.UBAfiuba
FACULTAD DE INGENIERÍA

Taller de Programación

(TA045) Taller de Programación I - Cátedra Deymonnaz

Proyecto: RustiDocs - 1C 2025

28 de julio de 2025

Figueroa, Camila
111204

Fu, Anibal
111206

Pazos, Giuliana
111268

Villegas, Tomás
106456

1. Introducción

En el presente informe se detallan los aspectos fundamentales del proyecto RusticDocs. Donde se incluiran diagramas de secuencia de las operaciones más relevantes, diagrama de componentes y módulos de la arquitectura general del diseño desarrollado acompañados de la explicación respectiva.

1.1. Contexto del Proyecto

La Universidad de Buenos Aires se encuentra en proceso de desarrollar un sistema de colaboración en línea que permita la creación y edición de documentos de texto y planillas de cálculo en tiempo real. Este sistema estará disponible para toda la comunidad de la UBA, así como también para usuarios de otras universidades a nivel mundial.

Debido a la alta demanda esperada y a la amplia distribución geográfica de sus usuarios, se requiere un diseño escalable y tolerante a fallos que garantice un rendimiento óptimo bajo carga intensa. Luego de analizar diferentes alternativas tecnológicas, se ha decidido construir el sistema sobre una arquitectura distribuida basada en *Redis Cluster*, utilizando esta tecnología tanto para el almacenamiento persistente de datos como para el intercambio de mensajes a través del mecanismo de *Publish/Subscribe*.

1.2. Objetivo del Proyecto

El objetivo principal es desarrollar una implementación en el lenguaje *Rust* de un cluster de Redis que respete su protocolo cliente-servidor y que contemple un protocolo interno de intercambio de mensajes entre nodos.

Como objetivo secundario, se busca que este proyecto funcione como una experiencia real de desarrollo de software de mediana envergadura, promoviendo buenas prácticas de ingeniería, entregas parciales y revisiones periódicas.

1.2.1. Requerimientos

- Una implementación del protocolo de Redis (*RESP*) y comandos para manejar cadenas de texto, listas y conjuntos, con almacenamiento persistente en disco.
- Soporte completo para el paradigma *Pub/Sub* que permita el intercambio de mensajes entre clientes sin acoplamiento directo.
- Una arquitectura distribuida con al menos tres nodos *master*, cada uno con dos réplicas, incluyendo mecanismos de detección de fallos basados en *gossip protocol* y promoción de réplicas.
- Configuración del servidor mediante archivos `redis.conf` y generación de logs sin mantener un manejador de archivo global.
- Implementación de medidas de seguridad que incluyan autenticación, autorización y cifrado de datos en tránsito.
- Una aplicación cliente con interfaz gráfica para edición colaborativa de documentos en tiempo real. Esta aplicación se comunicará con el cluster utilizando el protocolo de Redis y hará uso de Pub/Sub para sincronizar los cambios entre usuarios.
- Un microservicio de control y persistencia que funcionará como cliente del cluster y se encargará de gestionar las sesiones colaborativas y persistir el estado de los documentos de manera periódica.

1.3. Enfoque de Desarrollo e Investigación

A lo largo de este trabajo se realizó una **investigación exhaustiva y continua** centrada en la arquitectura de *Redis Cluster*, su protocolo *RESP* y los mecanismos nativos de *Pub/Sub*. La estrategia adoptada fue **iterativa e incremental**: antes de avanzar en cada módulo se identificaron los conceptos clave y se estudiaron fuentes primarias (documentación oficial de Redis) y materiales complementarios (artículos y videos) de internet.

Este enfoque práctico permitió:

- **Despejar desafíos específicos a medida que surgían** obtener **feedback** lo mas rapido posible.
- **Validar las decisiones de diseño en entornos controlados** mediente la implementacion de funcionalidades basicas, de cada modulo, que permitian documentar de forma experimental posibles desiciones finales a llevar a cabo.
- **Iterar rápidamente** permitiendo que cada nueva funcionalidad se integre sin comprometer la consistencia del proyecto y minimizando los conflictos .

Gracias a este proceso de investigación incremental, el proyecto alcanzó un equilibrio entre **rigor teórico** y **aplicación práctica**, logrando una versión funcional de Redis Cluster en Rust que respeta el protocolo cliente-servidor, implementa replicación y particionado, ofrece Pub/Sub distribuido y cumple con los requisitos de escalabilidad, disponibilidad y seguridad exigidos. Que luego fue utilizado para la implementacion de una aplicacion de mediana escala **RusticDocs**.

2. Cliente-Servidor

2.1. Modelo Cliente-Servidor

El modelo cliente-servidor adoptado en esta implementación se basa en una arquitectura concurrente orientada a la eficiencia y a la separación de responsabilidades. A continuación, se describe el flujo general de procesamiento de comandos y respuestas entre clientes y nodo. Figura 1:

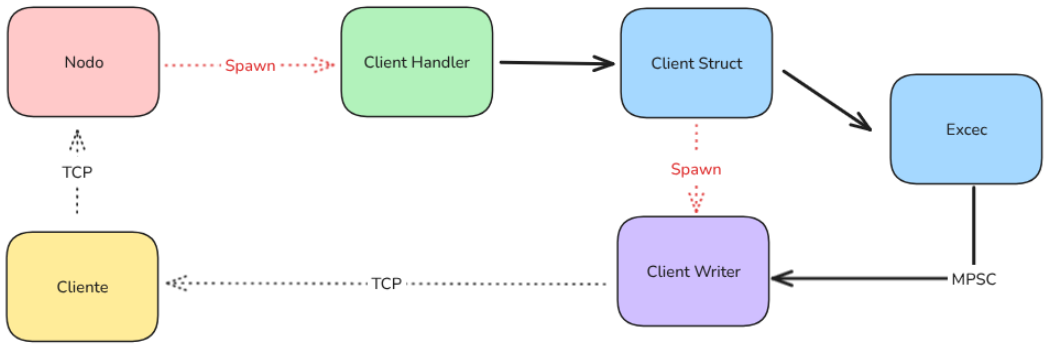


Figura 1: Estructura general del modelo Cliente - Servidor

Cuando un cliente se conecta a un nodo del cluster, dicho nodo crea un hilo dedicado para gestionar la conexión entrante. En ese contexto, se instancia una estructura de tipo `Client`, la cual encapsula toda la información relevante del cliente, incluyendo su identificador, dirección IP y la conexion TCP asociada.

Para lograr una separación clara de responsabilidades entre lectura y escritura, el stream TCP se clona en dos extremos independientes: uno exclusivamente destinado a la lectura de comandos y

otro al envío de respuestas. Además, se lanza un hilo adicional llamado **hilo escritor**, responsable únicamente de enviar respuestas al cliente.

El flujo de ejecución es el siguiente:

- El **handler del cliente** se encarga de leer comandos desde el stream TCP de lectura.
- Una vez recibido un comando, este es procesado por el módulo `Ex-ec`, el cual genera una respuesta en formato RESP.
- En lugar de escribir directamente al socket desde el módulo de ejecución, la respuesta generada se envía mediante un *channel* al hilo escritor del cliente correspondiente.
- Este hilo escritor toma la respuesta desde el canal y la transmite al cliente a través del stream de escritura TCP.

Este enfoque presenta varias ventajas:

- Centraliza la escritura hacia el cliente, evitando condiciones de carrera y mejorando la coherencia del flujo de salida.
- Mejora el rendimiento al reducir bloqueos innecesarios en distintas partes del sistema que intenten escribir simultáneamente.
- Facilita la incorporación de funcionalidades adicionales como logs, monitoreo o compresión de respuestas, al tener un único punto de escritura.

Este modelo es especialmente útil en escenarios donde múltiples subsistemas pueden generar respuestas (como comandos comunes y mensajes del sistema `Pub/Sub`), ya que garantiza un canal unificado de salida y reduce significativamente el acoplamiento entre componentes.

3. Protocolo de comunicacion Cliente - Servidor

3.1. Implementación del protocolo RESP

Se implementa el protocolo RESP (REdis Serialization Protocol), el cual es el formato estándar utilizado por Redis para la comunicación entre cliente y servidor. RESP está diseñado para ser eficiente, fácil de implementar y humanamente legible. Se han implementado completamente los tipos RESP:

- `SimpleString`: Representa una cadena simple RESP. Ejemplo: `+OK\r\n`
- `SimpleError`: Representa un mensaje de error simple. Ejemplo: `-Error message\r\n`
- `Integer`: Representa un número entero. Ejemplo: `:1000\r\n`
- `BulkString`: Representa una cadena con longitud prefijada. Ejemplo: `$6\r\nfoobar\r\n`
- `Arrays`: Representa una lista ordenada de elementos RESP.
  Ejemplo: `*2\r\n$3\r\nGET\r\n$3\r\nkey\r\n`
- `Set`: Representa un conjunto no ordenado de elementos únicos.
  Ejemplo: `3\r\n$3\r\naaa\r\n$3\r\nbbb\r\n$3\r\nccc\r\n`
- `Null`: Representa un valor nulo. Ejemplo: `_\r\n`
- `Map`: Representa un diccionario clave-valor.
  Ejemplo: `%2\r\n$3\r\nkey\r\n$5\r\nvalue\r\n$4\r\nname\r\n$5\r\nAlice\r\n`

La codificación y decodificación de mensajes RESP se realiza de forma síncrona en los extremos del canal TCP, permitiendo una interacción rápida y determinista.

