Informe de Diseño del Proyecto

Claudia Hernández Pérez Joel Aparicio Tamayo

Noviembre, 2025

Resumen

Telegraph es una aplicación de mensajería distribuida con una arquitectura peer-to-peer en la que cualquier cliente puede recibir y mandar mensajes con la garantía de que solo lo podrán ver los involucrados en la conversación. Existen dos entidades fundamentales: cliente y gestor de entidades. Los clientes son el sistema en sí y la comunicación entre estos es la piedra angular de la tarea, mientras que los gestores de identidades son los encargados de manejar la información de los usuarios.

1. Arquitectura

Los sistemas distribuidos modernos han evolucionado a través de diversos modelos arquitectónicos que buscan optimizar la escalabilidad, tolerancia a fallos y eficiencia en la comunicación. Entre estos modelos, las arquitecturas *peer-to-peer* (P2P) han demostrado ser particularmente efectivas para aplicaciones que requieren descentralización y colaboración directa entre nodos.

Las arquitecturas P2P se caracterizan por la ausencia de roles fijos entre clientes y servidores, donde todos los nodos participan activamente tanto en la provisión como en el consumo de servicios. Existen principalmente dos variantes: los sistemas **P2P estructurados**, que organizan los nodos en topologías deterministas como anillos o DHTs para permitir búsquedas eficientes; y los sistemas **P2P no estructurados**, donde las conexiones entre nodos se establecen de forma ad-hoc (no siguen un plan preestablecido).

1.1. Organización del sistema distribuido

Telegraph basa su funcionamiento en una arquitectura *peer-to-peer* no estructurada con un funcionamiento inspirado en *BitTorrent*: los clientes, para enviar mensajes, consultan a los gestores de identidades para obtener la `IP` del destinatario y establecer comunicación directa con él, lo cual es un comportamiento similar a la solicitud de `peers` a los `trackers`, para intercambiar `chunks` entre ellos.

1.2. Roles existentes en el sistema

En el sistema existen dos roles principales: clientes (*clients*) y gestores de identidades (*identity managers*).:

1. **gestor de identidades**: es el responsable de gestionar toda la información de las cuentas, tal como:

- Nombre de usuario
- Contraseña encriptada
- Dirección `IP` y puerto
- Estado de conexión (en línea ó desconectado)
- Última vez visto en línea

2. **clientes**: pueden comunicarse entre sí directamente sin un servidor centralizado que almacene mensajes temporalmente. Cada cliente puede realizar las siguientes acciones:

   - Enviar mensajes
   - Recibir mensajes
   - Notificar que un mensaje ha sido leído
   - Reintentar el envío de mensajes pendientes a usuarios fuera de línea

1.3. Distribución de servicios en ambas redes de Docker

La implementación del sistema distribuido se despliega utilizando *Docker Swarm* con una red `overlay attachable` que permite la comunicación transparente entre contenedores ejecutándose en diferentes hosts físicos. Esta configuración replica un entorno de producción real donde los servicios están distribuidos geográficamente.

- **Balanceo de Clientes**: Los clientes **P2P** pueden desplegarse en ambos hosts (tantos como se deseen), simulando usuarios en diferentes ubicaciones de red. Lo que se recomienda es balancear entre ambos hosts físicos para sobrecargar lo menos posible cada uno.

- **Distribución de Gestores**: Inicialmente el sistema contará con $4k - 2$ réplicas de gestores (siendo k la tolerancia a fallas esperada), donde cada host contará con $2k - 1$ gestores. De esta forma se garantiza que si se pierde la conexión entre ambos hosts físicos, en cada red independiente se logre una tolerancia a fallas $k - 1$. Esta desición puede estra sujeta a cambios futuros.

- **Descubrimiento de Servicios**: *Docker Swarm* proporciona un **DNS** interno que permite a los contenedores descubrir automáticamente los servicios desplegados en la red `overlay`. Además, se cuenta con una alternativa ante posibles fallos del **DNS** que se explicará en otra sección más adelante.

2. Procesos

Los sistemas distribuidos están constituidos por procesos que representan las unidades fundamentales de ejecución, responsables de llevar a cabo las diversas tareas y servicios que componen el sistema. Estos procesos, distribuidos en múltiples nodos de red, cooperan y se comunican para ofrecer funcionalidades integradas y transparentes al usuario final. La correcta organización y coordinación entre procesos determina en gran medida la escalabilidad, eficiencia y confiabilidad del sistema distribuido. En el contexto de *Telegraph*, el sistema se estructura alrededor de dos tipos principales de procesos: los procesos del cliente y los del gestor.

