

---

## Taller de programación

Cátedra Deymonnaz

Primer cuatrimestre 2025

Grupo X

Corrector - Firmapaz, Agustín

---

## Proyecto - RustiDocs

---

Alumno	Correo	Padrón
Bercellini, Erika Lucía	ebercellini@fi.uba.ar	110436
Bossi, Franco Alessandro	fbossi@fi.uba.ar	111122
Campillay, Edgar Matias	ecampillay@fi.uba.ar	106691
Gonzales Segura, Juan Manuel	jmgonzalezs@fi.uba.ar	110582

21 de agosto de 2025

## **Desarrollo del proyecto**

El proyecto consistió en implementar un sistema de trabajo colaborativo en tiempo real, que permitiera la edición de documentos de texto y planillas de cálculo, utilizando el lenguaje Rust como base tecnológica. Para alcanzar este propósito, se nos encomendó desarrollar una versión distribuida del sistema Redis, incorporando funcionalidades propias de un entorno colaborativo y extendiendo sus capacidades mediante la implementación de un clúster.

El desarrollo se estructuró en dos etapas principales. En la primera mitad, correspondiente a la entrega intermedia, se construyó la base del sistema, incluyendo los componentes fundamentales de almacenamiento, ejecución de comandos y la arquitectura general de un nodo. En la segunda mitad, se incorporaron las funcionalidades restantes y se completaron los aspectos avanzados del sistema, como el diseño del cluster, la replicación, la detección de fallos, el protocolo gossip, el sistema de pub/sub distribuido y la interfaz gráfica de cliente.

En las secciones siguientes se detallan las principales decisiones de diseño adoptadas, junto con la descripción de las funcionalidades clave desarrolladas a lo largo del proyecto.

## **Entrega intermedia**

### **Modelo de Actores y Arquitectura Modular**

Para organizar nuestro sistema de forma concurrente, robusta y modular, decidimos adoptar una adaptación del modelo de actores como base de nuestra arquitectura. Esta elección nos permitió estructurar el sistema en componentes independientes, cada uno encargado de una funcionalidad específica, con una clara separación de responsabilidades y comunicación segura entre ellos.

En este modelo, cada actor mantiene su propio estado interno, el cual es completamente inaccesible para el resto del sistema. La interacción entre actores se realiza exclusivamente a través del envío y recepción de mensajes asíncronos mediante `channels`. Este enfoque garantiza que cada actor procese un único mensaje a la vez, en orden FIFO, lo que simplifica el manejo de la concurrencia y elimina la posibilidad de condiciones de carrera. Además, los actores no necesitan conocer la ubicación física de sus destinatarios, lo que

permite que se comuniquen de manera transparente incluso si se encuentran en distintos hilos, núcleos o máquinas.

El sistema se compone de varios módulos funcionales, cada uno de los cuales incorpora uno o más actores para manejar su lógica interna. Entre los módulos más relevantes se encuentran:

- `networks`: gestiona las conexiones entre clientes y el servidor, y coordina la comunicación bidireccional.
- `commands`: interpreta y ejecuta los comandos que llegan desde los clientes.
- `pubsub`: implementa el sistema de publicación y suscripción distribuido.
- `configs`: administra la lectura del archivo de configuración inicial del nodo, compartido de manera estática por todo el servidor.
- `storage`: gestiona la persistencia y recuperación de datos.
- `log`: maneja el registro de eventos del sistema, aplicando filtros por nivel de log.
- `bin`: contiene los binarios principales para levantar tanto el servidor como los clientes con interfaz gráfica, permitiendo múltiples conexiones simultáneas.

## **Flujo de Ejecución y Manejo de Conexiones**

El flujo de ejecución comienza desde los archivos del módulo `bin`. En primer lugar, se lanza el servidor, que tiene la responsabilidad de inicializar los actores principales de cada módulo y establecer los canales de comunicación entre ellos. Posteriormente, uno o varios clientes pueden conectarse al servidor, levantando su propia interfaz gráfica.

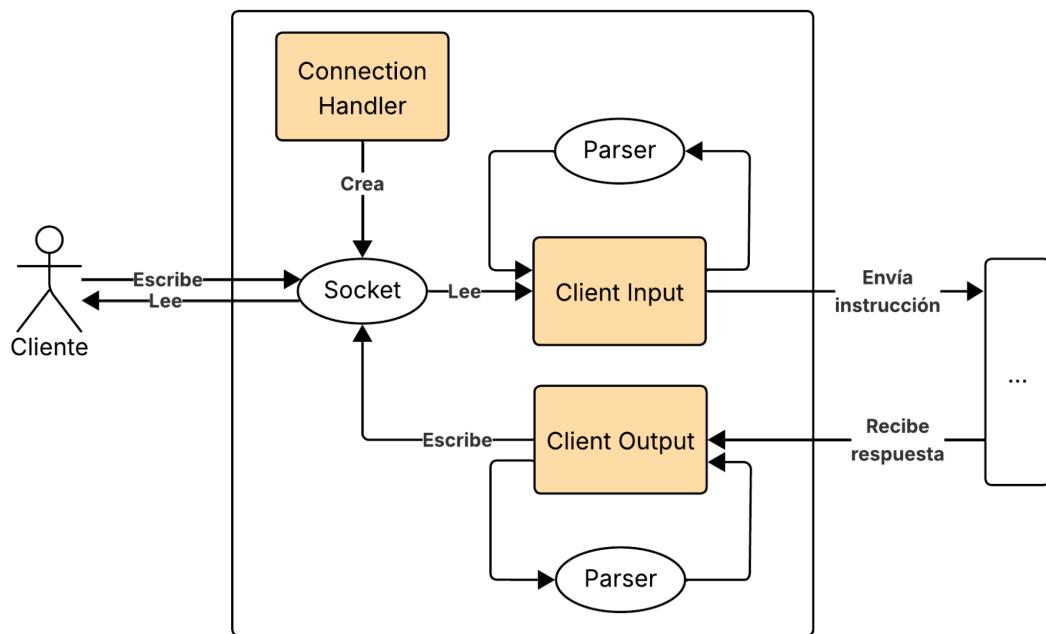
El módulo `networks` es el encargado de gestionar las conexiones cliente-servidor. Dentro de él, el archivo `connection_handler` se activa cuando un cliente solicita conexión por primera vez. Este componente acepta la conexión e inicializa las estructuras necesarias para la comunicación continua. Por cada cliente conectado, se instancian dos actores dedicados: `client_input` y `client_output`.

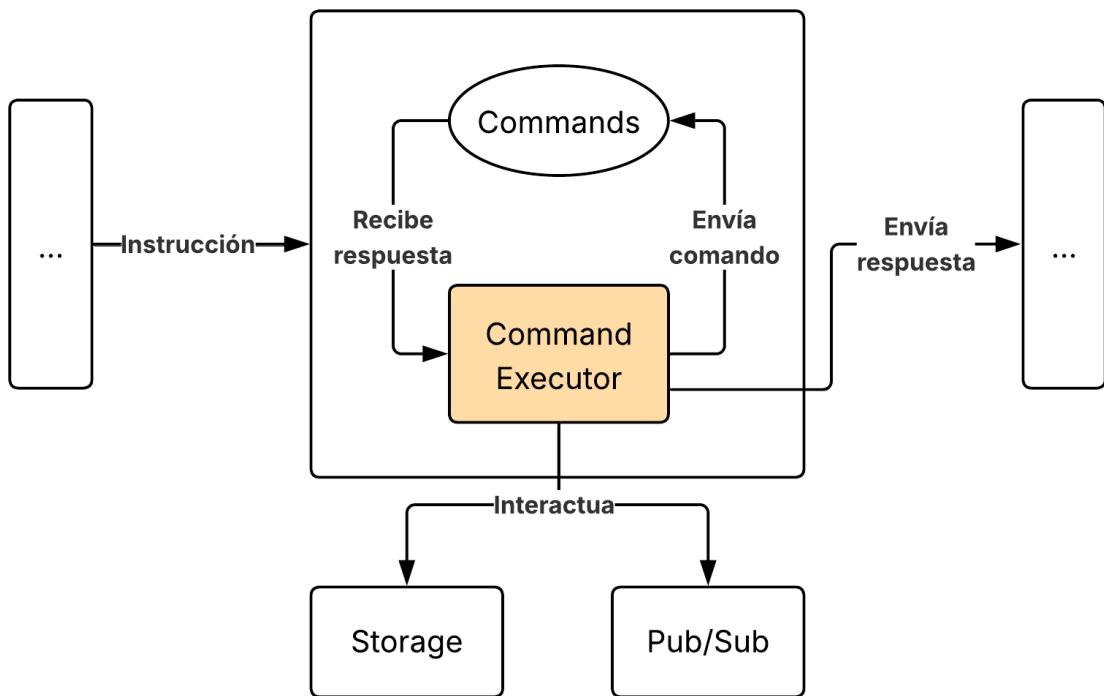
El actor `client_input` escucha los mensajes provenientes del cliente. Cada vez que recibe un comando, lo pasa por un parser que lo traduce a una estructura interna que puede ser comprendida por el sistema. Si el comando es

inválido, se notifica al usuario sin ejecutar ninguna acción. En cambio, si es válido, se reenvía al módulo de comandos.

En el módulo commands, el actor `command_executor` recibe la instrucción y ejecuta la lógica correspondiente. Dependiendo del tipo de comando, puede delegar acciones en otros módulos. Por ejemplo, si el comando requiere modificar el almacenamiento persistente, se comunica con el módulo `storage`. Si involucra operaciones de publicación o suscripción, interactúa con el módulo `pubsub`. Toda esta comunicación se realiza mediante el envío de mensajes entre actores, manteniendo la consistencia del modelo.

Una vez completada la ejecución del comando, la respuesta se envía de vuelta al módulo networks, donde el actor `client_output` la recibe y se la transmite al cliente correspondiente.





## Manejo de Desconexiones

El sistema también contempla la gestión controlada de desconexiones, tanto de clientes como del servidor. Se implementó el comando `QUIT`, que permite a un cliente cerrar su conexión con el servidor de forma ordenada, finalizando los hilos asociados a su actor `client_input` y `client_output` sin dejar recursos abiertos.

Además, se incorporó el comando `SHUTDOWN`, que puede ser ejecutado desde la consola del servidor para cerrar todo el sistema de manera segura. Este comando finaliza todos los hilos activos y libera la memoria utilizada, garantizando un cierre limpio sin fugas ni corrupción del estado persistente.

## Registro de Eventos (Logger)

El sistema implementa un componente dedicado al registro de eventos, diseñado específicamente para ofrecer trazabilidad, diagnóstico y soporte para la reconstrucción del estado del nodo en caso de fallo. Este logger funciona de forma asíncrona y estructurada, registrando todas las operaciones relevantes que se realizan sobre la base de datos.

El componente está encapsulado en una clase que actúa como un actor, aunque con una particularidad, a diferencia del resto del sistema, este actor no expone explícitamente su canal de comunicación. En su lugar, los métodos públicos del logger son responsables de encapsular los mensajes y redirigirlos al hilo de procesamiento interno, manteniendo así el ciclo de vida completamente encapsulado.