A nivel de código, cada tipo de dato se representa mediante un `struct`, el cual concentra la capacidad de acceso y modificación de la información. Esto permite encapsular la implementación interna de los mismos y posibilitar cambios (por ejemplo, utilizar listas enlazadas para el tipo `Array`) sin requerir modificaciones en el resto del programa, de forma transparente.

También se define un `enum` llamado `DatoRedis` y un `trait` denominado `TipoDatoRedis`. A través de estos, se logra mantener un alto nivel de abstracción y generalidad al operar con los datos, tanto desde el cliente como desde el nodo.

En el `trait TipoDatoRedis`, se incluyen las funciones `convertir_a_protocolo_resp(&self) ->String`, que genera la representación RESP de un dato, y `convertir_resp_a_string(&self) ->String`, que retorna un `String` representando el `struct` de forma más intuitiva para el cliente (es decir, en un formato más comprensible que el protocolo RESP).

Figura 2: convertir_a_protocolo_resp en TipoDatoRedis

Figura 3: convertir_resp_a_string en TipoDatoRedis

Por otro lado, el `enum DatoRedis` implementa *wrappers* de las funciones de creación y serialización/deserialización (`to_bytes` y `from_bytes`) para mantener la abstracción.

De esta manera, se utiliza polimorfismo para facilitar la interacción de ambas partes con los datos y el protocolo, permitiendo incorporar nuevos tipos de datos fácilmente, sin requerir modificaciones en el código existente.

3.2. Comunicación cliente-servidor mediante el protocolo

Para la comunicación cliente-servidor, se tienen también 4 funciones principales encargadas de la serialización y deserialización:

- `resp_client_command_write(comando: String, stream: &mut dyn Read) ->Result<(), DatoRedis>`: Utilizada por el driver de Redis implementado, a partir de un comando en texto plano (`String`), envía su parametrización a protocolo RESP por el `stream` recibido por parámetro.

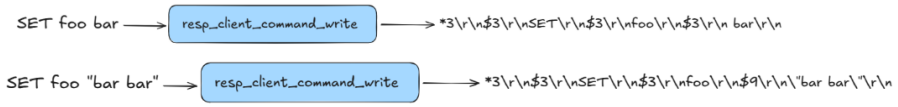


Figura 4: Escritura cliente-servidor

- `resp_server_command_read(stream: &mut dyn Read) ->Result<Vec<String>, DatoRedis>`: Luego de que el cliente envíe un comando mediante `resp_client_command_write`, el servidor lo obtiene mediante esta función, retornando la representación como vector de cadenas (`Vec<String>`).

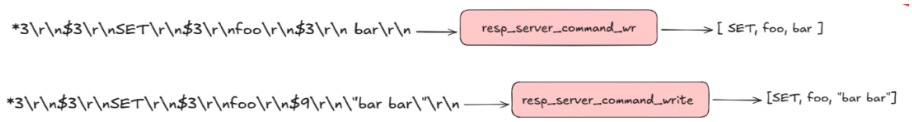


Figura 5: Lectura servidor-cliente

- `resp_server_command_write(retorno: String, stream: &mut dyn Read) ->Result<(), DatoRedis>`: Envía por el `stream` recibido la representación RESP de un comando, para ser leída por el cliente.

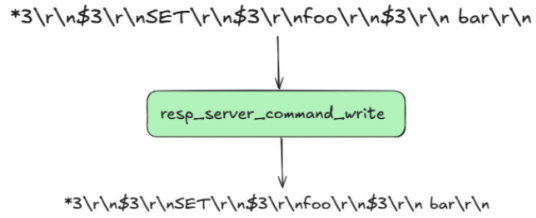


Figura 6: Escritura servidor-cliente

- `resp_client_command_read(stream: &mut dyn Read) ->Result<DatoRedis, DatoRedis>`: Recibe un dato Redis en formato RESP por el `stream`, decodificándolo a un `struct` del `enum DatoRedis`.

Figura 7: Lectura cliente-servidor

4. Comandos Redis

Durante el desarrollo de nuestro trabajo, trabajamos en la implementación de un conjunto de comandos fundamentales de Redis, cubriendo estructuras como strings, listas y sets, además del soporte básico para el sistema de pub/sub.

4.1. Strings

Para la estructura de strings, se desarrollaron los siguientes comandos:

```
GET, SET, DEL, GETDEL, APPEND, STRLEN, SUBSTR, INCR, DECR
```

Estos comandos permiten operaciones típicas como lectura, escritura, modificación y manipulación de enteros.

4.2. Listas

En el caso de las listas, se implementaron:

`LMOVE, LREM, LINDEX, LTRIM, LSET, LRANGE, RPOP, LPOP, LLEN, RPUSH, LPUSH, LINSERT`

Estas funciones permiten desde operaciones básicas de push/pop hasta manejo más avanzado como mover elementos entre listas, obtener rangos o insertar en posiciones específicas.

4.3. Sets

Los comandos disponibles para sets incluyen:

`SREM, SMEMBERS, SISMEMBER, SADD y SCARD.`

Con ellos se puede administrar conjuntos sin elementos duplicados, validar pertenencia y obtener cardinalidad, entre otras operaciones típicas de la estructura.

4.4. Pub/Sub

También se incorporó soporte básico para el modelo de pub/sub. Los comandos implementados son:

`SUBSCRIBE, UNSUBSCRIBE, PUBLISH, PUBSUB NUMSUB y PUBSUB CHANNELS.`

Esto permite a los clientes suscribirse a canales, recibir mensajes en tiempo real y consultar el estado de suscripciones activas. Todos los comandos fueron diseñados como funciones que reciben una instancia del storage central del sistema, lo que permite acceder de forma encapsulada a las operaciones `get`, `set` y `del`. Esto facilita el control del estado interno de la base de datos y nos permite serializar fácilmente las respuestas, encapsuladas en un `Result<DatoRedis, DatoRedis>`.

En resumen, esta capa de comandos fue diseñada para ser flexible, mantenible y extensible. Nos permitió tener una base sólida sobre la cual seguir construyendo funcionalidades más avanzadas, como persistencia en disco, replicación entre nodos o tolerancia a fallos.

5. Almacenamiento de Datos

Para manejar los datos dentro del sistema, desarrollamos una estructura central llamada `Storage`, integrada en cada nodo. Esta estructura está a cargo de administrar las claves que le corresponden según su rango de slots, siguiendo el criterio de particionamiento de Redis Cluster. Internamente, el `Storage` mantiene un hashmap de `(slot, hashmap<clave, DatoRedis>)`, lo que permite distribuir y organizar los datos de forma eficiente según el slot asignado. Desde los comandos, se interactúa con este almacenamiento mediante funciones públicas bien definidas: `get`, `set` y `del`. Esto nos permite abstraer la lógica de acceso a los datos y mantener el código modular y fácil de testear.

5.1. Persistencia

Para asegurar que los datos no se pierdan ante reinicios o fallos, implementamos dos mecanismos de persistencia configurables: **AOF** (Append Only File) y **RBD** (dump binario periódico). Ambos se configuran desde el archivo de configuración del nodo, lo que permite flexibilidad según las necesidades del entorno.

5.1.1. AOF::Append Only File

El modo **AOF** permite guardar todas las operaciones mutables ejecutadas sobre el almacenamiento. Si en el archivo de configuración se indica `appendonly = yes` y se especifica el path con `aof_file`, entonces el sistema irá registrando cada operación que modifica el estado (SET, DEL, etc.) en ese archivo. Al reiniciar el nodo, el sistema simplemente reejecuta todas esas operaciones en orden, reconstruyendo el estado previo del `Storage`. Esta estrategia es útil porque asegura una reconstrucción exacta del historial de comandos, pero a cambio tiene un mayor costo computacional al momento del arranque.

5.1.2. RDB::Redis Data Base

Como alternativa más liviana, implementamos también persistencia por volcado periódico del estado completo del `Storage` en formato binario (**RBD**). Para esto, se configura el valor de `save` (expresado en milisegundos) y el path con `storage_file`. Por ejemplo, si `save = 60000`, cada 60 segundos el sistema guarda una imagen completa del almacenamiento en un archivo binario. Esta estrategia permite una recuperación mucho más rápida, ya que al reiniciar el nodo se carga directamente ese snapshot en memoria, sin necesidad de volver a ejecutar comandos uno por uno.