2.1. Tipos de procesos dentro del sistema

El sistema implementa una arquitectura de procesos especializados que se dividen en dos categorías principales:

- **Procesos del gestor**:
  - **Servidor *Flask***: Proceso que expone `API REST` para autenticación, registro, resolución de direcciones `IP`, entre otras solicitudes.
  - ***Worker* de verificación**: Proceso periódico con APScheduler que monitorea los usuarios inactivos del sistema y actualiza su estado a *offline*.
  - **Servidor `UDP`**: Proceso que mantiene un `socket UDP` abierto para descubrimiento alternativo de gestores.
- **Procesos del cliente**:
  - **Interfaz Streamlit**: Proceso que renderiza la interfaz gráfica y gestiona la interacción del usuario.
  - **Servidor *Flask***: Proceso que expone `API REST` para comunicación entre clientes.
  - ***Worker* para tareas pendientes**: Tareas programadas con APScheduler para gestionar mensajes o actualizaciones de estado pendientes.
  - **Servicio de mensajería**: Componente que establece conexiones directas con otros clientes para envío de mensajes.
  - **Servicio para comunicación con gestores**: Componente encargado del descubrimiento inicial de gestores, así como de todas las solicitudes que les envían los clientes.

2.2. Organización o agrupación de los procesos en el sistema

Los procesos del sistema se agrupan en las dos categorías ya conocidas. Cada una posee un proceso principal y el resto ejecutan tareas en segundo plano:

- **Organización de procesos del gestor**:
  - **Proceso principal**: Servidor *Flask*
  - **Procesos independientes en segundo plano**: *Worker* de verificación y servidor `UDP`.
- **Organización de procesos del cliente**:
  - **Proceso principal**: Interfaz de *Streamlit*. Cada vez que se renderiza, se emplea el componente de comunicación con gestores para enviar señales de vida y obtener el listado de clientes activos. Con las interacciones del usuario se utiliza, además, el servicio de mensajería para comunicarse.

- **Procesos independientes en segundo plano**: *Worker* para tareas pendientes y servidor *Flask*. Aunque este último no se comunica directamente con el servicio de mensajería ni con el *Worker* dentro del propio cliente, sí son componentes relacionados, ya que los mensajes llegan de emisor a receptor por solicitudes `HTTP` que recaen sobre dicho servidor. Igualmente, los mensajes y notificaciones de lectura pendientes que se reintentan llegan a los *end-points* de *Flask* en el receptor.

2.3. Tipo de patrón de diseño con respecto al desempeño

El sistema implementa **concurrencia basada en hilos** bajo el modelo **Many-to-One**:

- **Hilos especializados**: Diferentes componentes ejecutan en hilos separados dentro del mismo proceso:
  - **cliente**: El proceso principal es el que ejecuta la interfaz gráfica de *Streamlit*. De él se desprenden dos hilos: uno para el *APScheduler* encargado de las tareas pendientes y otro para el servidor de *Flask*. El resto de tareas se ejecutan en el proceso principal.
  - **gestor**: El proceso principal es el que ejecuta el servidor de *Flask*. De él se desprenden dos hilos: uno para el *APScheduler* encargado de detectar usuarios inactivos y otro para el servidor `UDP`.
- **Hilo por solicitud**: Los servidores *Flask* con `threaded=True` implementan este patrón para manejar múltiples solicitudes `HTTP` concurrentemente.

3. Comunicación

La comunicación es uno de los pilares fundamentales en los sistemas distribuidos, permitiendo la interacción y coordinación entre componentes dispersos en diferentes nodos de red. En este tipo de sistemas, los procesos deben intercambiar información de manera confiable y eficiente, a pesar de las limitaciones inherentes de la red como latencia, pérdida de paquetes y fallos parciales. La elección adecuada de los mecanismos de comunicación determina en gran medida el desempeño, escalabilidad y confiabilidad del sistema como un todo.

3.1. Tipos de comunicación

Telegraph utiliza dos tipos de comunicación fundamentales:

- `API REST`: Utilizado como mecanismo principal para la comunicación. Permite ejecutar acciones sobre la base de datos a partir de solicitudes.
- `Sockets UDP`: Implementado como mecanismo de descubrimiento de servicios alternativo.

3.2. Comunicación cliente-servidor

La comunicación cliente-servidor se basa en el patrón **solicitud-respuesta** (*request-reply*) donde el cliente envía un mensaje de solicitud al servidor (gestor u otro cliente), que recibe y procesa la solicitud, devolviendo finalmente un mensaje de respuesta. En Tabla 1 se pueden apreciar ejemplos de la implementación.

Tipo	**Solicitud del Cliente**	**Respuesta del Servidor**
API REST	`/login`, method=[POST], content={username, password, ip, port}	`CG` {"message": "Login exitoso", "status": 200}
	`/users/<username>`, method=[GET]	`CG` {"message": {...}, "status": 200}
	`/heartbeat`, method=[POST], content={username}	`CG` {"message": "heartbeat recibido", "status": 200}
	`/users/reconnect/<ip>` `/<username>`, method=[PUT]	`CG` {"message": 'IP actualizada exitosamente', "status": 200}
	`/notify_read`, method=[POST], content={from, to}	`CC` {"marked": count}, 200
	POST `/receive_message`, method=[POST], content={from, to, text}	`CC` {"status": 'ok'}, 200
Socket UDP	{'action': 'discover'}	{'status': 'active'}

Tabla 1: Comunicación cliente-servidor. `CC` denota la comunicación entre clientes (uno actuando como servidor) y `CG` la comunicación del cliente con el gestor

3.3. Comunicación servidor-servidor

...