Se contemplan distintos tipos de eventos, agrupados bajo la enumeración `LogType`. Entre ellos se encuentran operaciones de escritura sobre la base de datos, eventos del sistema (como inicio y apagado), cambios de configuración, y actividades relacionadas con pub/sub y persistencia. Además, los logs se filtran por niveles (por ejemplo, `INFO`, `WARN`, `ERROR`), permitiendo modular la verbosidad del sistema según las necesidades del entorno de ejecución.

A nivel técnico, uno de los aspectos clave de la implementación es que el logger no utiliza un file handle global. Cada mensaje recibido por el actor se escribe directamente en el archivo de log correspondiente mediante operaciones de apertura y escritura rápidas (`append-only`). Este enfoque, inspirado en el modelo de Append Only File (AOF), reduce el overhead de sincronización, evita bloqueos concurrentes y mejora la eficiencia al minimizar el estado compartido.

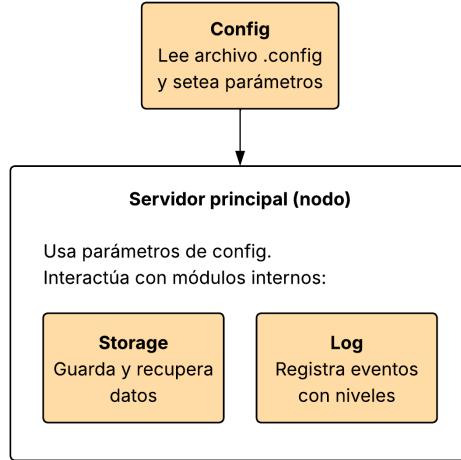
Este sistema de log proporciona no sólo visibilidad operativa, sino también una base confiable para el análisis posterior de errores, reconstrucción del estado del nodo y validación de comportamiento en entornos distribuidos.

Como detalle adicional el logger implementa el trait `Drop`, el cual aprovecha esta misma interfaz para cortar apropiadamente con su hilo en caso de cierre abrupto.

## **Almacenamiento y Persistencia de Datos**

El núcleo del almacenamiento en nuestro sistema es la estructura `DataStore`, que encapsula la información persistida de la base de datos. `DataStore` contiene tres `HashMaps` independientes, cada uno destinado a un tipo de dato: strings, listas y sets. Esta separación facilita la validación temprana de tipos y evita errores como `WRONGTYPE`, ya que al recibir un comando, basta con comprobar en cuál de los tres mapas se encuentra la clave, evitando ambigüedades.

Para garantizar seguridad y concurrencia, la estructura está protegida mediante `Arc<RwLock<DataStore>>`, permitiendo múltiples operaciones de lectura simultánea sin bloquear el acceso de escritura. Esta decisión arquitectónica



habilita funcionalidades como la persistencia en segundo plano sin interrumpir el servicio.

## Mecanismos de Guardado

La persistencia del sistema está diseñada para garantizar durabilidad incluso ante reinicios o fallos. Para ello se implementaron tres mecanismos de guardado que convergen en una misma función central, `create_dump()`:

- **Backup periódico:** un actor dedicado, `snapshot_manager`, corre en un hilo separado que permanece dormido hasta activarse a intervalos configurados. Durante su activación, accede a la base de datos en modo lectura no bloqueante y serializa su contenido.
- **Backup por umbral de escritura:** el actor `command_executor` lleva un contador interno de operaciones de escritura. Al alcanzar un múltiplo predefinido, se dispara un backup inmediato en primer plano.
- **Backup manual:** mediante los comandos `SAVE` (sin bloqueo, en primer plano) y `BGSAVE` (en hilo paralelo), el usuario puede forzar una persistencia de datos. En el caso de `BGSAVE`, se genera un evento de log indicando cuándo finaliza el proceso.

Todos estos caminos convergen en el uso del módulo `serializer`, encargado de transformar el estado de `DataStore` a un archivo `.rdb` legible y eficiente.

## Mecanismo de Recuperación

Al iniciar el nodo, el módulo `DiskLoader` intenta recuperar el estado de la base de datos leyendo el archivo `.rdb` especificado en el archivo `.conf`. Si dicho archivo existe, se deserializa su contenido utilizando el módulo `deserializer`; en caso contrario, se inicializa una `DataStore` vacía.

El proceso de deserialización comienza con la creación de una `DataStore` limpia. A continuación, se recuperan los datos en el siguiente orden: strings, listas y sets. Para cada entrada, se lee su longitud, se recupera su contenido desde el archivo, y se la inserta en la base de datos correspondiente. En los casos de listas y sets, primero se determina la longitud de la colección y luego se insertan sus elementos individualmente.

Este mecanismo garantiza que el nodo pueda restaurar su estado previo con precisión, incluso en entornos donde la concurrencia y la consistencia son críticas.

## Sistema Pub/Sub interno

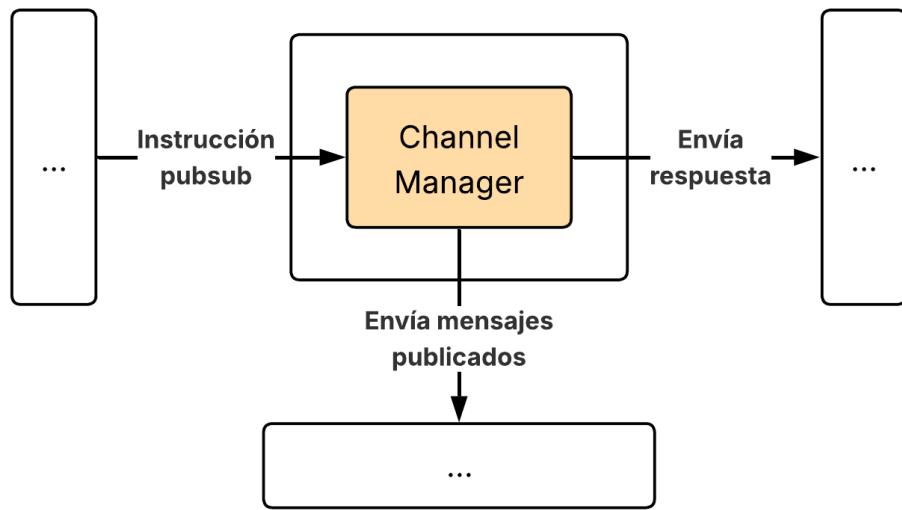
La arquitectura del sistema pub/sub también incluye un componente interno especializado: el `channel_manager`, encargado de manejar de forma eficiente los canales y sus suscriptores.

Este módulo mantiene un `HashMap` cuya clave es el nombre del canal, y cuyo valor es otro mapa que relaciona IDs de clientes con sus correspondientes `Sender`. Esta estructura permite enviar mensajes directamente a cada cliente suscrito a un canal determinado, sin necesidad de rutas adicionales.

Cada vez que un cliente realiza una operación relacionada con pub/sub, se envía un mensaje al `channel_manager` con la siguiente información:

- ID del cliente
- Comando a ejecutar (`SUBSCRIBE`, `UNSUBSCRIBE`, `PUBLISH`)
- Un `Sender` para responder sobre el resultado de la operación
- Un `Sender` adicional para enviar mensajes al cliente si corresponde

Este modelo permite gestionar suscripciones y publicaciones de manera eficiente, garantizando aislamiento, escalabilidad y entrega confiable de mensajes.

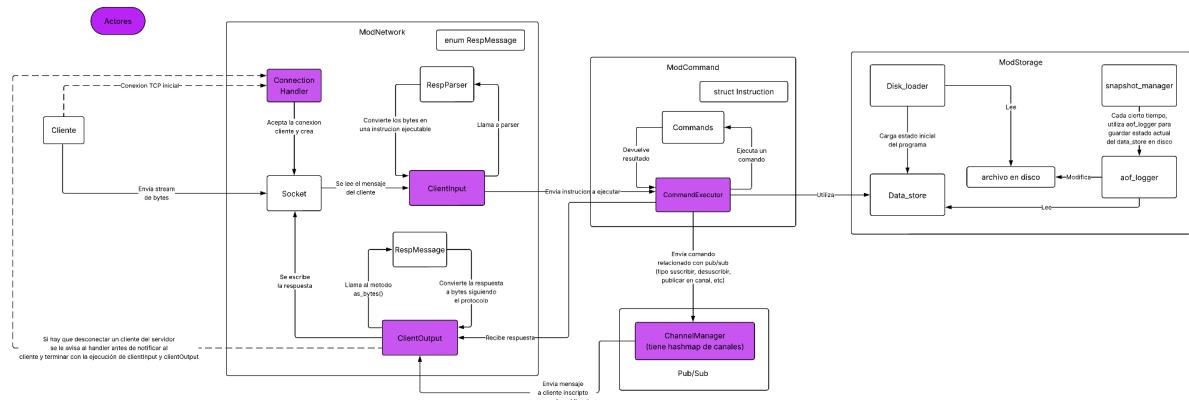


## Interfaz Gráfica

Con el objetivo de ofrecer una experiencia de usuario sencilla y efectiva, desarrollamos una interfaz gráfica utilizando egui, una biblioteca de GUI inmediata escrita en Rust. Esta librería destaca por su bajo peso, facilidad de integración y naturaleza reactiva, lo que nos permitió construir una interfaz interactiva sin incurrir en complejidades innecesarias.

La interfaz fue desarrollada sobre eframe, el framework oficial de aplicaciones para egui, lo que simplificó el ciclo de desarrollo y permitió una integración directa con nuestro cliente. Gracias a esto, fue posible conectar múltiples clientes al servidor en simultáneo, visualizar respuestas, ejecutar comandos y observar el estado del sistema en tiempo real.

## Diagrama de un nodo



## Entrega final

### Estructura del clúster

El sistema desarrollado está diseñado para operar como un clúster distribuido compuesto por múltiples nodos. Cada nodo se inicializa a partir de una estructura central denominada `ClusterNode`, que se encarga de instanciar y coordinar todos los componentes necesarios para su funcionamiento e integración dentro del clúster.

Durante la inicialización, `ClusterNode` crea una instancia de `NodeData`, que encapsula la información del nodo local, incluyendo su identificador, el rango de slots que administra y otros metadatos. Además, mantiene una lista de `KnownNodes`, que contiene los datos relevantes de los demás nodos conocidos en el sistema. Esta estructura permite establecer una red de nodos interconectados, donde cada uno conoce al menos una parte del estado global del clúster. Gracias a este diseño modular, el sistema permite una incorporación dinámica de nuevos nodos, favoreciendo la escalabilidad sin necesidad de reconfiguración manual ni interrupciones.

### Jerarquía de nodos

El cluster cuenta con un sistema de asignación y promociones basado en tanto la cantidad de nodos masters como el estado de los mismos.