5.1.3. Metadata persistente

Además, agregamos una capa de persistencia para ciertos datos de configuración del nodo que no deberían cambiar entre ejecuciones, como el `node_id`. Esto lo guardamos en un archivo de metadata, el cual se especifica en el archivo de configuración y se verifica al momento de reiniciar un nodo. De esta forma, no solo simplificamos la restauración del estado, sino que también podemos detectar errores o inconsistencias, como intentos de levantar múltiples veces el mismo nodo, cambios no permitidos o problemas en la red del clúster. graphicx listings xcolor

6. Sistema Publisher Subscriber

6.1. ¿Por qué *Pub/Sub*?

El paradigma de publicación y suscripción (*Pub/Sub*) es fundamental para el desarrollo de sistemas colaborativos que requieren intercambio de información en tiempo real entre múltiples participantes. Redis proporciona soporte nativo para este modelo, permitiendo implementar de manera eficiente mecanismos de difusión de eventos sin acoplamiento directo entre los emisores y los receptores de mensajes.

Este enfoque favorece la escalabilidad y la modularidad del sistema, ya que los emisores de eventos no necesitan conocer a los receptores, ni mantener conexiones directas entre sí. Además, la estructura de los canales puede ser utilizada para organizar los mensajes por tipo de recurso, categoría temática o contexto de colaboración.

El uso de *Pub/Sub* en Redis como capa de comunicación en tiempo real proporciona una base robusta y eficiente para la sincronización distribuida de cambios en entornos colaborativos.

6.2. Implementacion

Se adopta como base el modelo de **Message Broker**, en el cual una entidad central actúa como intermediario entre los publicadores y los suscriptores. Esta entidad es responsable de recibir los mensajes publicados por los clientes, identificar a los suscriptores interesados según el canal correspondiente, y reenviar los mensajes a dichos suscriptores de forma eficiente y desacoplada. De esta manera, se logra una arquitectura en la que los emisores y receptores de información no requieren conocerse entre sí, lo que favorece la escalabilidad y flexibilidad del sistema. Como se

muestra en la Figura 20, el Message Broker actúa como intermediario para la ejecucion de comandos especificos del tipo *Pub/Sub*.

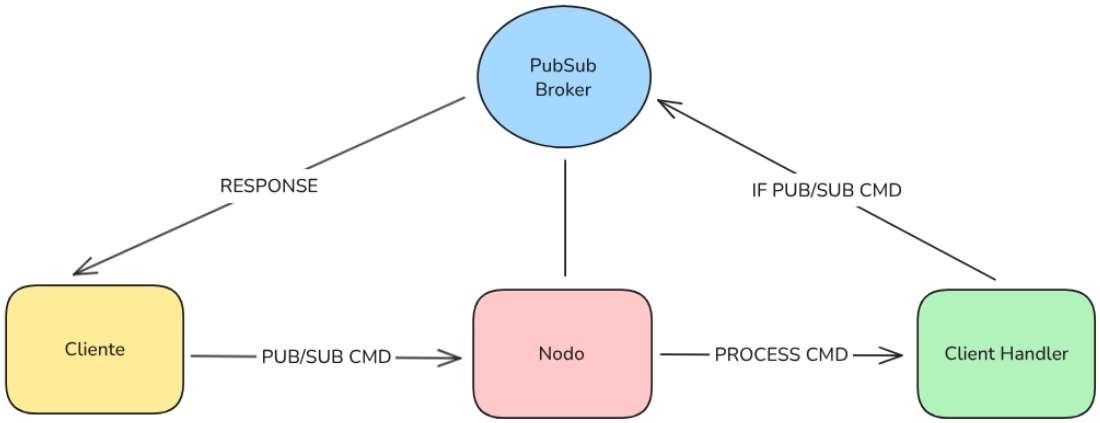


Figura 8: Ejemplo de arquitectura basada en Message Broker

6.2.1. Estructura interna

Cada nodo cuenta con su propio módulo **PubSub Broker**, responsable de gestionar la lógica relacionada al sistema de publicación y suscripción. La Figura 9 ilustra la estructura general del componente.

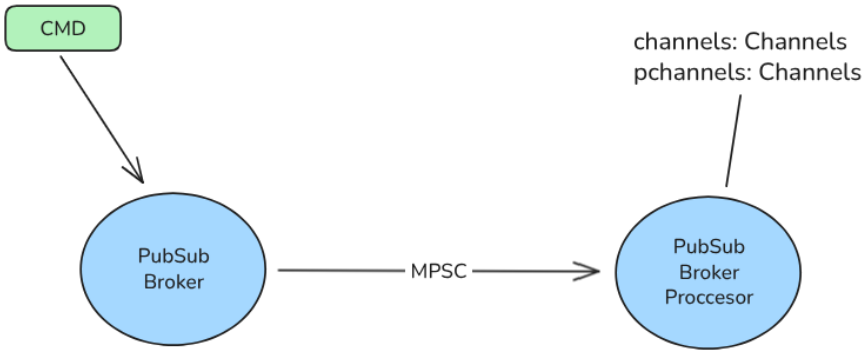


Figura 9: Estructura interna del módulo PubSub Broker

Cuando un cliente emite un comando correspondiente al paradigma *Pub/Sub*, este es derivado automáticamente al **PubSub Broker**. Internamente, dicho broker delega la ejecución al *PubSub Broker Processor*, un hilo dedicado exclusivamente al procesamiento eficiente de estos comandos.

Con el fin de mantener un registro de los canales activos, se utilizan dos estructuras principales:

- `channels`: para gestionar el sistema *Pub/Sub* clásico.
- `pchannels`: para el sistema de suscripción por patrones.

Ambas estructuras tienen el siguiente tipo:

```
type Canal = String;
type Channels = HashMap<Canal, HashMap<String, Sender<String>>>;
```

En estas estructuras, cada canal está representado por una clave de tipo `String`, y contiene un mapa asociado de clientes suscritos. Cada cliente es identificado por un `ID` único y vinculado

a un objeto `Sender<String>` que permite enviarle mensajes en tiempo real de manera eficiente. Este sender hace referencia al hilo escritor de cada cliente, viste en el apartado de modelo cliente servidor.

Se ha mencionado que momento de recibir una nueva conexión de un cliente, se instancia una estructura `Client` asociada (ver apartado Sección 2.1). Esta estructura contiene, entre otros, dos atributos clave que participan directamente en el modelo de *Pub/Sub*:

```
struct Client {
    ...
    channels: HashSet<Canal>,
    pchannels: HashSet<Canal>,
    ...
}
```

Estas estructuras permiten mantener un registro eficiente de los canales a los cuales el cliente está suscripto, tanto para suscripciones exactas (`channels`) como para suscripciones por patrones (`pchannels`).

Este diseño facilita significativamente la gestión de eventos como la desuscripción masiva (por ejemplo, al ejecutar un comando `UNSUBSCRIBE` sin parámetros) y la desconexión del cliente. En estos casos, contar con un registro local de los canales suscriptos permite realizar una limpieza eficiente y consistente de las referencias en el `PubSub Broker`, evitando inconsistencias o referencias colgantes.

7. Redis Cluster

7.1. Main Components

En esta sección se detalla la implementación de las funcionalidades fundamentales que conforman el Redis Cluster. El diseño sigue los lineamientos presentados en la documentación oficial, con especial énfasis en los aspectos críticos de un sistema distribuido moderno, como la distribución de claves, la replicación de datos y la tolerancia a fallos.

Se aborda la arquitectura basada en nodos *master* y *replica*, incluyendo los mecanismos de *replica promotion* y el protocolo de comunicación entre nodos. Asimismo, se implementa un protocolo de tipo *gossip* para la detección de fallos y la actualización de la topología del cluster.

Además, se incluye soporte para el paradigma *Pub/Sub* a nivel distribuido, permitiendo la propagación eficiente de mensajes entre clientes conectados a diferentes nodos del cluster. De forma opcional, se considera la extensión a un modelo de *Sharded Pub/Sub* para mejorar la escalabilidad horizontal.

7.2. Distribución de Claves

El sistema distribuido está diseñado para operar con un total de 9 nodos, organizados en una arquitectura de tres nodos *master*, cada uno acompañado por dos réplicas, formando así tres ternas de nodos (master + 2 replicas). Figura 16. Esta organización responde a una estrategia de particionado horizontal, donde el espacio total de claves se divide en **16.384 slots**, tal como lo especifica el protocolo de Redis Cluster.

Cada nodo *master* es responsable de una porción equitativa de estos slots, es decir, aproximadamente 5.462 slots por nodo, permitiendo una distribución balanceada de la carga tanto en almacenamiento como en procesamiento de comandos. Esta segmentación tiene como objetivo principal facilitar la escalabilidad del sistema, ya que nuevos nodos pueden integrarse al cluster redistribuyendo slots sin afectar la operatividad general.

Las réplicas, por su parte, no participan activamente en el procesamiento de comandos de escritura o asignación de slots, pero mantienen una copia actualizada del estado de su nodo *master* correspondiente. En caso de falla del nodo *master*, una de sus réplicas puede ser promovida rápidamente, garantizando así disponibilidad y continuidad del servicio (ver Sección 7.8).