3.4. Comunicación entre procesos

En el sistema, cada nodo (gestor o cliente) ejecuta un proceso principal y varios procesos independientes en segundo plano. Como todos los componentes se ejecutan como hilos dentro del mismo espacio de memoria, la interacción entre ellos se realiza mediante acceso directo a objetos compartidos, por lo que sí se maneja coordinación, sin embargo, **no existe comunicación entre procesos en el mismo nodo**,

4. Coordinación

La coordinación se realiza principalmente mediante: - Estados con sello temporal: los usuarios registran `last_seen` en `AuthService.update_last_seen` y el identity manager

utiliza ese campo para decidir inactividad. - Heartbeats: los clientes envían latidos periódicos con `ApiHandlerService.send_heart_beat` y el manager actualiza `last_seen`. - Jobs de mantenimiento: `check_inactive_users` en el identity manager marca usuarios offline y fuerza desconexiones cuando procede.

5. Nombrado y localización

Los recursos (usuarios) se nombran por `username`. La localización se resuelve consultando el identity manager mediante `/users/<username>` y obteniendo `ip` y `port`. El cliente también mantiene un pequeño repositorio local (`ClientRepository`) que guarda el nombre de usuario en `.env` dentro del directorio de datos.

6. Consistencia y replicación

La implementación actual no incorpora replicación distribuida ni un anillo Chord. En su lugar se aplica una persistencia local en cada cliente (archivos JSON) y un registro centralizado de usuarios en el identity manager (fichero JSON `/data/users.json`).

Consecuencia: no hay tolerancia a particiones en lo que respecta al registro de usuarios; si el manager que contiene el fichero de usuarios deja de estar disponible, los clientes solo pueden actuar con la información cacheada localmente o hasta que descubran otro manager.

7. Tolerancia a fallos

Medidas actuales: - Descubrimiento múltiple: `ApiHandlerService` intenta localizar managers vía DNS (resolviendo `identity-manager`) y, si falla, hace broadcast sobre la red overlay para localizar managers activos; mantiene una lista `api_urls` y reintenta peticiones a diferentes managers. - Reintentos y marcadores locales: `MessageService` marca mensajes como `pending` si no puede contactar al receptor, y reintenta su envío periódicamente desde las tareas en background. - Detección de inactividad: el identity manager marca usuarios como offline si no reciben heartbeats y fuerza desconexiones.

Limitaciones: - No existe replicación de la base de datos de usuarios ni de los mensajes a otros gestores; la tolerancia a fallos está limitada a la disponibilidad de los ficheros JSON del manager y de los propios clientes.

8. Seguridad

8.1. Autenticación y contraseñas

Las contraseñas de los usuarios se hashean con bcrypt (`AuthService.hash_password`) antes de almacenarlas en el repositorio de usuarios (`UserRepository` guarda datos en JSON en `/data/users.json`). La verificación se realiza con `checkpw`.

8.2. Canales y cifrado

En la versión actual no hay cifrado de transporte (no hay TLS configurado en las peticiones HTTP internas). Esto es una área prioritaria de mejora: se recomienda habilitar HTTPS entre managers y clientes y/o establecer túneles VPN / mTLS en despliegues reales.

8.3. Consideraciones adicionales

- Validación básica de payloads en endpoints Flask. - No hay control de acceso granular ni lista blanca de managers; cualquiera que responda al discovery UDP puede ser usado por clientes si le han descubierto.

9. Conclusiones y trabajo futuro

La implementación actual es una prueba de concepto funcional que cubre: - Registro, login y localización de usuarios mediante un *identity manager* ligero. - Intercambio directo de mensajes entre clientes mediante HTTP y persistencia local de mensajes. - Mecanismos básicos de disponibilidad (heartbeat, detección de inactividad, reintentos de envío).

Mejoras recomendadas: - Implementar replicación y tolerancia a fallos entre gestores (por ejemplo, un anillo Chord o replicación maestro-esclavo para `users.json`). - Asegurar canales con TLS/mTLS y proteger el UDP discovery. - Añadir pruebas automáticas y scripts de despliegue reproducible.

Apéndice: Referencias a código

Se listan los ficheros más relevantes y su propósito: - `src/client/app/main.py`: UI Streamlit y arranque de servicios. - `src/client/app/server.py`: servidor Flask del cliente (endpoints para recibir mensajes y recibos). - `src/client/app/background_tasks.py`: loop de sincronización y reintentos. - `src/client/app/services/`: lógica de negocio cliente (API handler, client info, message service). - `src/client/app/repositories/`: persistencia local (mensajes, usuario local). - `src/identity-manager/app/api.py`: server REST del manager. - `src/identity-manager/app/udp_discovery.py`: discovery por UDP. - `src/identity-manager/app/services/auth_service.py`: autenticación y gestión de usuarios.