- Un nodo será asignado como máster si:
  - Hay **menos de 3 masters** presentes con estado **válido** en todo el cluster.
  - Es una réplica cuyo master falló **definitivamente**.
- Un nodo será asignado como slave si:
  - Hay más de 3 masters en estado válido.

Como detalles extra, un nodo slave al momento de recibir un mensaje de tipo `JoinMessage`, lo redirige a un master conocido si el próximo nodo a agregar debe ser un master para evitar problemas de consistencia en el nodo slave ya presente el cluster. Y en caso de asignar el nuevo miembro como slave al momento de asignar un master se elige el que posea menos réplicas, para así mantener el **cluster equilibrado**.

## Comunicación entre nodos

Los canales de salida y de entrada de comunicación del nodo, están centralizados en las funciones del módulo `NodeInput` y la estructura `NodeOutput`.

- `NodeInput`, consiste de varias funciones que escuchan activamente por nuevas comunicaciones dentro del puerto `CLIENT_PORT + 1000` abriendo un hilo por cada cliente nuevo en el mismo. Dentro se llama a la función `handle_connection` que escucha el socket, guarda los mensajes recibidos y los delega al módulo correspondiente para el procesamiento del mismo.
- `NodeOutput`, actor que se encarga de **mantener la comunicación de salida del nodo**. Guarda un diccionario `HashMap<NodeId, TcpStream>` y ya sea mediante el método `send_to_node` o el `Sender<NodeId, NodeAddr, Option<Vec<u8>>` compartido al crear, crea la conexión de salida con el destino y comparte los bytes en caso de haberlos. Esta estructura también posee soporte para broadcasts y el envío de mensajes especiales para el pubsub distribuido.

## Protocolo de comunicación

Respecto a los mensajes enviados y recibidos por el nodo se encuentran:

- `GossipMessage`, actúa como un `PING` que se manda periódicamente cada 1 segundo. Únicamente contiene el `ping_id` (identificador único del ping), `pong_id` (replica el `ping_id` recibido), las `FLAGS` del nodo y el vector de entradas `gossip` (`GossipEntry` que contiene la información del nodo a distribuir).
- `JoinMessage`, uno de los más importantes, permite a un nodo nuevo unirse a un cluster. Triggereado al pasar por parámetro de configuración la dirección del nodo conocido o al usar el comando `MEET`.
- `RehashMessage`, un mensaje que reorganiza los slots del receptor y le asigna rol.
- `PubSubMessage`, útil para la comunicación pubsub entre nodos (ver detalles en las siguientes secciones).
- `FailMessage`, utilizado para el broadcast al comunicar la falla definitiva de un nodo.
- `PromotionMessage`, encargado de la transmisión de información durante la elección de réplicas.

- `PSyncMessage`, distribuye los slots entre master y réplica para mantener la información al día.

Como último todas los mensajes se envuelven dentro de otra clase `NodeMessage`, que contiene la información del remitente, el tipo de mensaje que transmite y la longitud junto con los bytes a leer para parsear correctamente el mensaje.

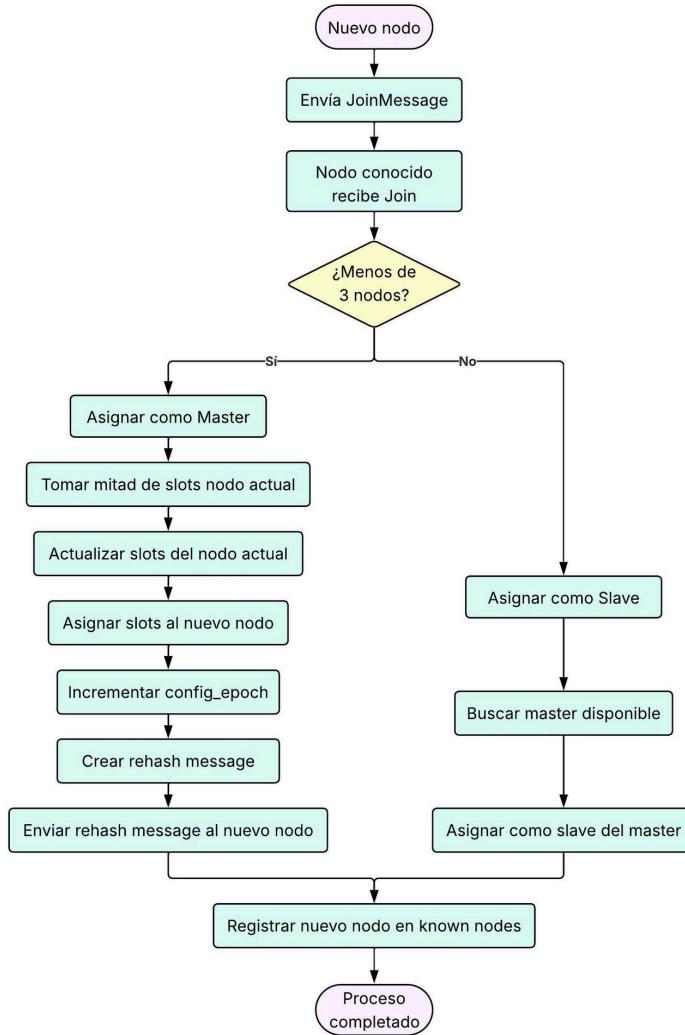
## Sistema de Sharding

Para distribuir la carga de trabajo y los datos de manera eficiente entre los nodos del clúster, el sistema implementa un mecanismo de sharding. Este proceso consiste en asignar cada clave de datos a uno de los 16.384 hash slots disponibles, numerados del 0 al 16.383. Cada nodo master es responsable de un subconjunto de estos slots, lo cual permite distribuir equitativamente la información entre múltiples nodos y escalar horizontalmente a medida que crece la demanda.

La asignación de slots se realiza utilizando **el algoritmo CRC16-XMODEM**, implementado en el módulo `hash_slot.rs`. Este algoritmo, por ser determinístico, garantiza que una misma clave sea siempre asignada al mismo slot, lo que asegura la consistencia de la distribución.

Cada nodo mantiene en su estructura `NodeData` información sobre los slots que administra. Esta información permite determinar rápidamente si un nodo es responsable de una clave determinada. En caso de que un cliente envíe un comando a un nodo incorrecto, el componente `CommandExecutor` se encarga de redirigir automáticamente el comando al nodo correcto, haciendo que el sistema resulte transparente desde la perspectiva del cliente.

Al iniciarse el primer nodo del clúster, se le asignan **todos los slots**. Posteriormente, cuando se incorporan nuevos nodos y se los configura como masters, el sistema redistribuye los slots existentes entre ellos. Esta redistribución se realiza mediante la función `rehash`, ubicada en el módulo `join_message.rs`, que reasigna parte de los slots al nuevo nodo de forma automática, garantizando así un balance de carga sin comprometer la disponibilidad del sistema.



## Detección de Fallos y Protocolo de Gossip

El sistema implementa un protocolo de gossip asincrónico y eventualmente consistente para la detección de fallos y la propagación de información sobre el estado del clúster. Este protocolo permite que los nodos intercambien periódicamente información sobre otros nodos conocidos, asegurando que todos los integrantes del clúster converjan hacia una visión compartida y actualizada del sistema.

La selección del nodo destino para el envío de mensajes PING se realiza de manera aleatoria entre todos los nodos conocidos que no estén marcados como fallidos. Esta aleatoriedad es esencial para evitar sesgos en la comunicación y lograr una propagación eficiente de la información. Cada mensaje PING incluye además un conjunto de entradas de gossip sobre otros nodos (típicamente tres

por mensaje), lo cual permite distribuir información de manera incremental sin sobrecargar la red.

El sistema de detección de fallos se basa en un esquema de timeouts. Cada nodo mantiene un componente `TimeTracker`, que registra tanto el momento del último PING enviado como el de la última respuesta PONG recibida por cada nodo conocido. Si no se recibe una respuesta dentro del tiempo límite (establecido en 10 segundos), el nodo en cuestión se marca en estado `PFAIL` (posible fallo). Este estado se incluye en los mensajes de gossip, lo que permite su propagación a otros nodos. Este `TimeTracker` consiste de una estructura tipo `HashMap<ping_id, nodo_destino_id>` junto con una cola auxiliar que guarda los `ping_id` para mantener trackeado el orden en caso de paquetes PING recibidos fuera de orden.

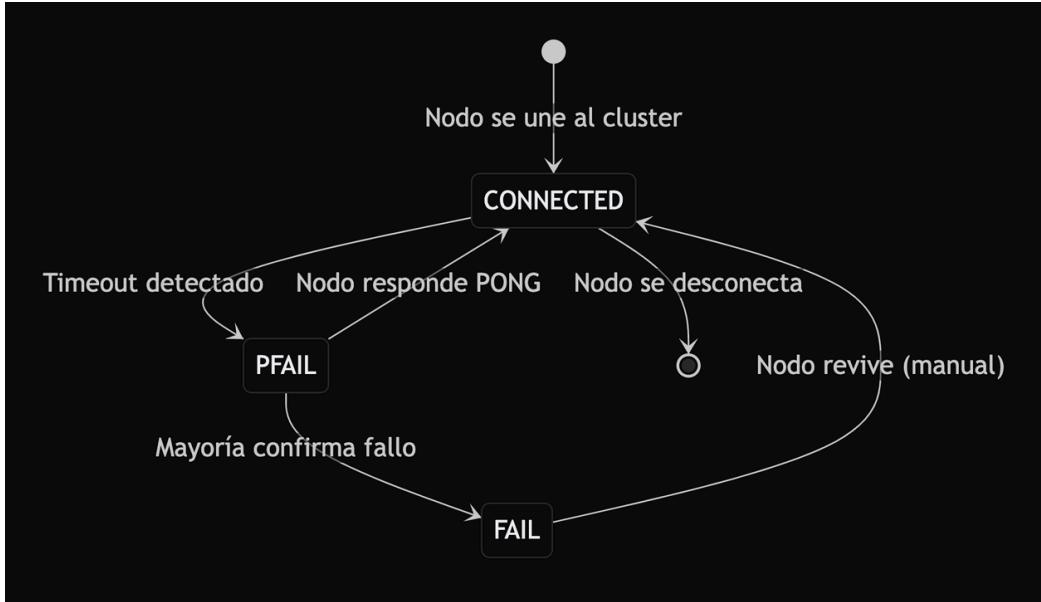
La transición de `PFAIL` a `FAIL` (fallo definitivo) no es automática: requiere un proceso de votación entre los nodos master. Cuando uno de ellos detecta un `PFAIL`, espera un período adicional (usualmente el doble del timeout) para recolectar votos de otros masters. Si la mayoría confirma el fallo, el nodo se declara oficialmente como `FAIL` y se inicia el **proceso de recuperación**.

Además de la detección de fallos, el protocolo de gossip también propaga información sobre eventos importantes del clúster, como redistribuciones de slots, cambios de roles entre masters y slaves, o incorporación de nuevos nodos. Para garantizar la consistencia, cada nodo mantiene un epoch de configuración que se incrementa ante cada cambio significativo. Los mensajes de gossip incluyen esta información, y cada nodo solo actualiza su estado si detecta que la información recibida es más reciente (ya sea por epoch o por timestamp).