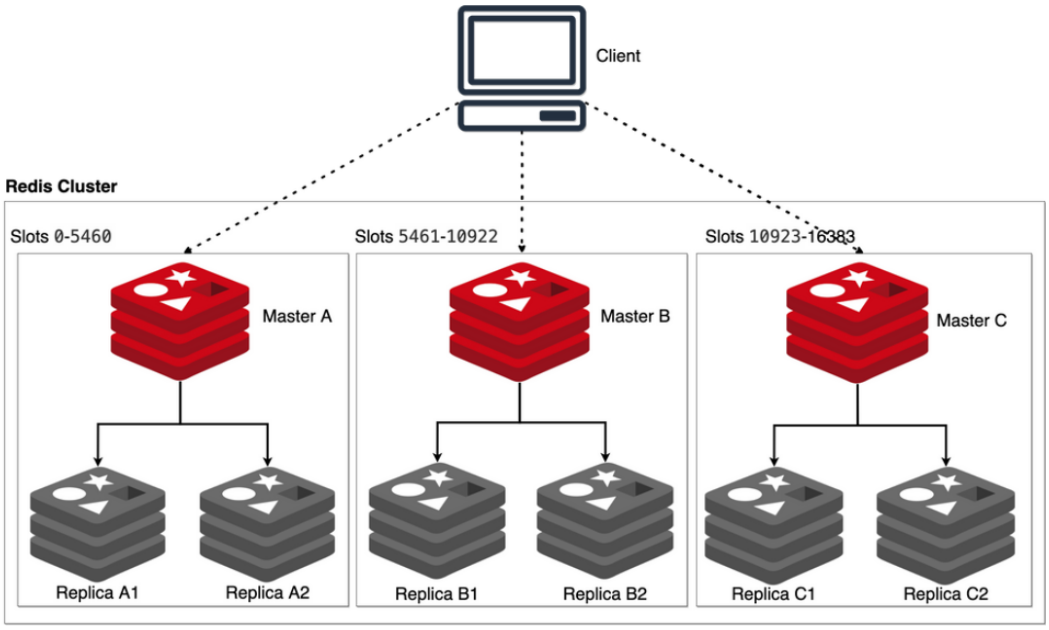


Figura 10: Esquema general del Cluster

La asignación inicial de slots y la distribución de nodos se realiza mediante un archivo de configuración (ver Sección 8), y cada nodo mantiene información actualizada sobre la topología del cluster, incluyendo qué slots pertenecen a qué nodos, a través de un mecanismo de intercambio periódico de información (ver Sección 7.6).

Este esquema de distribución permite que las consultas de lectura/escritura sean dirigidas al nodo correspondiente con mínima latencia y sin necesidad de una coordinación centralizada, cumpliendo así con los principios de escalabilidad y eficiencia requeridos por el sistema.

7.3. Protocolo de Comunicación Inter-Nodo

La comunicación entre los distintos nodos del clúster se implementa mediante un protocolo binario propio **RIP** (**R**edis **I**nternal **P**rotocol), diseñado específicamente para intercambiar mensajes internos de forma eficiente y extensible. Cada mensaje transmitido entre nodos se representa mediante la estructura `ClusterMessage`, compuesta por dos secciones principales: el encabezado (`MessageHeader`) y el cuerpo del mensaje (`ClusterMessagePayload`).

El encabezado (`MessageHeader`) incluye metainformación esencial del nodo emisor, como su identificador único, el rango de *hash slots* que administra, su estado dentro del clúster, información de época (*epoch*) para sincronización de configuraciones, y sus direcciones de red tanto para clientes como para otros nodos del clúster. Esto permite a los nodos receptores interpretar correctamente el contexto del mensaje y tomar decisiones informadas respecto al estado global del clúster.

La sección de `payload` (`ClusterMessagePayload`) representa una parte variable del mensaje, cuya semántica depende del tipo de mensaje transmitido. Este diseño modular permite encapsular distintos tipos de información según el propósito del mensaje. Entre los tipos de payload soportados se encuentran:

- `Gossip`: utilizado para compartir información sobre otros nodos del clúster, como parte del mecanismo de detección de fallos.
- `Fail`: notificación sobre un nodo que ha sido considerado como fallido.
- `RedisCommand` y `PubSub`: permiten la propagación de comandos internos del protocolo Redis,

incluyendo aquellos asociados al sistema *Pub/Sub*.

- `FailAuthReq` y `FailAuthAck`: utilizados en el proceso de *replica promotion*, permitiendo a las réplicas solicitar y recibir votos de los nodos *master*.
- `FailNegotiation`: contiene información de *replication offset* para coordinar la promoción de una réplica.
- `Meet` y `MeetMaster`: utilizados durante el descubrimiento inicial de nodos en el clúster y posteriormente nuevas actualizaciones de estado de nodos, por ejemplo, debido a la promocion de replicas.
- `Empty`: reservado para mensajes que no requieren datos adicionales.

Este esquema de comunicación garantiza que todos los mensajes intercambiados entre nodos contengan tanto la información del emisor como una sección específica que define la acción a tomar, facilitando la implementación de funcionalidades clave como detección de fallos, replicación, sincronización de estado y publicación de eventos.

Para soportar este protocolo binario interno, se definieron los traits `SerializeRIP` y `DeserializeRIP`, los cuales permiten serializar y deserializar estructuras de datos en una secuencia de bytes compatible con el formato del protocolo. De esta forma, cada tipo de mensaje puede ser convertido a su representación binaria y reconstruido desde un stream, facilitando la transmisión y recepción eficiente entre nodos del clúster.

7.4. Cluster Bus y Topología de Red

Cada nodo del clúster expone un puerto adicional exclusivo para la comunicación interna entre nodos, conocido como **Cluster Bus**. Este puerto se obtiene sumando `10000` al puerto de datos.

La comunicación entre nodos se realiza exclusivamente a través del *Cluster Bus*, utilizando un protocolo binario interno.

En cuanto a la topología, Redis Cluster establece una red de tipo **malla completa** (*full mesh*), donde cada nodo mantiene conexiones TCP persistentes con todos los demás nodos del clúster. En un clúster de N nodos, cada nodo mantiene $N - 1$ conexiones salientes y $N - 1$ conexiones entrantes.

7.4.1. Estructura interna de un Nodo dentro del cluster

Desde la perspectiva de un nodo individual dentro del clúster, el diseño adoptado se basa en la concurrencia mediante la creación de múltiples hilos, donde cada uno cumple un rol específico en el procesamiento y mantenimiento de las tareas del clúster. En particular, cada nodo lanza cinco hilos dedicados exclusivamente al manejo de las responsabilidades internas del clúster. Figura 11:

- **HandlerConnectionIncoming**: este hilo es responsable de aceptar y gestionar las conexiones entrantes provenientes de otros nodos del clúster. Actúa como el punto de entrada para la comunicación entre nodos.
- **Hilo de Lectura Periódica**: encargado de escanear todos los streams de conexión abiertos con otros nodos, en ciclos periódicos con un retardo fijo (100 ms) para evitar espera activa (*busy waiting*). Cuando se detecta un mensaje entrante, este se reenvía a través de un canal MPSC al hilo correspondiente para su procesamiento.
- **Procesador de Mensajes y Escritura**: recibe mensajes del hilo lector, los interpreta según su tipo (por ejemplo, `Gossip`, `Fail`, `Update`, etc.) y ejecuta las acciones necesarias. Si el protocolo lo requiere, también se encarga de generar respuestas y enviarlas a los nodos correspondientes.

