

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 (tanto 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 decisión puede estar 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 **conurrencia 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
Socket UDP	POST /receive_message, method=[POST], content={from, to, text}	CC {"status": 'ok'}, 200
	{'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 en sistemas distribuidos aborda el desafío de lograr que múltiples componentes independientes trabajen de manera consistente y organizada. En un entorno donde los procesos ejecutan concurrentemente en diferentes nodos, es esencial establecer

mecanismos que sincronicen acciones, gestionen el acceso a recursos compartidos y permitan la toma de decisiones colectivas. Esta sección detalla las estrategias de coordinación implementadas en el sistema de mensajería distribuida.

4.1. Sincronización de acciones

El sistema implementa varios mecanismos de sincronización para coordinar acciones entre componentes distribuidos:

- **Heartbeats y Monitoreo de Estado:** Los clientes envían periódicamente mensajes de heartbeat a los gestores, permitiendo sincronizar el estado de conexión de todos los usuarios en el sistema.
- **Sincronización de Confirmaciones de Lectura:** Mediante el endpoint `/notify_read`, el sistema sincroniza el estado de lectura de mensajes entre emisor y receptor, garantizando que ambos tengan una visión consistente.
- **Actualización Coordinada de Estado:** Cuando un gestor detecta que un usuario ha excedido el timeout de inactividad, notifica al cliente mediante `POST /disconnect` para sincronizar el estado en ambos extremos.
- **Replicación con Raft:** Los gestores utilizan el algoritmo de consenso Raft para sincronizar operaciones críticas, asegurando que todos los nodos repliquen las mismas acciones en el mismo orden.

4.2. Acceso exclusivo a recursos. Condiciones de carrera

Para prevenir condiciones de carrera y garantizar el acceso exclusivo a recursos compartidos, el sistema implementa:

- **Bloqueos en Base de Datos:** Los gestores utilizan mecanismos de locking a nivel de base de datos para garantizar que operaciones concurrentes sobre el mismo usuario (login, logout, actualización de estado) se ejecuten atómicamente.
- **Sincronización en Memoria Compartida:** Dentro de cada proceso (gestor o cliente), los múltiples hilos coordinan el acceso a estructuras de datos compartidas mediante locks y mutex, previniendo condiciones de carrera en:
 - Acceso a la lista de usuarios conectados
 - Procesamiento de mensajes entrantes
 - Actualización de estados de mensajes
- **Exclusión Mutua Distribuida:** En la implementación de Raft, solo el líder puede procesar operaciones de escritura, actuando como un mecanismo de exclusión mutua distribuido para operaciones críticas.
- **Serialización de Operaciones:** El sistema garantiza que operaciones conflictivas sobre el mismo recurso (ej: múltiples login del mismo usuario) se serialicen correctamente.

4.3. Toma de decisiones distribuidas

El sistema emplea varios mecanismos para la toma de decisiones en un entorno distribuido:

- **Algoritmo de Consenso Raft:** Los gestores utilizan Raft para tomar decisiones colectivas sobre:
 - Elección del líder que coordinará las operaciones
 - Compromiso de operaciones en el log distribuido
 - Gestión de membresía del cluster
- **Descubrimiento de Servicios:** Los clientes implementan un mecanismo de votación implícita mediante UDP broadcast, donde múltiples gestores pueden responder y el cliente selecciona el más adecuado.
- **Decisión de Rutas de Mensajería:** El sistema decide automáticamente la ruta óptima para entregar mensajes, consultando a los gestores para obtener la ubicación actual de los destinatarios.
- **Resolución de Conflictos de Estado:** Cuando se detectan estados inconsistentes (ej: usuario marcado como offline pero enviando mensajes), el sistema toma decisiones basadas en timestamps y mecanismos de heartbeat para resolver la inconsistencia.

Estos mecanismos de coordinación permiten que el sistema distribuido funcione como una entidad cohesiva, manteniendo la consistencia y confiabilidad a pesar de la naturaleza descentralizada de sus componentes.

5. Nombrado y Localización

La identificación y localización de recursos constituye uno de los desafíos fundamentales en el diseño de sistemas distribuidos. En un entorno donde los componentes están físicamente dispersos a través de múltiples nodos de red, es esencial contar con mecanismos que permitan a los procesos encontrar y acceder de manera eficiente a los servicios y datos que necesitan para operar.

5.1. Identificación de los datos y servicios

El sistema almacena dos tipos de información:

- **Usuarios:** Se identifican por el campo `username` único para cada usuario.

```
1 {  
2   "username": "alice",  
3   "password": "$2b$12$LQv3c1yqBWVHxkd0g8f7QuYKThbMvOoiuTM.ZlbzOnY9qU8pMq.LK",  
4   "ip": "192.168.1.150",  
5   "port": 8000,  
6   "status": "online", /* puede ser "offline" tambien */  
7   "last_seen": "2025-11-15T14:30:25.123456Z"  
8 }
```

- **Mensajes:** Se identifican por la llave (`from`, `to`, `timestamp`) única por mensaje.

```

1  {
2    "from": "alice",
3    "to": "bob",
4    "text": "Aprobamos Distribuidos",
5    "timestamp": "2025-11-15T16:45:30.789123Z",
6    "read": false,
7    "status": "ok" /* puede ser "pending" tambien */
8 }
```

Además de datos, es posible identificar nodos y servicios:

- **Gestores:** Todos poseen el mismo alias en la red: `identity-manager`; aunque cada uno tiene su propia dirección IP.
- **Clientes:** Guardan su dirección IP en los datos del usuario conectado desde ese nodo.
- **Servicios REST:** Se identifican por una `url`: `Node_IP:Puerto/endpoint`. Cada funcionalidad se expone mediante `endpoints` específicos como `/login`, `/register`, `/peers`, `/receive_message`, etc.