Se han implementado mecanismos específicos para evitar bucles de propagación y garantizar la convergencia del sistema. Los nodos ignoran información obsoleta y solo aceptan actualizaciones más recientes. También se utilizan flags para representar estados transitorios (propios del POV del sender) como:

- `HANDSHAKE` (durante la incorporación de nuevos nodos).
- `NOADDR` (se conoce al nodo pero no se estableció conexión alguna todavía).
- `CONNECTED` (cuando la conexión ya fue establecida) e informar a los demás miembros del cluster de del estado del nodo actual con `CONNECTED`, `PFAIL`, `FAIL`, `MASTER` y `SLAVE`.

En conjunto, este protocolo de gossip permite que el sistema mantenga su operatividad incluso en presencia de fallos o latencias variables, lo que lo convierte en un componente clave para la escalabilidad y resiliencia del clúster.



## Promoción Automática de Rélicas

Para mantener la disponibilidad ante fallos de nodos master, el sistema incorpora un mecanismo automático de promoción de rélicas. Este proceso se activa tras la declaración definitiva de fallo (`FAIL`) de un master, y tiene como objetivo redistribuir sus slots a una de sus rélicas, asegurando así la continuidad del servicio.

La lógica comienza en el módulo `failing_node.rs`, dentro de la función `start_fail_procedure`, la cual no solo marca al nodo como `FAIL` tras el consenso de votación, sino que también desencadena la promoción automática llamando a `start_promotion`. Este procedimiento se ejecuta en un hilo separado, para no bloquear el procesamiento principal del nodo.

La promoción se gestiona desde el módulo `replica_promotion.rs`. El sistema primero introduce un pequeño retardo (1 segundo) para permitir que el estado de fallo se propague por completo al resto del clúster. A continuación, identifica todas las rélicas asociadas al nodo master fallido que estén activas y no hayan sido marcadas como fallidas. Entre estas, se selecciona como candidata la réplica con cuyo `TimeStamp` de mensaje `PSync` recibido correctamente sea más reciente, ya que es la que contiene los datos más actualizados de su master.

Si no hay réplicas disponibles o todas están en estado FAIL, el proceso **se interrumpe sin realizar promoción**. En cambio, si se identifica una candidata válida, se genera un mensaje de promoción que contiene su ID, el ID del master fallido, el conjunto de slots a reasignar y un nuevo epoch de configuración. Este mensaje se difunde a todos los nodos del clúster utilizando un broadcast a todos los nodos conocidos.

La función `process_promotion_msg` es la encargada de procesar estos mensajes. Antes de proceder, realiza varias validaciones: verifica que el nodo promovido sea efectivamente una réplica del master fallido, que el master esté realmente marcado como FAIL, y que el epoch sea válido. Si todo está en orden, se actualiza el estado de la réplica, promoviéndola a master y asignándole los slots correspondientes. También se eliminan los slots del nodo fallido, evitando inconsistencias o conflictos.

Para asegurar la consistencia global, el nuevo epoch de configuración garantiza que todos los nodos reconozcan la nueva estructura como más reciente. Este enfoque evita que configuraciones obsoletas sobrescriban el nuevo estado del clúster.

El proceso de promoción se encuentra totalmente integrado al sistema de comunicación distribuida. Los mensajes se transmiten y procesan de manera asíncrona, lo que permite que el clúster continúe funcionando normalmente durante la recuperación. Esta funcionalidad refuerza la disponibilidad del sistema y lo prepara para operar de forma robusta ante fallos sin intervención manual.



## Sistema de Pub/Sub Distribuido

El sistema desarrollado incluye una arquitectura de publish/subscribe (pub/sub) completamente distribuida, que permite a los clientes suscribirse a canales desde cualquier nodo del clúster y recibir mensajes publicados desde cualquier otro nodo de forma transparente. Esta implementación elimina la necesidad de un coordinador central y garantiza una propagación eficiente de los mensajes entre los nodos participantes.

Cada nodo cuenta con un componente denominado `DistributedPubSubManager`, que actúa como gestor de pub/sub local. Este componente mantiene dos estructuras de datos principales: por un lado, un mapa que asocia cada canal con los suscriptores locales conectados al nodo actual; y por otro, un registro de los nodos remotos que tienen clientes suscritos a cada canal. Esta organización permite que los mensajes publicados se distribuyan únicamente a los nodos relevantes, optimizando el tráfico en la red.

El sistema utiliza tres tipos de mensajes para coordinar la actividad de pub/sub entre nodos:

1. `Subscribe`, que notifica a los demás nodos que un cliente se ha suscripto a un canal;
2. `Unsubscribe`, que informa sobre una desuscripción; y
3. `Publish`, que se encarga de propagar los mensajes a todos los suscriptores del canal, estén donde estén.

Cuando un cliente envía un comando `SUBSCRIBE`, el `CommandExecutor` lo identifica como una operación de pub/sub y delega su manejo al `DistributedPubSubManager`. El cliente se registra como suscriptor local del canal, y el sistema propaga un mensaje `Subscribe` al resto de los nodos. Cada nodo receptor registra que el nodo origen tiene suscriptores en ese canal, manteniendo así un mapa actualizado de la distribución de suscripciones en el clúster.

En el flujo de publicación, un cliente envía un comando `PUBLISH` con el canal y el mensaje correspondiente. El nodo receptor entrega el mensaje a todos los suscriptores locales y luego propaga un mensaje `Publish` al resto de los nodos del clúster. Cada nodo que recibe este mensaje verifica si tiene suscriptores locales del canal, y en tal caso reenvía el mensaje, completando así la entrega distribuida.

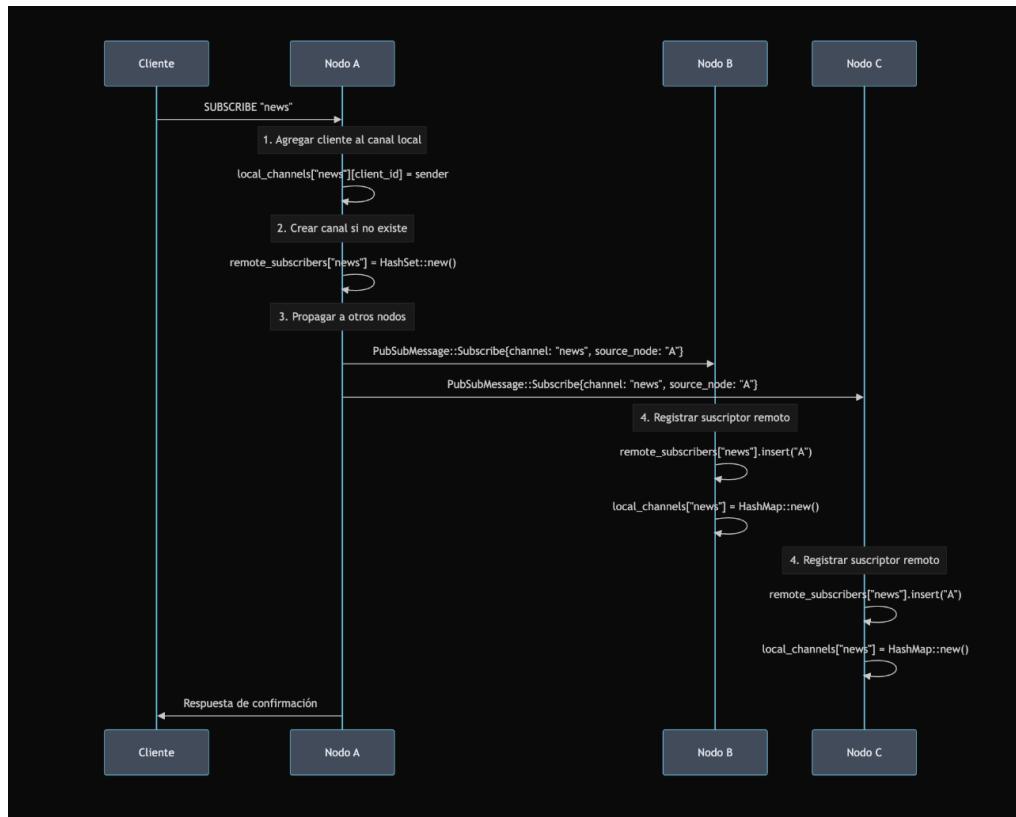
El proceso de desuscripción sigue un flujo inverso. Al recibir un comando `UNSUBSCRIBE`, el sistema remueve al cliente del canal local. Si el canal queda sin suscriptores en ese nodo, se elimina de la estructura local, y se envía un mensaje `Unsubscribe` a los demás nodos para que actualicen su registro.