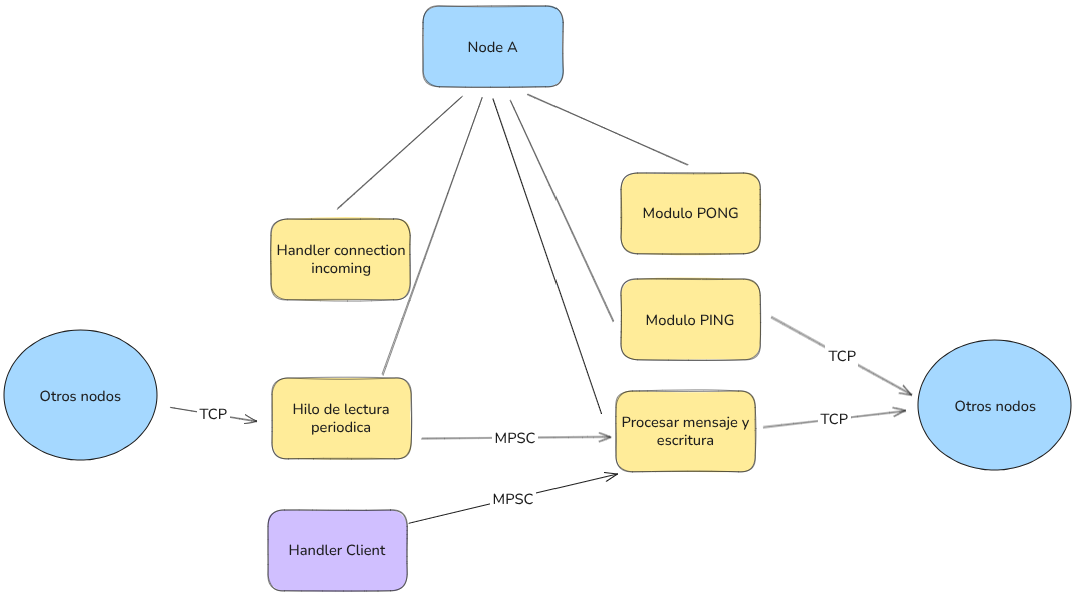


Figura 11: Nodo en el cluster

- **Módulo PING**: ejecuta periódicamente el envío de mensajes `PING` hacia un subconjunto aleatorio de nodos. Esta funcionalidad es esencial para la implementación del protocolo de *Gossip* y la detección de fallos.
- **Módulo PONG**: realiza monitoreos periódicos (cada 100 ms) para detectar nodos potencialmente caidos. Verifica que, tras enviar un `PING`, se reciba una respuesta `PONG` dentro del tiempo esperado (`NODE TIMEOUT`). Si no se obtiene respuesta dentro del umbral definido, el nodo en cuestión es marcado como en estado de `PFAIL`.

Este enfoque modular y concurrente permite distribuir claramente las responsabilidades de cada nodo dentro del clúster, logrando un comportamiento reactivo, eficiente y mantenible.

7.4.2. Estructuras compartidas entre hilos del clúster

Para coordinar adecuadamente las tareas concurrentes en un nodo del clúster, se mantienen ciertas estructuras de datos compartidas entre los distintos hilos responsables de la lógica interna del clúster. En particular, se utilizan tres estructuras principales, todas implementadas como diccionarios (`HashMap`), que comparten un esquema común: utilizan el identificador único del nodo remoto (`NodeId`) como clave.

- **Nodos conocidos:** esta estructura almacena como valor una estructura que representa información relevante de cada nodo conocido por el nodo actual (por ejemplo, estado del nodo, tiempo del último `PING`/`PONG`, etc.). Esta información se actualiza dinámicamente como resultado del intercambio de mensajes tipo `PING`, `PONG` y `MEET`.
- **Streams de lectura:** esta estructura mantiene una referencia al stream de lectura correspondiente a cada nodo con el que se ha establecido una conexión. Es utilizada por el hilo encargado de realizar la lectura periódica desde los nodos vecinos del clúster.
- **Streams de escritura:** análoga a la anterior, pero destinada exclusivamente a los streams de escritura. Es utilizada por el hilo responsable de enviar mensajes a los otros nodos.

Durante el proceso de *handshake* entre nodos —específicamente, al recibir un mensaje de tipo `MEET`— el stream entrante se clona, generando así dos manejadores independientes: uno para lectura

y otro para escritura. Esta duplicación permite separar responsabilidades y evitar bloqueos entre hilos. Además, en ese mismo instante se registra la información del nodo que se conecta en la estructura de nodos conocidos. Se detalla mas en la siguente seccion.

7.5. Node Handshake y conexiones iniciales

Para establecer un clúster con topología de *full mesh* (es decir, una conexión completa entre todos los nodos), se sigue un proceso de conexión incremental basado en dos mecanismos principales: el **handshake inicial** y el **protocolo de Gossip**.

7.5.1. Fase de Inicio: Conexión al Nodo Semilla

Durante el arranque del cluster, es necesario inicializar primero un nodo semilla, usualmente un nodo `master`. Luego, el resto de los nodos se conectan progresivamente a este nodo: primero los otros `masters`, y luego las `réplicas`.

Cuando un nodo arranca, se conecta al nodo semilla usando una conexión TCP. Inmediatamente, envía un mensaje de tipo `MEET` utilizando el protocolo interno del clúster. Este mensaje contiene la información de identificación del nodo que desea ingresar al clúster.

El nodo semilla, por su parte, recibe la conexión mediante su módulo `HandleConnectionIncoming`, procesa el mensaje `MEET`, y registra la información del nodo en su estructura interna de nodos conocidos. Posteriormente, responde también con un mensaje de tipo `MEET`, cerrando así el proceso de handshake. En paralelo, ambos nodos registran sus respectivos streams de lectura y escritura TCP, los cuales son gestionados separadamente para evitar conflictos de concurrencia y mejorar la eficiencia.

7.5.2. Expansión de la Conectividad: Protocolo Gossip

Una vez realizadas las conexiones iniciales con el nodo semilla, el clúster se completa automáticamente mediante el protocolo de *Gossip*. Cada nodo envía periódicamente mensajes de tipo `PING` que contienen una sección de tipo `Gossip`, en la cual se describen otros nodos del clúster.

Cuando un nodo recibe un mensaje `PING`, inspecciona la sección `Gossip` para identificar nodos que aún no conoce ni con los cuales mantiene conexión. Si detecta nodos desconocidos, inicia una nueva conexión TCP hacia ellos, enviando nuevamente un mensaje de tipo `MEET`. El receptor repetirá el proceso simétricamente, generando así el intercambio mutuo de información.

Este mecanismo asegura que, partiendo de una conexión parcial hacia un nodo semilla, se genere de forma progresiva una red completamente conectada entre todos los nodos del clúster, sin necesidad de configurar explícitamente todas las conexiones punto a punto.

7.6. Detección de Fallos y Protocolo *Gossip*

La tolerancia a fallos del cluster se basa en un mecanismo de *heartbeat* implementado mediante los mensajes `PING`/`PONG` y un protocolo *gossip* que propaga el estado de cada nodo conocido por otro. Cada nodo ejecuta dos hilos dedicados (ver Sección 7.4.1):

Cronómetros internos Cada entrada de nodos conocidos mantiene:

- `ping_sent_time` – instante en que se envió el último `PING`.
- `pong_received_time` – instante del último `PONG` recibido.

Estos valores son actualizados por el **Módulo PING** (al enviar) y por el hilo **Procesador de Mensajes** (cuando recibe `PONG`), y son leídos por el **Módulo PONG** para determinar el estado del enlace.

- **Módulo PING**: cada 1 s selecciona un subconjunto aleatorio de nodos y les envía un mensaje `PING`. El paquete incluye una sección `Gossip` con la información de estado (*NodeId*, *flags* y *SocketAddr*) de $n/2$ nodos aleatorios que el emisor conoce. Adicionalmente se envian siempre `PING` a aquellos nodos que superan los `NODETIMEOUT / 2` sin recibir `PING`.
- **Módulo PONG**: despierta cada 100 ms y compara, para cada nodo vecino, el instante del *último PING enviado* con el del *último PONG recibido*, ademas se verifica la condicion de que `ping_sent_time` < `pong_received_time`, esta segunda condicion es para evitar falsos positivos. Si se cumplen estas dos condiciones el nodo se marca localmente con el estado `PFAIL`.

Estados y transiciones

1. **NORMAL → PFAIL**: cuando un nodo A detecta que un vecino B lleva más de `node_timeout` sin responder, lo marca como `PFAIL` y propaga dicha marca en la sección `Gossip` de los siguientes `PING`.
2. **PFAIL → FAIL**: si un nodo es reportado como `PFAIL` por al menos $\lfloor N/2 \rfloor + 1$ nodos maestros distintos, pasa a estado `FAIL`. Este cambio dispara, en caso de tratarse de un nodo `master`, el proceso de *replica promotion* (véase Sección 7.8).

Una vez que un nodo marca localmente a otro como `PFAIL`, este estado comienza a propagarse a través del clúster mediante los mensajes de tipo `PING`. En rondas posteriores de envío de `PING`, si el nodo marcado como `PFAIL` es incluido en la sección de *gossip*, los demás nodos que reciban este mensaje podrán detectar que un par ha sido considerado como potencialmente fallido.

Cuando un nodo receptor analiza la sección de *gossip* de un mensaje `PING`, compara el estado actual de sus nodos conocidos con la información recibida. Si identifica que otro nodo considera a un tercero como `PFAIL`, y si el nodo receptor también tiene conocimiento de ese tercero, entonces actualiza su propia visión del estado del nodo, marcándolo como `PFAIL` si aún no lo había hecho.

Este mecanismo descentralizado permite la diseminación eficiente del estado de falla potencial dentro del clúster. Cuando una mayoría de nodos independientes coinciden en que un nodo está en estado `PFAIL`, puede iniciarse el proceso de promoción de réplica (*replica promotion*) en caso de que el nodo en cuestión sea un `master`.

7.7. Replicación de datos y Modelo *Publish/Subscribe*

Dentro de cada nodo conviven dos flujos lógicos de mensajes:

1. **Mensajes de cliente (*self*)**: comandos originados por un cliente conectado localmente al nodo.
2. **Mensajes de clúster**: comandos o eventos recibidos desde otros nodos a través del *Cluster Bus*.

