
 pid_sala = Sala.iniciar(nombre_sala)
 send(pid, {:sala_creada, nombre_sala})
```

### NodoCliente:

- **Archivo:** nodo_cliente.exs
- **Responsabilidades:**
    1. Conectarse al servidor.

2. Autenticar al usuario y disponer de los comandos.
3. Mostrar el menú.

**Autenticación:**

```
usuario = Usuario.autenticar()
send({:servidor, servidor_node}, {:autenticacion, self(), usuario})
```

**Comando para enviar mensajes:**

```
send({:servidor, servidor_node}, {:mensaje_sala, sala_actual, usuario.nombre,
 mensaje})
```

### Sala

- **Archivo:** sala.ex
- **Responsabilidades:**
  1. Mantener a los usuarios conectados en la sala.
  2. Enviar los mensajes a todos los integrantes de la sala.
  3. Guardar los mensajes en el historial.

**Guardado de mensajes:**

```
File.write!("historial_\#{nombre_sala}.txt", mensaje <> "\n", [:append])
```

### Usuario

- **Archivo:** usuario.ex
- **Responsabilidades:**
  1. Realiza la autenticación por medio del usuario y contraseña.
  2. Registra los nuevos usuarios en el archivo usuarios.csv
  3. **Registro automático:**

```
registrar_usuario(nombre, contrasena)
```

### Util

- **Archivo:** util.ex
- **Responsabilidad:**
  1. Ingresar los datos del usuario, mostrar mensajes, encriptar y desencriptar mensajes.
  2. Validación para las entradas numéricas o texto.

### Supervisor

- **Archivo:** supervisor.exs
- **Responsabilidades:**
  1. Supervisa al servidor en todo momento.
  2. Hace el reinicio automático si el servidor "muere".

**Código de reinicio:**

```
{:EXIT, _pid, motivo} - >
    iniciar_servidor()
```

## Seguridad

- **Autenticación:** se realiza mediante el nombre y la contraseña.
- **Almacenamiento seguro:** este se guarda en **usuarios.csv**

- **Control de acceso:** únicamente los usuarios que se encuentren autenticados pueden crear las salas y enviar mensajes.

**Estructura usuario:**

```
%Usuario{nombre: nombre, contrasena: contrasena, pid: self()}
```

## Persistencia de datos

- **Usuarios:** los usuarios se almacenan en **usuarios.csv**. Cada línea sigue el formato de: **nombre,contraseña.**
- **Mensajes:** se guardan por sala en archivos **historial:<nombre_sala>.txt.**
- **Log del servidor:** en **log.txt,** aquí se registran los eventos importantes tales como salas creadas, usuarios unidos a una sala, desconexión inesperada de un usuario, etc.

**Ejemplo del historial:**

```
si > historial_si.txt
  1   [11:55 p.m.] andres: krodddddd
  2   [11:56 p.m.] juli: krollllllllll
```

Como se puede observar, los mensajes se guardan de manera encriptada en el historial para mayor seguridad, sin embargo cuando el usuario utiliza comandos como **/search** o **/history** dichos mensajes se muestran desencriptados al usuario para que pueda leerlos con facilidad.

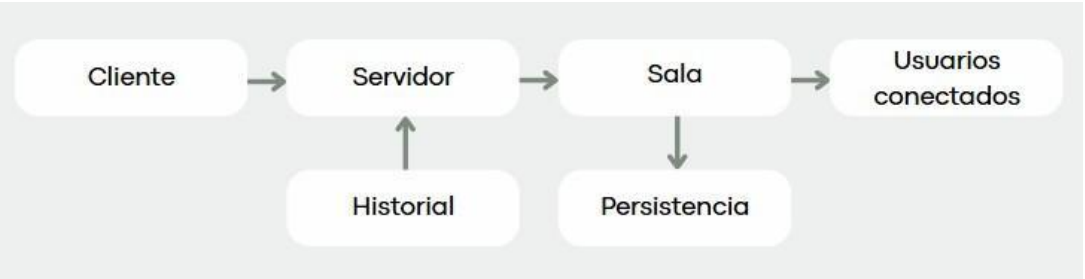
```
[si] > /history

Historial de mensajes:
  - [11:55 p.m.] andres: holaaaaa
  - [11:56 p.m.] juli: holiiiiiiiiii
```

## Flujo completo de uso

1. El **cliente** conecta al servidor con **Node.connect.**
2. El usuario es **autenticado.** Si este no existe, el sistema procede a registrarlo.
3. El usuario dispone de las siguientes opciones dependiendo de si está o no en una sala:
   - Crear una sala **/create <nombre>**
   - Unirse a una sala **/join <nombre>**
   - Enviar un mensaje **/msg <mensaje>**
   - Ver el historial **/history**
   - Buscar mensajes específicos **/search <palabra>**
   - Listar los usuarios **/list**
   - Mostrar menú de comandos **/help**
   - Salir de una sala **/exit**
   - Cerrar sesión **/close**

**Diagrama:**



## Ejecución

**Requerimientos:**

- Elixir >= 1.14
- Earlang/OTP >= 25

**Pasos:**

1. Poner a flote el servidor:

```
elixir –sname servidor –cookie my_cookie nodo_servidor.exs
```

2. Iniciar clientes:

```
elixir –sname cliente –cookie my_cookie nodo_cliente.exs
```

3. Supervisión automática:

```
elixirc supervisor.exs
```

Ejemplo de conexión en código:

```
servidor_node = :"servidor@192.168.100.91"
Node.connect(servidor_node)
```

## Escalabilidad y distribución

- **Distribución horizontal:** se agregan más nodos servidor para lograr balanceo de carga.
- **Procesos concurrentes:** cada sala tiene un único proceso.
- **Alta disponibilidad:** el supervisor hace el reinicio del servidor en caso de un fallo.

## Archivos clave en el proyecto

A continuación se presentan los archivos fundamentales para el sistema:

- **nodo_servidor.exs:** lógica del servidor.
- **nodo_cliente.exs:** lógica de los clientes.
- **sala.ex:** módulo para la gestión de las salas.
- **usuario.ex:** registro y autenticación.
- **util.ex:** son funciones auxiliares.
- **supervisor.exs:** módulo para la tolerancia a fallos.
- **usuarios.csv:** base de datos de los usuarios.
- **historial_<nombre_sala>.txt:** mensajes por cada sala.
- **log.txt:** log del sistema.

## Conclusión

El proyecto *Sistema de Chat Distribuido* es un ejemplo ideal para la demostración de cómo hacer uso de **Elixir** y **Erlang/OTP** para la creación de una aplicación distribuida, tolerante a los fallos y concurrente. Gracias a la arquitectura que ofrece, permite la adaptación a entornos de una mayor demanda.

## Créditos

Este sistema fue desarrollado como proyecto final académico en la Universidad del Quindío, Facultad de Ingeniería. Tiene como objetivo explorar la programación distribuida y concurrente usando el lenguaje de programación Elixir, aplicándolo en un entorno de mensajería.