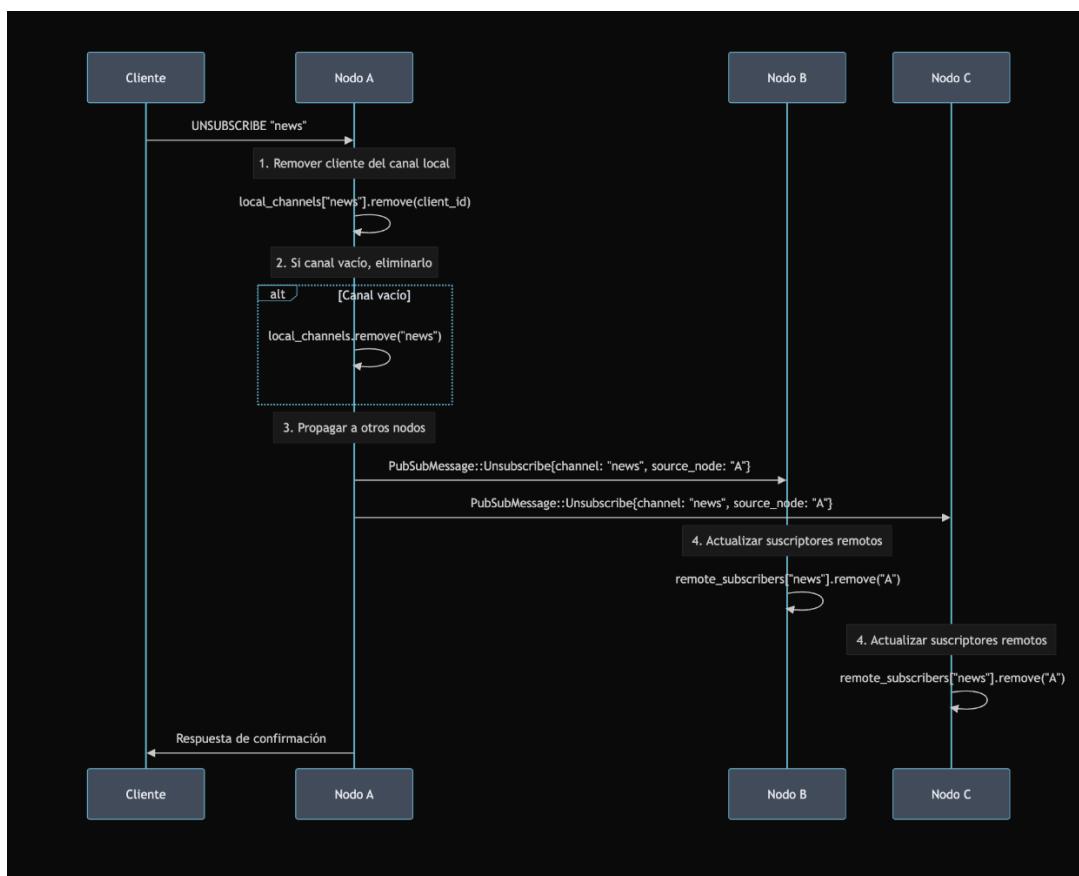
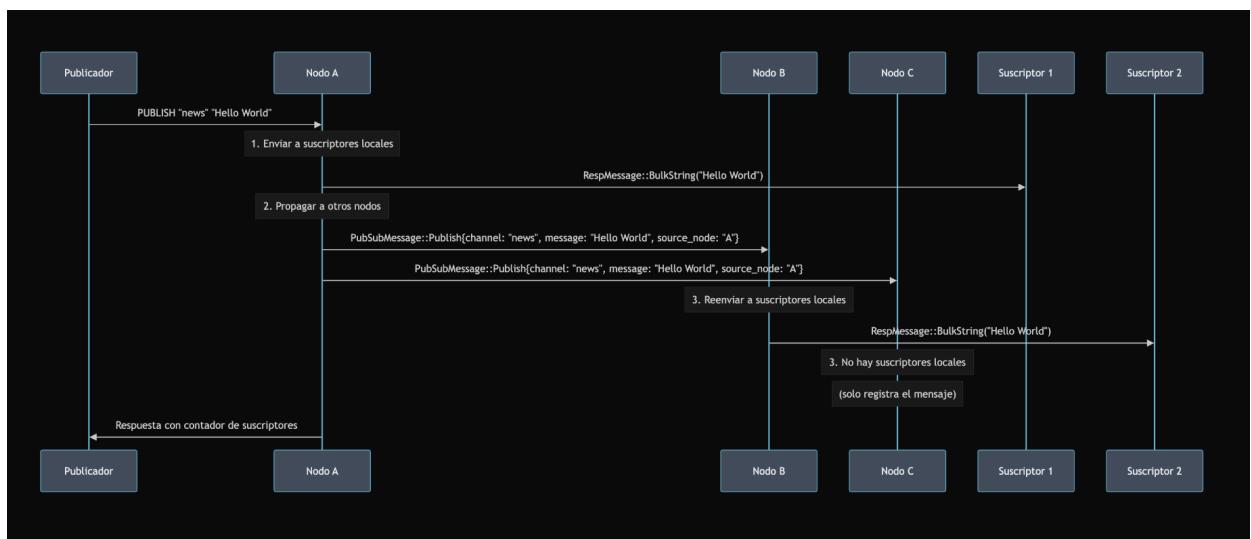
La comunicación entre nodos se gestiona a través del `ClusterCommunicationManager`, que se encarga de serializar y deserializar los mensajes de pub/sub, y de integrarlos dentro del canal de comunicación general del clúster. Esto permite un uso eficiente de los recursos y asegura la coherencia de la información entre nodos.

Entre las características destacadas del sistema se incluyen la creación automática de canales al momento de la suscripción, y su eliminación automática cuando ya no existen suscriptores, lo que contribuye a una gestión eficiente de la memoria. Además, la propagación de mensajes es selectiva: solo se envían a nodos que efectivamente poseen suscriptores del canal correspondiente, evitando tráfico innecesario.

El diseño permite una alta tolerancia a fallos, ya que si un nodo se desconecta, los demás pueden continuar operando sin interrupciones. A su vez, los clientes no requieren conocimiento sobre la distribución interna del clúster, ya que pueden suscribirse y publicar desde cualquier nodo, beneficiándose de una experiencia transparente y consistente.

Este enfoque distribuido al pub/sub aporta escalabilidad, alta disponibilidad y baja latencia en la entrega de mensajes, todo ello con una consistencia eventual que garantiza que los eventos publicados alcancen a todos los destinatarios relevantes, incluso en presencia de fallos temporales o latencias en la red.





## Protocolo Psync

En nuestro sistema para la gestión de datos y almacenamiento entre nodos réplicas y sus maestras.

## Mecanismo de Sincronización

Las réplicas, actuando como iniciadores del proceso, envían periódicamente su estado actual al maestro correspondiente. Esta aproximación proactiva asegura que las réplicas nunca permanezcan desactualizadas por largos períodos, mejorando la consistencia global sin necesidad de implementar protocolos complejos de consenso distribuido como Raft o Paxos. El intervalo de cinco segundos entre sincronizaciones proporciona un equilibrio óptimo entre frescura de datos y sobrecarga de red.

## Implementación Técnica

La implementación técnica del protocolo evidencia un diseño cuidadoso. El uso de estructuras de datos bien definidas como `PsyncMessage` facilita la serialización y deserialización de información entre nodos, mientras que el empleo de canales (`channels`) de Rust para la comunicación asíncrona permite un desacoplamiento efectivo entre los componentes del sistema. El código muestra una clara separación de responsabilidades: el `psync_sender` se enfoca exclusivamente en la emisión de solicitudes, mientras que el `psync_reciever` se especializa en el procesamiento y respuesta. Esta modularidad facilita el mantenimiento y la evolución del código.

## Eficiencia del Protocolo

El protocolo implementado destaca por su eficiencia operativa. Aunque envía el `DataStore` completo en cada sincronización, esta decisión simplifica enormemente la lógica de reconciliación. El enfoque "todo o nada" elimina la necesidad de rastrear cambios incrementales o mantener registros de operaciones, reduciendo significativamente la complejidad del sistema. Para deployments con volúmenes de datos moderados, esta estrategia ofrece un excelente equilibrio entre simplicidad implementativa y rendimiento.

## Relación entre Tipos de Datos y Funciones:

La implementación muestra una clara relación entre tipos de datos y funciones que facilita el flujo de información a través del sistema:

- 1) `NodeData` y `DataStore`: Existe una relación complementaria donde `NodeData` contiene la información de identidad y rol del nodo, mientras que `DataStore` contiene los datos reales que necesitan sincronizarse. Esta separación de preocupaciones permite que el protocolo `PSYNC` se

concentre exclusivamente en la sincronización de datos sin mezclar aspectos de gestión de nodos.

- 2) `PsyncMessage` y `NodeMessage`: Estas estructuras forman una jerarquía de encapsulamiento donde `NodeMessage` actúa como envoltura general para comunicaciones entre nodos, mientras que `PsyncMessage` contiene la información específica para la sincronización (`node_id` y `data_store`). Esta relación de composición permite al sistema de mensajes general enrutar correctamente los mensajes PSYNC.
- 3) Relación Funcional: La función `process_psync_message` establece una clara relación de transformación entre el mensaje recibido, los datos locales y el mensaje de respuesta, actuando como un transformador que toma datos de entrada de múltiples fuentes y produce una salida coherente y actualizada.
- 4) Canal de Comunicación: El parámetro `output: &Sender<(NodeId, SocketAddr, Option<Vec<u8>>)>` establece una relación de comunicación asincrónica entre el procesador PSYNC y el sistema de envío de mensajes, permitiendo desacoplar el procesamiento de la comunicación de red.

## **Encriptación de datos en tránsito (in-transit)**

Con el objetivo de proteger la confidencialidad e integridad de las comunicaciones dentro de nuestro sistema distribuido, analizamos la arquitectura del proyecto e identificamos varios puntos críticos que requerían mecanismos de cifrado: la comunicación entre nodos del clúster (gossip, replicación), entre nodos y el microservicio (persistencia y control), y entre el microservicio y la interfaz (gestión del sistema).

## **Componentes Principales del Sistema de Encriptación**

- **Módulo de Criptografía:** Contiene algoritmos básicos de cifrado simétrico, un generador de números pseudoaleatorios (xorshift) y funciones hash (FNV-1a) para autenticación de datos. También proporciona `EncryptedStream`, que permite cifrar flujos de datos sobre TCP de forma transparente.
- **Protocolo TLS Simplificado:** Implementa un protocolo de handshake en cuatro fases (`ClientHello`, `ServerHello`, `KeyExchange`, `Finished`), que establece una clave simétrica compartida entre cliente y servidor.

Ambos extremos operan como streams estándar, lo que facilita su integración en la lógica existente.

- **Gestión de Certificados:** Utilizamos un esquema de certificados autofirmados, con campos básicos como versión, número de serie, sujeto, emisor, validez temporal, clave pública y firma. Se incluyen funciones para generar, guardar y cargar certificados en formato PEM, especialmente útil en entornos de desarrollo.

## Integración con el Sistema de Clúster

El sistema permite configurar dinámicamente el tipo de encriptación:

- Sin encriptación: útil en entornos de desarrollo o pruebas controladas.
- Con TLS simplificado: recomendado para garantizar seguridad en producción.

La API permite verificar el estado de encriptación y el tipo activo, proporcionando flexibilidad según los requisitos de seguridad.

## Flujo Completo de Encriptación Nodo a Nodo

El proceso completo de encriptación entre nodos sigue las siguientes etapas:

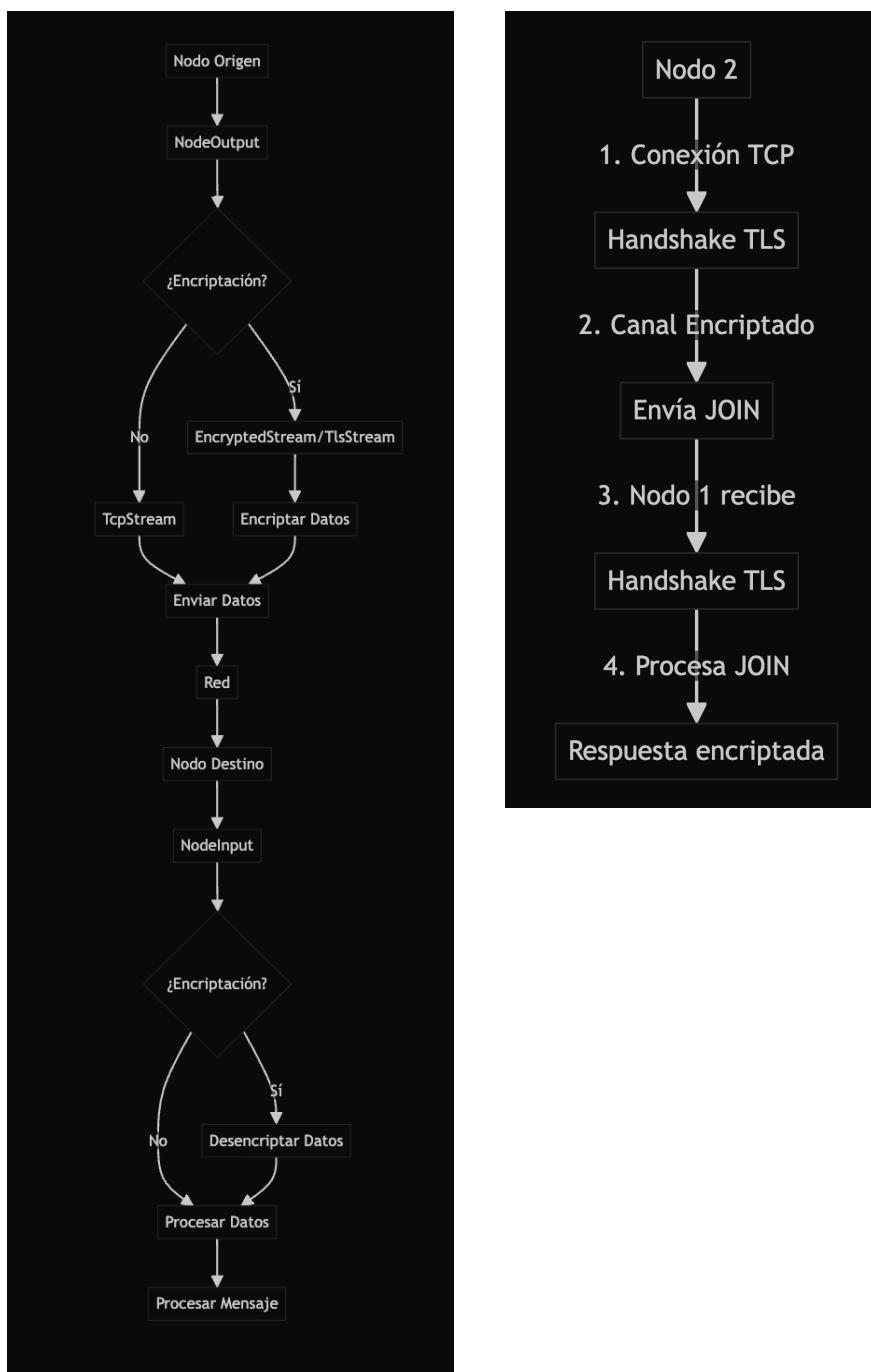
- Inicialización
  - Se configura cada nodo con el tipo de encriptación deseado (ninguna o TLS simplificado).
  - Se crean los streams encriptados adecuados según esa configuración.
  - Si se emplea TLS, se ejecuta el handshake para negociar una clave compartida.
- Transmisión de Datos
  - Los mensajes se serializan en el formato interno del sistema.
  - Se cifran utilizando el algoritmo configurado.
  - Los datos encriptados se envían a través de la red.
  - El nodo receptor desencripta los datos y procesa el mensaje.

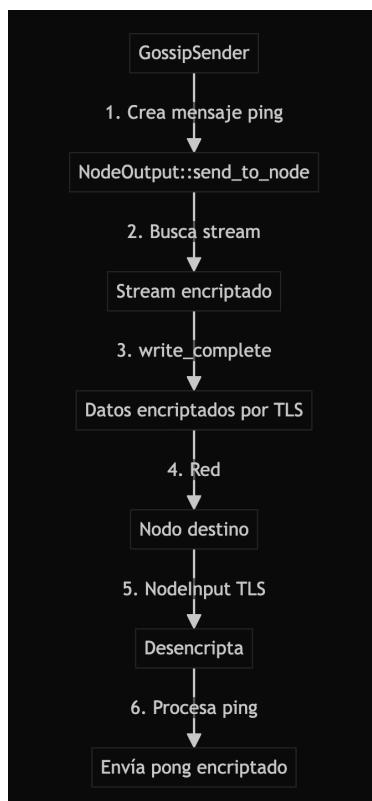
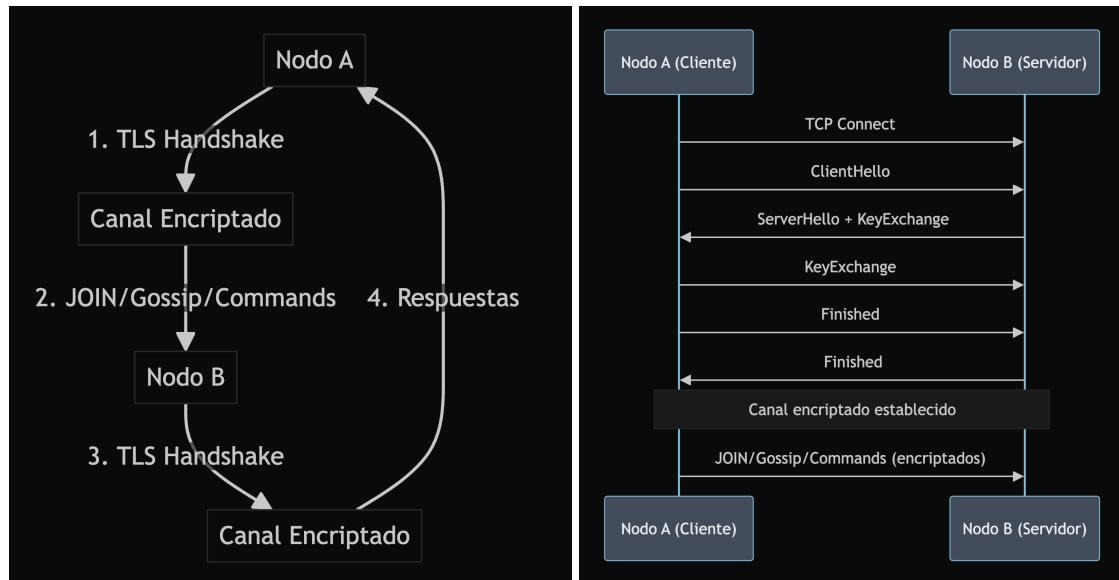
## Seguridad y Consideraciones

- **TLS Simplificado:** Aunque no implementa completamente el estándar TLS, nuestro protocolo básico incluye intercambio de claves y validación, ofreciendo un nivel razonable de seguridad para desarrollo avanzado o entornos controlados.

- **Compatibilidad entre Nodos:** Todos los nodos involucrados en una comunicación deben utilizar el mismo tipo de encriptación para evitar errores de interpretación y garantizar interoperabilidad.
- **Tolerancia a fallos:** En caso de que falle el proceso de encriptación o desencriptación, se interrumpe la conexión para evitar la exposición accidental de datos sin protección.

A continuación, se incluyen gráficos explicativos del proceso de encriptación y desencriptación implementado.





## Autenticación

Para llevar a cabo el manejo de privilegios de usuarios y la autenticación, se utilizó una lista de control de accesos ACL presente en el submódulo `users` parte del módulo `security`.

Cada usuario es declarado dentro del archivo `user.acl` con la siguiente sintaxis.

```
user nombre >contraseña +comando_permitido
```

Hay tres tipos de privilegios:

- Super, declarado con \* en lugar de una instrucción/es particular. Permite a los privilegios de dicho usuario acceder a **todas** las funcionalidades del sistema (acceso a interfaz completa).
- Limitados, son aquellos cuyas operaciones permitidas solo se limitan a las declaradas en su línea.
- Solo lectura, es un usuario que no puede ejecutar comandos SET o no posee comando alguno (acceso a interfaz con interactividad limitada).

Para asegurar el correcto funcionamiento del sistema cada usuario debe autenticarse tan pronto como se conecte al servidor utilizando el comando AUTH. Los canales propios del pubsub distribuido utilizan un superusuario especial para poder realizar sus actividades sin limitación alguna.

## **Interfaz gráfica**

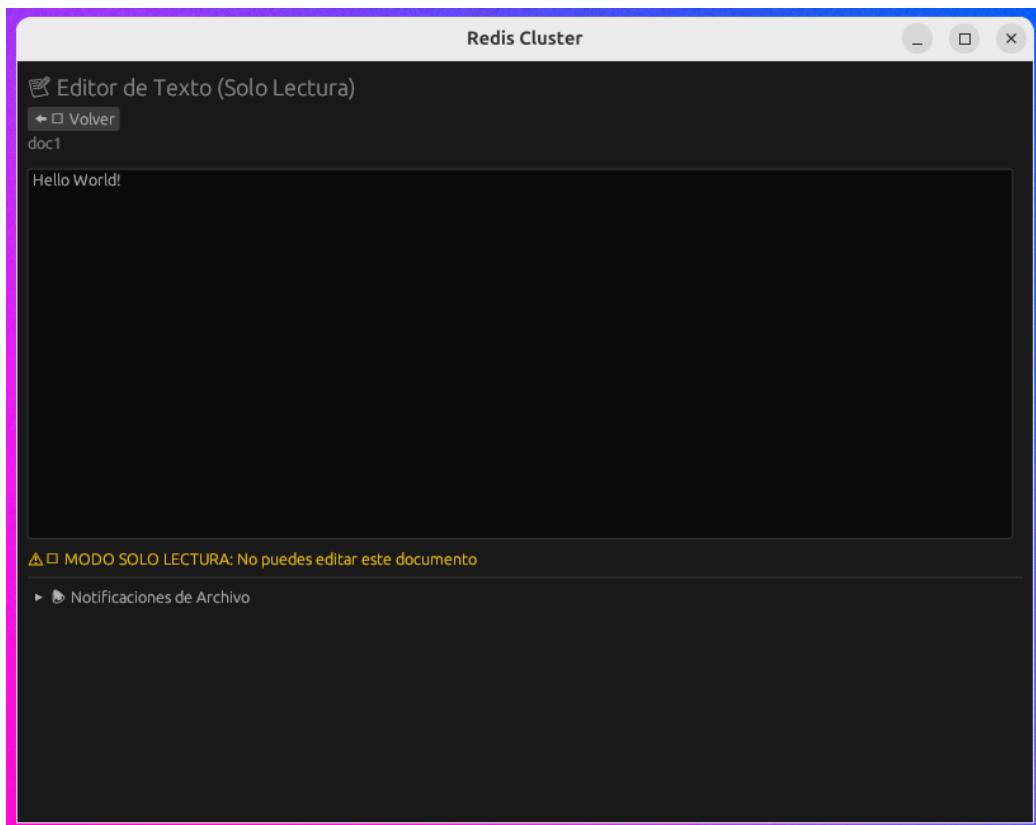
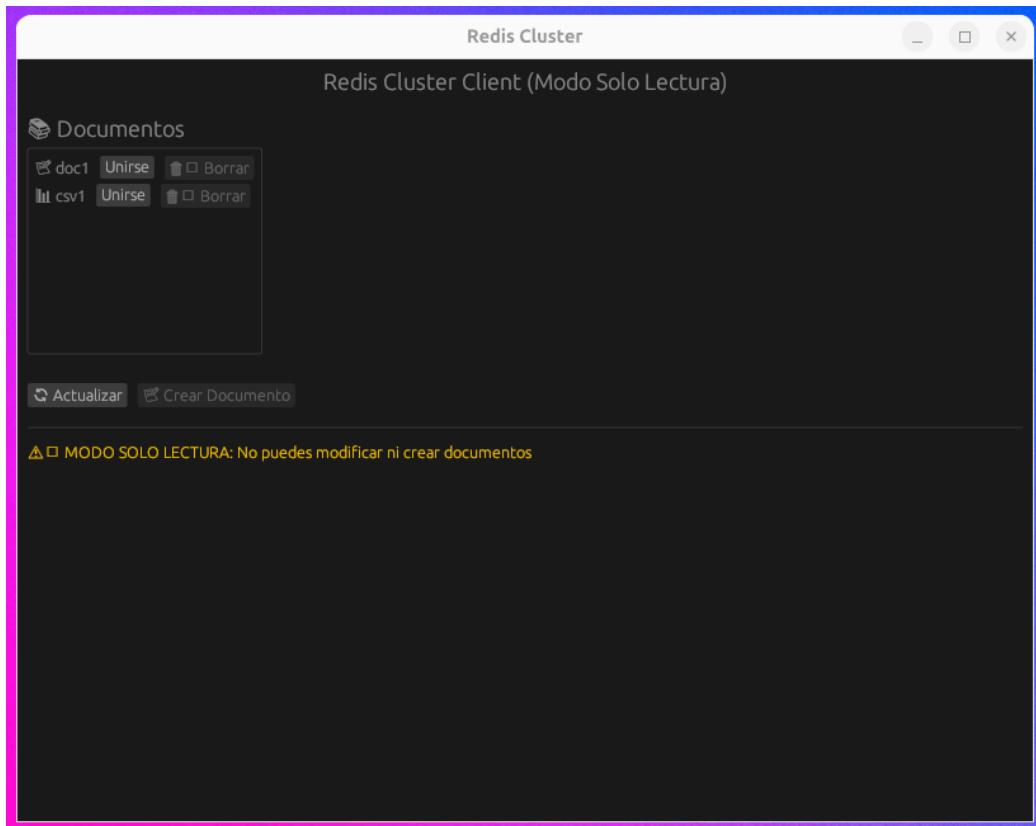
Para la entrega final creamos otra interfaz utilizando las mismas librerías para contemplar el uso de los documentos de texto y planillas de cálculo en entornos multiplataformas.