5.2. Ubicación de los datos y servicios

Los datos del sistema se almacenan en archivos `.json` distribuidos de la siguiente manera:

- **Datos de Usuarios:** Almacenados localmente en cada gestor.
- **Mensajes:** Almacenados localmente en los clientes que intervienen en la conversación.

Los servicios REST se distribuyen de la siguiente forma:

- **Servicios de Gestores:** Desplegados con puerto fijo 8000.
- **Servicios de Clientes:** Desplegados en un puerto dinámico, por defecto: 8000.

5.3. Localización de los datos y servicios

Los nodos del sistema implementan dos mecanismos de descubrimiento de gestores, evitando depender de un único servicio que, eventualmente, podría fallar.

- **Descubrimiento Primario:** Todos los gestores tienen el mismo alias, que se utiliza para solicitar al **DNS** de *Docker* todas las direcciones IP registradas con ese nombre.
- **Descubrimiento Secundario:** Como mecanismo de *fallback*, los clientes escanean un rango de IP predefinido cuando el **DNS** de *Docker* no está disponible. Luego, envían a cada dirección un mensaje vía `socket` al puerto UDP en el que se encuentren escuchando los gestores, por defecto: 5353.

Los clientes entre sí no necesitan descubrirse unos a otros. Por la arquitectura propuesta, los clientes consultan a los gestores mediante `GET /users/<username>` para obtener la dirección IP y el puerto del servicio REST de cualquier otro. En caso de que los clientes cambien de dirección IP por alguna desconexión y reconexión de la red, notifican cambios de ubicación mediante `PUT /users/reconnect/<ip>/<username>` en cuanto se activen nuevamente.

6. Consistencia y Replicación

En sistemas distribuidos, la replicación de datos es una técnica fundamental para mejorar la disponibilidad, confiabilidad y rendimiento. Al mantener múltiples copias de los mismos datos en diferentes nodos, los sistemas pueden ofrecer acceso continuo incluso ante fallos parciales, reducir la latencia mediante la ubicación estratégica de réplicas cercanas a los usuarios, y distribuir la carga de trabajo para soportar un mayor número de peticiones concurrentes. Esta capacidad de replicar información de manera confiable transforma sistemas frágiles en infraestructuras resilientes capaces de operar ininterrumpidamente en entornos distribuidos complejos.

6.1. Distribución de los datos

El sistema maneja dos tipos de datos que se distribuyen de la siguiente manera:

- **Datos de usuarios:** Distribuidos entre los gestores mediante replicación completa en todas las réplicas.
- **Mensajes:** Cada cliente almacena localmente los mensajes que ha enviado y recibido.

6.2. Replicación, cantidad de réplicas

El sistema implementa un esquema de replicación configurable según los requisitos de disponibilidad. Como la toma de decisiones se basa en consenso, para lograr una tolerancia a fallas K se necesitan al menos $2K + 1$ réplicas de los datos.

Suponiendo que existen N nodos físicos en la red de *Docker Swarm*, para lograr que, ante problemas de red, cada nodo sea tolerante a $K - 1$ fallas, se necesita en cada uno al menos $2(K - 1) + 1 = 2K - 1$ réplicas, por lo que en total en el sistema se requieren $N(2K - 1)$.

En particular, para lograr la tolerancia requerida inicialmente $K = 2$ con dos nodos en la red de *Swarm*, se necesitan $2(2 * 2 - 1) = 6$ réplicas, 3 en cada nodo físico.

6.3. Confiabilidad de las réplicas de los datos

Los datos que se replican en el sistema son los de los usuarios. Lograr que las réplicas sean confiables luego de una actualización es una ardua tarea, la cual llevamos a cabo de la siguiente manera:

- **Solicitud al líder:** Los clientes envían todas sus peticiones al gestor que funciona como líder según **RAFT**. El líder es el único que realiza operaciones críticas sobre los datos con sus tareas en segundo plano.
- **Petición de consenso:** Antes de tomar acción sobre los datos, el líder convoca al resto de gestores. Si la mayoría responde, se lleva a cabo la acción, si no, queda invalidada.
- **Propagación de la solicitud:** Si se logra el consenso, el líder ejecuta la acción y propaga la solicitud a todos los *followers*. Estos solo ejecutan la acción si su *term* es menor que el *term* del líder, que se envía junto con la solicitud.

De esta manera las réplicas son confiables, ya que todas se actualizan en cada operación, sin generar pérdida de datos, quedando todas exactamente iguales.

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

La seguridad en sistemas distribuidos representa un desafío multidimensional que abarca desde la protección de las comunicaciones en tránsito hasta la correcta implementación de mecanismos de control de acceso. En un entorno donde los componentes están dispersos a través de redes potencialmente inseguras, es fundamental establecer estrategias defensivas que protejan la confidencialidad, integridad y disponibilidad de los recursos del sistema.

8.1. Seguridad con respecto a la comunicación

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.

8.2. Seguridad con respecto al diseño

La aplicación utiliza una arquitectura **P2P** donde los mensajes nunca pasan ni se almacenan en un gestor centralizado. Esto protege la privacidad y asegura que solo el usuario pueda acceder a su historial de mensajes.

8.3. Autorización y autenticación

Las sesiones están protegidas mediante un nombre de usuario único y una contraseña cifrada almacenada por los gestores, garantizando autenticación y autorización seguras. Las contraseñas se hashean con `bcrypt` antes de almacenarlas en el repositorio de usuarios. La verificación se realiza con `checkpw`.