Para distinguir ambos flujos, el hilo **Procesador de Mensajes y Escritura** recibe un:

```
pub enum TipoMensajeNode {
    ClusterNode(ClusterMessage),
    InnerNode(InnerMensajeNode),
}
```

que encapsula el origen del mensaje y su tipo lógico. Es decir, mensajes de tipo *self* (inner Node) y de otros nodos (outer Node). El flujo general es el siguiente:

1. El **hilo Handler–Cliente** lee el comando RESP desde el socket del cliente y lo procesa localmente.

2. El resultado se reenvía al cliente. Simultáneamente, el comando se envía por un canal MPSC al `Procesador de Mensajes y Escritura` para su eventual propagación.
3. El `Procesador de Mensajes y Escritura` clasifica y actúa:
   - **Comandos *Publish/Subscribe*** (`PUBLISH`): siempre se etiquetan como `Self`. El hilo los reempaqueta en un mensaje interno y los difunde a *todos* los nodos del clúster. Cada nodo los inyecta en su propio módulo `PubSubBroker`, garantizando que los clientes suscritos, sin importar a qué nodo estén conectados, reciban el mensaje.
   - **Comandos de escritura persistente** (`SET`, `LPUSH`, `SADD`, ...): si el nodo emisor es un `master`, reenvía el comando únicamente a sus réplicas asignadas. Al recibirlo, cada réplica ejecuta la misma operación para mantener la consistencia de los datos.

Este diseño presenta dos ventajas clave:

- **Aislamiento de responsabilidades:** la generación de la respuesta al cliente permanece desacoplada de la difusión interna, evitando bloqueos en el *hot path*.
- **Propagación selectiva:**
  - Los eventos *Pub/Sub* se difunden a todo el clúster, garantizando entrega global.
  - Las operaciones de escritura se replican sólo a las réplicas del `master` correspondiente, minimizando ancho de banda y evitando trabajo redundante.

En consecuencia, el mecanismo asegura tanto la *consistencia de los datos* como la *entrega en tiempo real* de los mensajes publicados, sin comprometer el rendimiento del nodo.

7.8. Replica promotion

Cuando un *master* detecta a otro en estado `FAIL`, propaga esta información en el cluster (Figura 12). Al recibir las réplicas esta información, lanzan un hilo de *replica promotion*. Este proceso es asincrónico, por lo que cada réplica tiene un hilo propio y actúan de manera independiente, solo compartiendo información mediante las herramientas del protocolo interno detalladas a continuación.

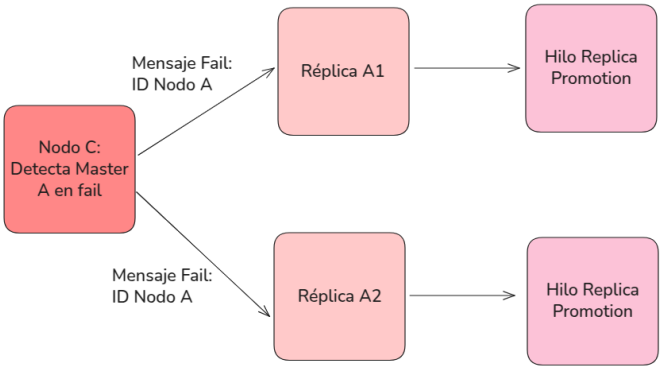


Figura 12: Propagación del mensaje FAIL

En primer lugar, aumentan su `currentEpoch` (donde se reflejan los grandes cambios en el cluster, para marcar el inicio del proceso) e intercambian mensajes de tipo `FailOverNegotiation`, donde determinan cuál tiene un mayor `replication offset`, un atributo del nodo que indica cuántas veces el mismo fue actualizado (Figura 13).

Mediante estos mensajes, se forma el *replica rank*, que asigna a cada réplica un número que indica su nivel de actualización frente a las otras (siendo el 0 la réplica con mayor `replication`

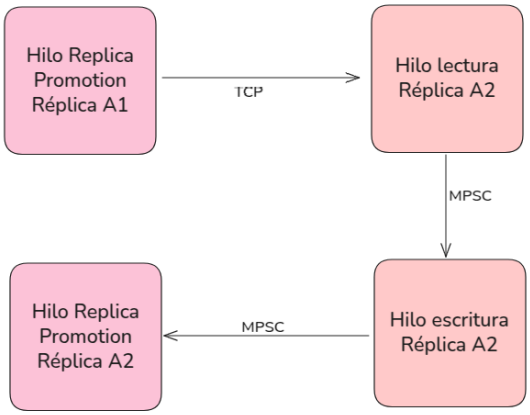


Figura 13: Failover negotiation (propagación replication offset)

`offset`). Mediante el *replica rank*, cada réplica calcula un *delay*, luego del cual comienza a pedir votos a los *masters*. Este *delay* corresponde a una cantidad fija de tiempo, un valor aleatorio dentro de un rango, y una componente proporcional al *replica rank*.

De esta manera, se prioriza que las réplicas más actualizadas pidan primero los votos (y probablemente los obtengan), pero no se impone de ninguna manera, como especifica la documentación de *Redis Cluster*.

Para enviar los pedidos de votos, se utilizan los mensajes `FailoverAuthRequest`. Al recibir los *masters* un mensaje de este tipo, evalúan si pueden efectuar el voto. Para esto, tienen en cuenta que haya pasado un tiempo fijo (una constante) desde su último voto hacia ese *master*, que el *master* que se busca reemplazar efectivamente se encuentre en estado `FAIL`, y que el `currentEpoch` del candidato sea mayor al propio y al del último voto por ese *master* (es decir, si votó a una réplica más reciente, no vota por una anterior).

En caso de decidirse que el pedido es inválido, simplemente se ignora. En otro caso, se actualiza su último voto y se envía un mensaje `FailoverAuthAck`, efectuando el voto (Figura 14). La réplica recibe este voto positivo, y lo suma a su contador de votos (en el hilo de *replica promotion*).

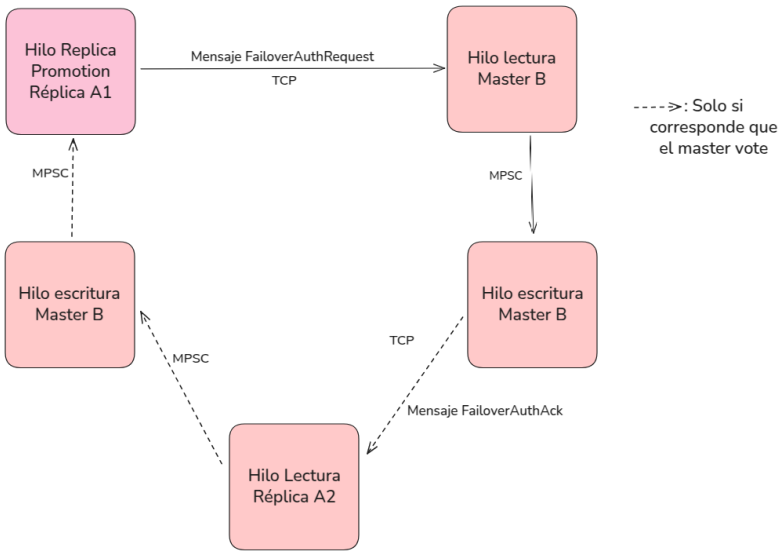


Figura 14: Propagación de un pedido de voto y respuesta

Luego del tiempo establecido para recibir votos, cada réplica revisa si ganó la votación, es decir,

fue elegida por la mayoría de los *masters*. En ese caso, se promueve a *master* y envía dos mensajes: `Update` y `MeetNewMaster`, cerrando el hilo de promoción.

El mensaje `Update` es enviado a todos los nodos, y los notifica de su nuevo rol, para que puedan actualizarlo en `NeighboringNodes`, y, por ejemplo, puedan efectuar redirecciones `MOVED` correctamente.

Por otro lado, `MeetNewMaster` es solo para las demás réplicas del proceso de promoción. Al recibirse este mensaje, actualizan su atributo `Master` y finalizan también el proceso de *replica promotion* (Figura 15).