## **Vistas de usuario**

Para complementar el sistema de autenticación se implementaron 2 tipos de vistas de usuario.

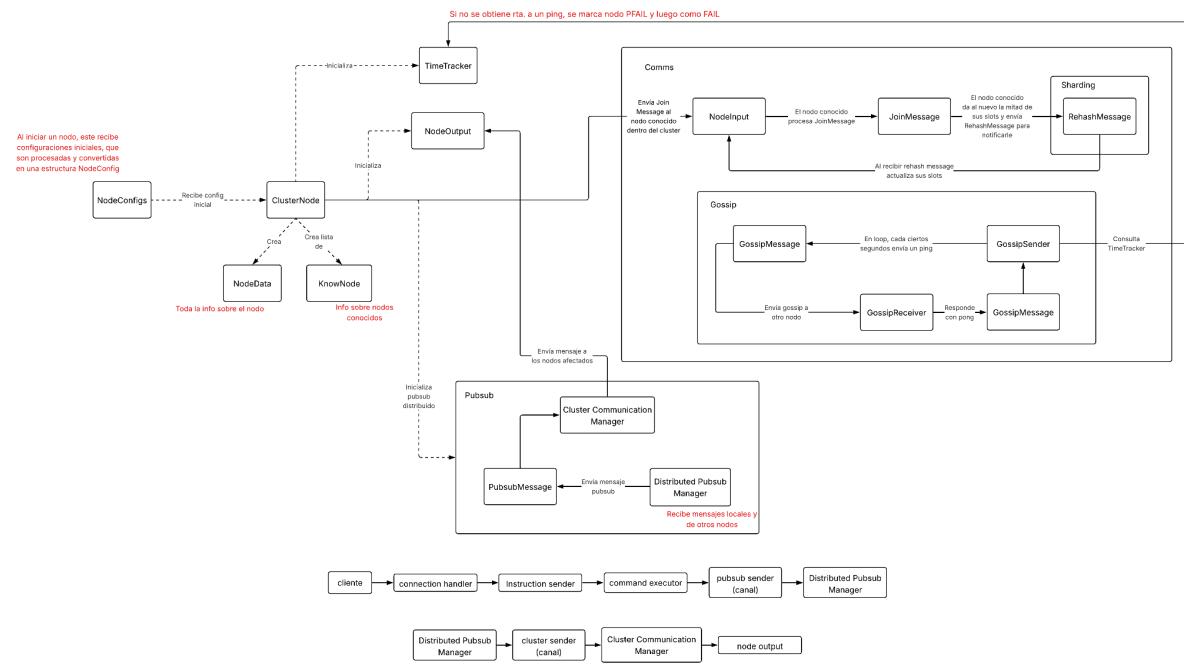
1. Escritura, la vista común, permite el acceso irrestricto a los usuarios a todas las funcionalidades del sistema.
2. Solo lectura, restringe las funcionalidades habilitadas para el usuario a solo permitirle actualizar la lista de documentos y unirse a los mismos sin posibilidad de edición sobre los mismos, solo observando sus cambios (este flujo se triggereá cuando el usuario autenticado es de tipo lectura).

## Interfaz solo lectura



Notar que para un usuario tanto tipo “solo lectura” como usuario “normal” puedan recibir actualizaciones sobre los documentos y sus contenidos, **ambos deben tener accesos a los comandos SUBSCRIBE y PUBLISH** (capturas de pantallas logradas con el usuario “nada”, contraseña “123”).

## Diagrama del clúster



---

## Caso de uso: Documentos compartidos

### Distribución de Información

Para procesar la información que cada cliente agrega al documento de manera paralela, se diseñó un motor de transformación de operaciones basado en un sistema de versionado. Este motor funciona de forma genérica para cualquier tipo de documento o estructura de datos, siempre que cumpla ciertas propiedades fundamentales.

La estructura de datos y operaciones genéricas implica un modelo altamente modular y escalable, que permite asociar fácilmente nuevos tipos de documentos, simplemente hay que implementar un conjunto de operaciones representativas de ese documento sin necesidad de hacer cambios en la infraestructura.

Todo tipo de comunicación entre clientes y servicios se hace mediante el protocolo Pub/Sub implementado, de la misma manera se utiliza la red de redis para persistir los archivos.

### Requisitos de las Operaciones

Una operación representa un cambio puntual en el documento. Las operaciones deben estar correctamente definidas para uno o varios tipos de datos. Para que un conjunto de operaciones se considere válido para un tipo de dato, debe implementar dos traits específicos:

- **Aplicable:** Dado un tipo de dato **D**, la operación debe poder aplicarse sobre la estructura de datos correspondiente.
- **Transformable:** Dadas dos operaciones, debe ser posible transformar una operación en base a la otra, considerando que la operación transformada se realiza posteriormente a la base.

---

\*Al final del informe hay una nota con algunos ejemplos útiles de transformaciones de texto.

## Versionado

Cada operación tiene una versión asociada que indica con qué versión del documento fue emitida. La versión es un identificador único que permite ordenar y sincronizar las operaciones en el tiempo.

Además, cada operación lleva el ID del cliente que la emite y un ID local que representa el número de operación que realizó ese cliente específico.

---

## Microservicio

### Control

El microservicio de control es el componente encargado de llevar el control de versiones y operaciones. Tiene tres atributos fundamentales: el estado actual del documento, la versión actual y el registro de operaciones.

Todas las operaciones pasadas se almacenan en este registro. Cuando llega una nueva operación, se compara su versión con la versión actual del sistema:

- **Si las versiones coinciden**, la operación se aplica directamente al documento. Luego, se actualiza la versión (incrementándose en 1), se registra la operación con su nueva versión y se distribuye el cambio a todos los clientes.
- **Si la versión es anterior a la actual**, significa que la operación fue generada con un estado obsoleto. En este caso, antes de aplicarla, debe ser actualizada.

Supongamos que la versión actual es **a** y se recibe una operación con versión **b**, tal que **a > b**. Entonces:

1. Se toma del registro la operación correspondiente a la versión **b + 1**.
2. Se transforma la operación recibida en base a esa operación.
3. Ahora la versión de la operación pasa a ser **b + 1**.

4. Si sigue siendo menor que **a**, se repite el proceso con **b + 2**, y así sucesivamente.
  5. Este proceso se repite **a - b** veces hasta que la operación tenga versión **a**.
  6. Una vez actualizada, se aplica al estado, se registra con la versión **a + 1** y se distribuye a todos los clientes.
- **Si la versión recibida es superior a la actual**, indica un caso de desincronización entre el cliente y el microservicio, y debe manejarse como un error.
- 

## Persistencia

El servicio también persiste su estado actual automáticamente cada cierta cantidad de operaciones.

---

## Cliente

Cada cliente mantiene su propio estado, la versión asociada a ese estado, un contador local y un registro de operaciones locales que aún no fueron validadas ni versionadas por el servicio de control.

### Aplicación optimista

Cuando el cliente realiza un cambio localmente, este se aplica de inmediato sobre el estado actual. Luego, se registra en el conjunto de operaciones locales y se emite al servicio de control.

Al emitirse, cada operación recibe un ID único asociado.

**Nota:** La versión del documento **no se incrementa** cuando se aplican operaciones localmente. El cliente lleva un seguimiento de todas las operaciones que aplicó pero que el servicio aún no ha validado.

### Operaciones remotas

Cuando se recibe una operación remota desde el servicio de control, pueden darse dos casos:

- **La operación fue emitida por otro cliente:**
  - Se transforma la operación remota en base a todas las operaciones locales.
  - Luego se aplica al estado actual del cliente.
  - Se incrementa la versión del documento.
  - Finalmente, todas las operaciones locales se transforman en base a esta nueva operación.
- **Nota:** La operación remota se transforma con las operaciones locales porque el estado del cliente puede estar parcialmente más avanzado. Luego, las operaciones locales también deben actualizarse para reflejar la nueva versión remota, evitando quedarse atrás.
- **La operación fue emitida por el propio cliente:**
  - Se elimina del registro local (usando su ID local).
  - Se incrementa la versión del documento.

---

## Microservicio - Índice de Documentos

Además del motor de operaciones, el microservicio mantiene un índice de documentos creados. Cada documento tiene asociado su propio servicio de control y funcionan en paralelo.

El microservicio puede crear o eliminar documentos de cualquier tipo. Se considera un tipo de documento como válido siempre que existan operaciones y estructuras de datos asociadas que cumplan con las propiedades mencionadas anteriormente.

Al igual que el servicio de control, el índice de documentos también es persistido automáticamente.

---

## Ejemplos

### Operaciones de texto

Las operaciones implementadas para estructuras de texto son tres:

- **Insertar carácter:** `Insert(pos, char)`
- **Eliminar carácter:** `Delete(pos)`
- **Operación nula:** `NullOp`

Estas operaciones pueden aplicarse sobre la estructura de texto `String`.

