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.

- **Red Overlay**: Se crea una red Docker Swarm de tipo overlay que conecta ambos hosts físicos, permitiendo que los contenedores se comuniquen como si estuvieran en la misma red local.
- **Distribución de Gestores**: Los $2K + 1$ nodos del cluster Raft se distribuyen estratégicamente entre ambos hosts físicos para garantizar tolerancia a fallos incluso ante la caída de un host completo.
- **Balanceo de Clientes**: Los clientes P2P se despliegan de forma balanceada entre ambos hosts, simulando usuarios en diferentes ubicaciones de red.
- **Descubrimiento de Servicios**: Docker Swarm proporciona DNS interno que permite a los contenedores descubrir automáticamente los servicios desplegados en la red overlay.

2. Procesos y servicios

2.1. Procesos en el cliente

- Interfaz principal: `main.py` arranca la UI en Streamlit y lanza un hilo para el servidor Flask (`start_flask_server`). - Servidor Flask embebido: `server.py` expone endpoints HTTP para recibir mensajes (`/receive_message`), notificación de lectura (`/notify_read`) y desconexión (`/disconnect`). - Tareas en segundo plano: la función `background_tasks` (`background_tasks.py`) es programada periódicamente con APScheduler y realiza: - Reintento de recibos no sincronizados (`MessageService.retry_unsynchronized_rece`

- Consulta de usuarios online (`ApiHandlerService.get_online_users`). - Actualización de la IP local (`ApiHandlerService.update_ip_address`). - Envío de mensajes pendientes (`MessageService.send_pending_mssgs`).

2.2. Procesos en el identity manager

- Servidor REST (Flask): rutas para `/register`, `/login`, `/logout`, `/peers`, `/users`, `/heartbeat` y endpoints de utilidad para reconectar IPs. - UDP discovery: `udp_discovery.run_server` escucha en el puerto DNS (por defecto 5353) y responde a peticiones de descubrimiento para permitir que clientes encuentren managers en la red overlay. - Job de limpieza: un job periódicamente marca como `offline` a usuarios inactivos (en `api.py` con APScheduler) y notifica a clientes para desconectar.

3. Comunicación

3.1. Protocolos utilizados

La comunicación entre componentes utiliza HTTP(S) sobre la red overlay (en la implementación actual se usan peticiones HTTP simples con la librería `requests`). Para descubrimiento en la red local se usa UDP broadcast en `udp_discovery.py`.

3.2. Patrones de interacción

- Cliente <->Identity Manager: peticiones REST (REQ-REP) para registrar, autenticar, consultar usuarios y enviar heartbeat (implementado en `ApiHandlerService`). - Cliente <->Cliente: los clientes exponen un servidor HTTP (Flask) para recibir mensajes y recibos. El remitente realiza una petición POST a `/receive_message` del receptor. - Descubrimiento: UDP broadcast/response para detectar managers disponibles.

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.