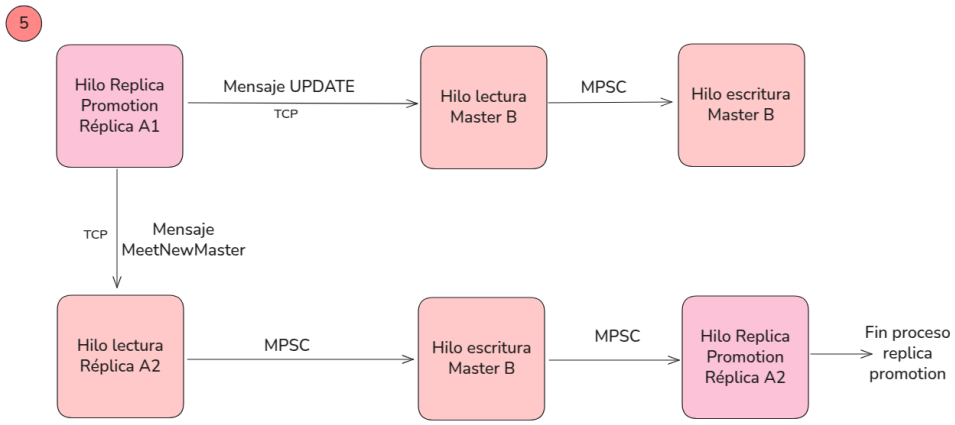


Figura 15: Mensajes Update y MeetNewMaster

De haber un empate al contar votos, simplemente se reinicia el proceso en el mismo hilo, aumentando el `currentEpoch` para reflejar un nuevo cambio en el clúster. En el caso particular de haber solo una réplica, la misma es promovida automáticamente; y, de no haber réplicas, el proceso de votación simplemente no se inicia (caso de `Cluster FAIL`).

7.9. Solo lectura, Errores de Enrutamiento y Estado del Clúster

Replica `READ`

Se implementa tambien la funcionalidad de solo lectura por parte de las replicas. Es decir, un cliente podria conectarse a una réplica y ejecutar comandos de solo lectura. De esta forma, un cliente que solo realiza lectura no necesita conectarser a un master, permitiendo una mayor distribucion de cargas entre nodos.

Comando `MOVED`

Cuando un cliente ejecuta un comando con una clave asociada a un *slot* que no está asignado al nodo actual, se devuelve un error `MOVED`. Este error indica al cliente que debe redirigir el comando al nodo responsable del *slot* correspondiente. La verificación se realiza en el módulo de almacenamiento (`storage`), que chequea si el slot de la clave pertenece al nodo actual. En caso negativo, se genera la respuesta de error con el formato definido por Redis.

Estado `CLUSTERDOWN`

El estado `CLUSTERDOWN` indica que el clúster no puede operar con normalidad, generalmente porque no hay nodos activos para cubrir todos los *slots* o por la pérdida del *quorum* de masters. Esto se verifica constantemente, se recorre la estructura de nodos conocidos o vecinos, validando si existen ***masters*** suficientes para la replica promotion (**quorum**) y si todos los **slots** están siendo cubiertos.

8. Redis Config

Para permitir una inicialización flexible y adaptable a distintos entornos, implementamos un sistema de configuración mediante archivo `.conf`. Este archivo define los parámetros clave con los que se levanta cada nodo del clúster y permite encapsular toda su configuración de forma clara y centralizada. Nuestro sistema de configuración permite ajustar tanto aspectos de red y particionamiento, como también opciones relacionadas a la persistencia, límites de clientes, autenticación, logging y más. Esto facilita no solo la puesta en marcha de un nodo individual, sino también la orquestación de múltiples nodos dentro del clúster de manera más predecible y reproducible.

8.1. Campos soportados

- `ip`: Dirección IP donde se levanta el nodo.
- `port`: Puerto en el que el nodo acepta conexiones externas (cliente).
- `slot_range_start` / `slot_range_end`: Definen el rango de slots que maneja el nodo. Es clave para determinar qué claves le corresponden a cada nodo según el hashing del clúster.
- `max_clients`: Límite máximo de clientes que pueden estar conectados simultáneamente a este nodo.
- `aof_file`: Ruta al archivo donde se guardan las operaciones mutables en modo AOF.
- `metadata_file`: Ruta donde se almacena la metadata persistente del nodo (por ejemplo, su `node_id`).
- `storage_file`: Ruta al archivo binario donde se guarda el estado completo del `Storage` en modo RBD.
- `log_file`: Ruta al archivo donde se escriben los logs del sistema.
- `users_file`: Ruta al archivo que define los usuarios habilitados para conectarse, junto con sus credenciales.
- `appendonly`: Flag (`yes`/`no`) que indica si se quiere habilitar persistencia AOF.
- `save`: Intervalo (en milisegundos) para guardar el estado binario del `Storage` (modo RBD).

Este archivo facilita el despliegue de nuevos nodos y la automatización del armado del clúster, ya que cada instancia puede levantar su configuración desde un archivo dedicado, minimizando errores y mejorando la reproducibilidad del sistema.

9. Logger

Para tener un mayor control sobre lo que ocurre dentro de nuestras aplicaciones, desarrollamos nuestra propia implementación de un logger.

Aunque registrar información en un archivo de texto puede parecer algo trivial, no puede realizarse de forma secuencial dentro del flujo principal de la aplicación. Hacerlo así podría generar cuellos de botella y afectar el rendimiento general.

Para resolver este problema, utilizamos channels. Esta solución nos permitió delegar el registro de logs a un hilo dedicado, que permanece en ejecución y se encarga exclusivamente de recibir mensajes de log y escribirlos en el archivo correspondiente. De esta manera, el flujo principal de la aplicación solo necesita enviar los mensajes a través del channel, mientras que el hilo dedicado se ocupa del resto.

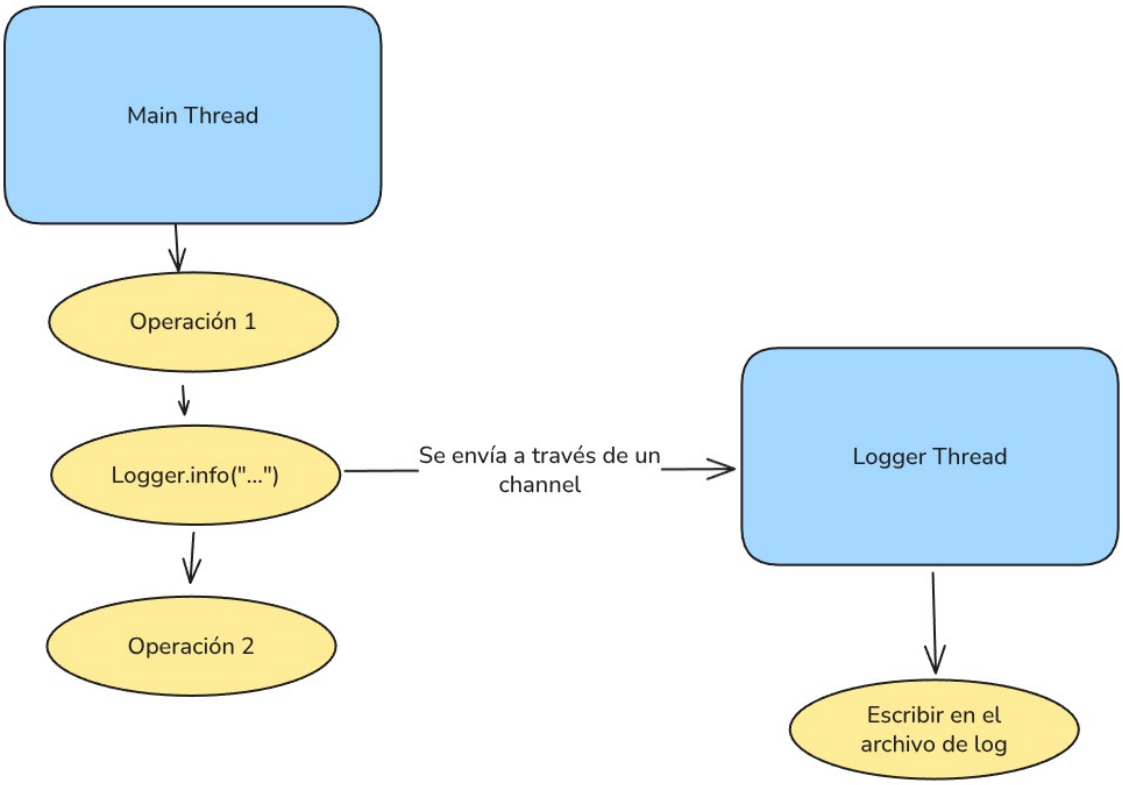


Figura 16: Arquitectura del Logger

10. Seguridad

Para reforzar la seguridad de nuestro clúster, implementamos tres mecanismos clave: autenticación, autorización y encriptación de datos en tránsito. La idea fue no solo proteger la comunicación entre cliente y nodo, sino también establecer controles claros sobre quién puede acceder y ejecutar comandos dentro del sistema.

10.1. Autenticación y autorización

Cada nodo valida a los clientes que intentan conectarse mediante un sistema de autenticación basado en usuarios. El proceso se realiza a través del siguiente handshake:

```
AUTH user password
```

Este mecanismo valida las credenciales enviadas por el cliente contra un archivo de usuarios permitidos. Dicho archivo debe estar referenciado en el archivo de configuración mediante el atributo `users_file`, y contiene una lista de pares usuario/contraseña autorizados para interactuar con el sistema.

Este enfoque nos permite un control explícito sobre qué usuarios están habilitados para operar, separando la lógica de permisos desde el arranque del nodo.