### Transformaciones

Algunos ejemplos de cómo funcionan las transformaciones en el caso de las operaciones de texto.

Caso 1:

- `A = Delete(7)`
- `B = Delete(7)`

Al transformar A en base a B, A se convierte en una **operación nula**, ya que el carácter en la posición 7 ya fue eliminado.

Caso 2:

- `A = Insert(9, 'd')`
- `B = Insert(8, 'h')`

Al transformar A en base a B, A se convierte en `Insert(10, 'd')`, ya que la inserción previa en la posición 8 desplazó una posición hacia la derecha.

Caso 3:

- `A = Insert(9, 'd')`
- `B = Insert(14, 'x')`

Al transformar A en base a B, A queda igual (`Insert(9, 'd')`), ya que la inserción posterior no afecta su posición.

Caso 4:

- `A = Insert(9, 'd')`
- `B = Delete(5)`

Al transformar A en base a B, A se convierte en `Insert(8, d)`, porque al eliminar un carácter anterior, todas las posiciones siguientes se desplazan una hacia la izquierda.

## **Implementación Final - LLM SERVICE**

Este servicio actúa como un puente entre nuestro microservicio y el modelo de ia Gemini, permitiendo la generación de texto para poder insertarlo en el documento.

### **1. Inicialización y Conexión**

Al iniciar, el servicio LLM se conecta al microservicio, usando su API KEY.

Se suscribe al canal `LLM_REQUESTS` para recibir solicitudes de generación de texto desde otros componentes del sistema.

### **2. Recepción de Solicitudes**

Cuando recibe una solicitud, la deserializa a una estructura interna (`LLMRequest`).

## **4. Llamada a la API LLM**

El proveedor LLM construye el prompt adecuado según el tipo de tarea (escribir, mejorar o borrar).

Realiza una petición HTTP a la API de Gemini, pasando el prompt.

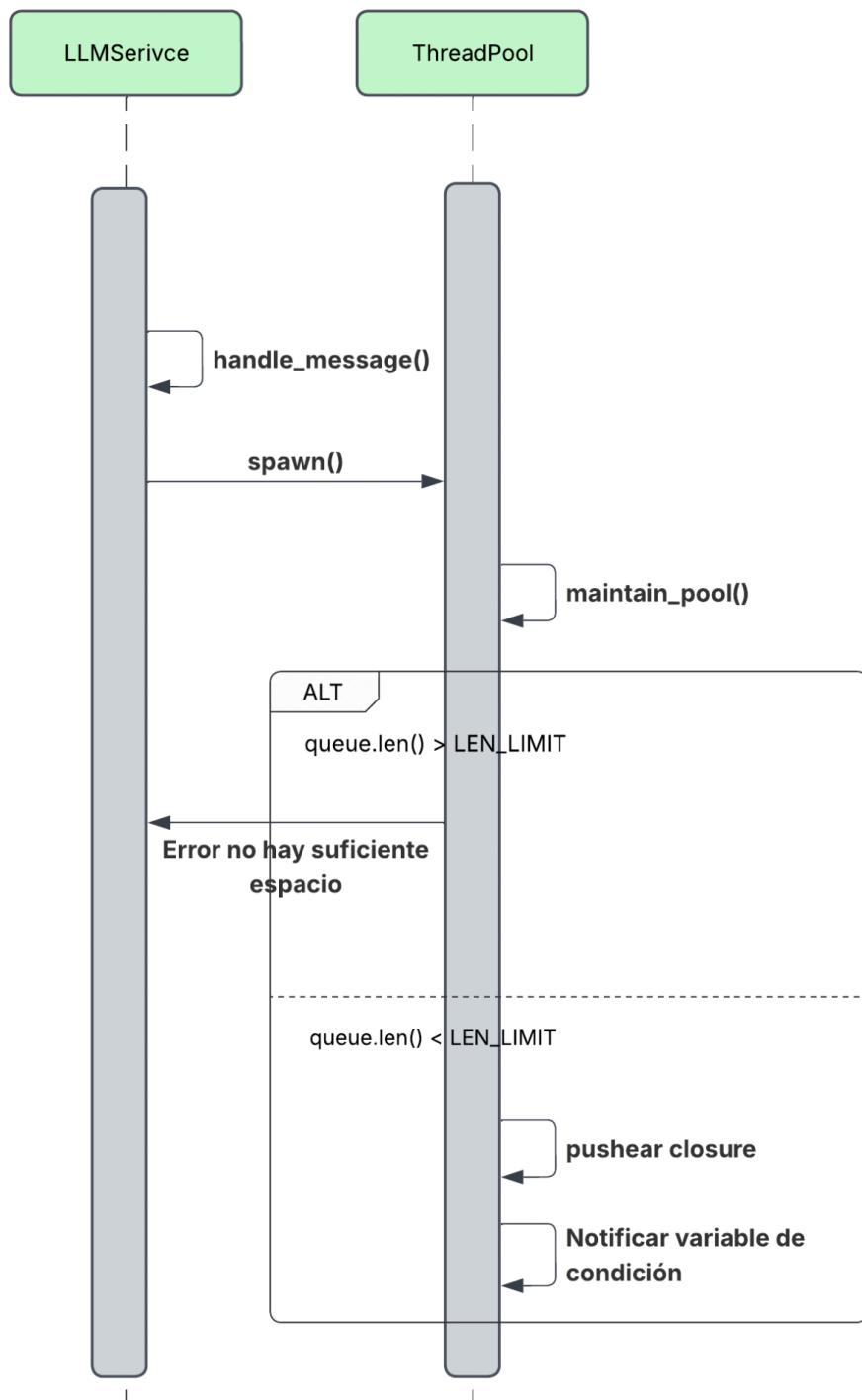
Recibe la respuesta de la API y extrae el texto generado.

### **4.1. Paralelismo**

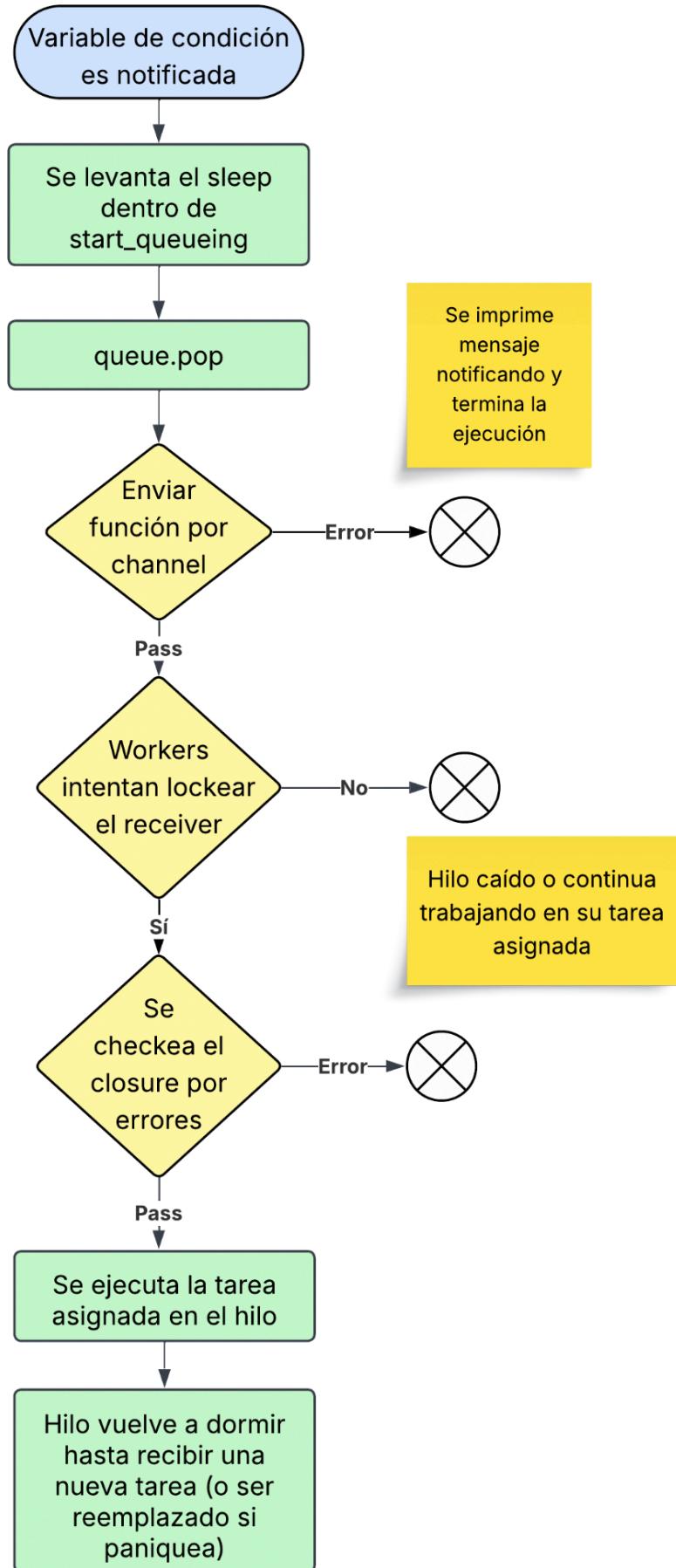
Para dotar al sistema de paralelismo en las solicitudes de la API se utilizó ThreadPool con capacidad de 10 hilos en simultaneos.

Esta estructura cuenta con dos componentes principales:

1. Cola de mensajería, donde se guardan primero las funciones enviadas, cuyo uso triggereá una variable de condición que avisa al hilo que la utiliza que saque su elemento más viejo y lo envíe por el canal a un hilo.
2. Vector de hilos, la estructura central del ThreadPool, es un vector de hilos en donde cada uno escucha continuamente un canal esperando a recibir funciones anónimas, una vez que la reciben las ejecutan y vuelven a quedarse escuchando el canal.



También para dotar a la estructura de mayor robustez se le dio un mecanismo recuperación que se triggereá con cada intento de uso que se le de al ThreadPool, este método (`maintain_pool()`) lo que hace es revisar si hay hilos caídos y los levanta nuevamente.



## **5. Publicación de Respuestas**

El servicio de IA (LLM) solo genera texto a partir de un prompt y devuelve el resultado como un string plano. No realiza ninguna operación de edición ni conoce la estructura interna del documento.

Cuando el usuario acepta una respuesta de la IA, el texto generado se transforma en operaciones de texto que luego son aplicadas al documento.

El microservicio de IA nunca aplica ni envía operaciones de edición; solo responde con texto generado.

Toda la lógica de transformación, inserción, reemplazo o sincronización de texto está en el cliente y en el servicio de control, no en el servicio de IA.

## **Enlace a presentaciones utilizadas**

### **Entrega intermedia:**

[Presentación intermedia - Taller \[Grupo X\] - Genially](#)

### **Entrega final:**

[Presentación final - Taller \[Grupo X\] - Genially](#)

### **Final de la materia:**

[RustiDocs Final - Taller \[GrupoX\] | Genially](#)