10.2. Encriptación en tránsito

Además del control de acceso, también implementamos encriptación de los datos en tránsito utilizando el algoritmo AES, con el objetivo de asegurar que los mensajes intercambiados entre cliente y nodo no puedan ser interceptados ni leídos por terceros.

El cifrado se aplica directamente a nivel de protocolo: antes de enviar un mensaje, el nodo lo serializa y luego lo encripta; del lado del cliente, el proceso se invierte: primero se desencripta el mensaje y luego se deserializa. Este flujo puede visualizarse en la Figura 17.

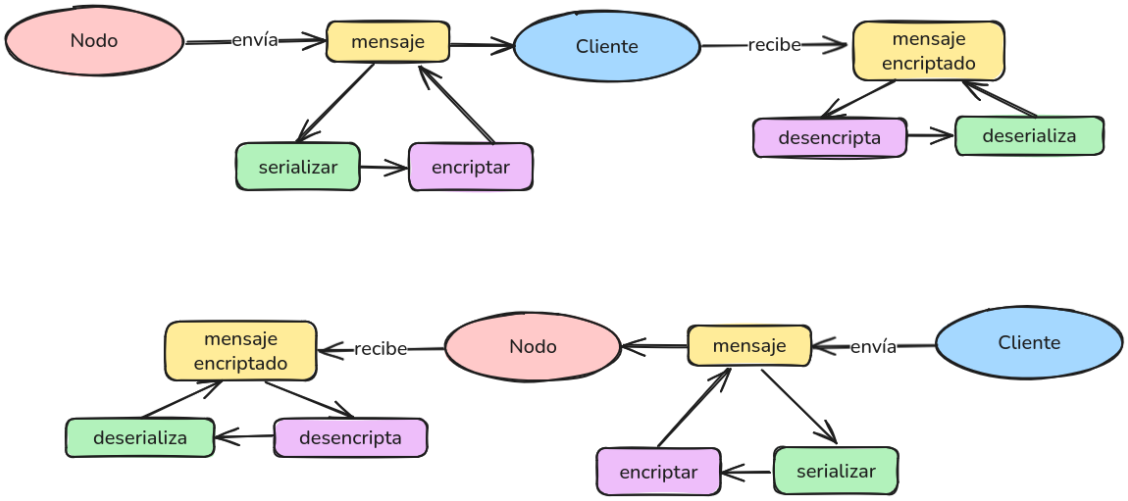


Figura 17: Esquema de encriptado y serialización de datos entre nodo y cliente.

Este enfoque mantiene la lógica de mensajes simple y nos permite abstraer la seguridad sin acoplarla directamente a las otras funcionalidades del cluster

11. Aplicacion cliente

La aplicación del cliente utiliza la librería egui, una librería open source que permite la creación de interfaces medianamente sencillas, con solo pocas dependencias, podemos levantar una interfaz funcional.

11.0.1. Flujos del usuario

Inicialmente, el usuario va a tener que hacer log in contra la aplicación para poder conectarse, esto para que pueda operar de manera segura.

Una vez dentro de la aplicación el usuario verá 4 opciones

- **Crear Documento** Permite al usuario crear un documento con un nombre especifico
- **Crear Hoja de Cálculo** Permite al usuario crear una hoja de cálculo con un nombre especifico
- **Abrir Documento** Le permitirá al usuario elegir un archivo y conectarse para la edición del mismo
- **Abrir Hoja de Cálculo** Le permitirá al usuario elegir una hoja de cálculo y conectarse para la edición del mismo

Vale aclarar que si bien, no permitimos modificar el tamaño de la grilla en su creacióon, esta del lado del backend puede tener valores arbitrarios, pero nos encontramos con limitaciones técnica de la librería al tener una grilla de más de 17 celdas.

En la hoja de cálculo se soportan las siguientes operaciones:

- **Operaciones con 2 argumentos** (`a` y `b`)

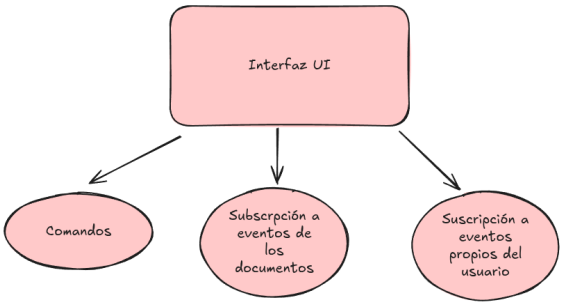


Figura 18: Estructura de la Interfaz UI

- `SUMA`: Suma $a + b$.
- `RESTA`: Resta $a - b$.
- `MUL`: Multiplicación $a \times b$.
- `DIV`: División entera a/b (error si $b = 0$).
- `MOD`: Módulo (resto) $a\,\%b$.
- `PER`: Porcentaje $(a \times 100)/b$.

- **Operaciones con rango (`Vec<i32>`)**
  - `SUMA`: Suma todos los elementos $\sum_i \text{nums}[i]$.
  - `RESTA`: Resta secuencial $\text{nums}[0] - \text{nums}[1] - \ldots$.
  - `MUL`: Producto $\prod_i \text{nums}[i]$.
  - `PROMEDIO`: Promedio entero $\left\lfloor \frac{\sum_i \text{nums}[i]}{\text{len}(\text{nums})} \right\rfloor$.
- **Manejo de casos especiales**
  - División por cero: Retorna `."Error: Division by zero"`.
  - Vectores vacíos:
    - En `SUMA`, `MUL` y `PROMEDIO`, retorna `"0"`.
    - En `RESTA`, retorna `"0"` (si el vector está vacío).
  - Comando desconocido: Retorna `Ünknown command: {cmd}"`.

11.1. Backend de la Interfaz

Internamente, la interfaz está conectada a nuestra base de datos de Redis y solamente se comunicará con el microservicio mediante el prótocolo PubSub.

Por lo que, tenemos una estructura en la que tendremos 3 procesos corriendo en segundo plano que realizan o se ocupan de distintas funcionalidades.

- **Proceso de Comandos** Un proceso en el cual solamente enviará comandos de Redis
- **Proceso de Suscripción a Documentos** Proceso en el cual se recibirán los eventos relacionados al documento que estoy actualmente editando
- **Proceso de Suscripción de Eventos propios** Este proceso se encargará de recibir todas los eventos que surgen en respuesta a peticiones al microservicio.

12. Microservicio

Uno de los criterios de aprobación consiste en desarrollar un microservicio responsable de la persistencia de los archivos, el control de su creación y edición, y la comunicación con los clientes mediante el protocolo PubSub. Es importante destacar que este microservicio será siempre la fuente de verdad respecto al contenido de los archivos. Por lo tanto, cada vez que un usuario se una a la edición de un documento, será el microservicio quien se encargue de enviar el contenido actual mediante eventos a través de PubSub.

Las tareas del microservicio comienzan cuando este recibe eventos a través de los canales a los que está suscripto. Sin embargo, antes de avanzar con esta lógica, fue necesario resolver una dificultad inicial: los documentos y hojas de cálculo se crean de forma dinámica, por lo que no es viable suscribirse manualmente a cada canal individual. Hacerlo implicaría lanzar un thread por cada canal, dado que cada subscripción representa un contexto completamente independiente para cada documento o hoja.

Para solucionar este inconveniente, se implementó un mecanismo de suscripción por patrones, lo que permitió reducir la cantidad de threads a uno solo. Este hilo es el encargado de recibir los eventos correspondientes a todos los archivos que se encuentren actualmente en edición, sin importar cuántos sean.

A continuación, se detallan los canales a los que el microservicio permanece suscripto, a la espera de eventos:

- **sheets:*** Suscripción por patrones que permite recibir cualquier evento relacionado con hojas de cálculo.
- **documents:*** Suscripción por patrones que cumple la misma función que la anterior, pero para documentos.
- **sheets:utils** Canal reservado para utilidades relacionadas con la creación y listado de hojas de cálculo.
- **documents:utils** Canal reservado para utilidades relacionadas con la creación y listado de documentos.

12.1. Eventos soportados

Creación de documento

```
- topic: documents:utils
- action: create
  document_name: string
  user_id: string
```

Listado de archivos

```
- topic: documents:utils
- action: list
  user_id: string
  file_type: string
```

Operaciones en documentos

```
- topic: documents:<document_id>
- action: edition
  user_id: string
```

```
  op: insert
  position: usize
  content: string

- action: edition
  user_id: string
  op: delete
  start_position: usize
  end_position: usize

- nota: múltiples operaciones se separan con "|"
```

Unión a un documento

```
- topic: documents:<document_id>
- action: join
  user_id: string
```

Desconexión de un documento

```
- topic: documents:<document_id>
- action: disconnect
  user_id: string
```

Sincronización de documento

```
- topic: documents:<document_id>
- action: sync
  content: string
  users: {user1, user2, ...}
```

Creación de hoja

```
- topic: sheets:utils
- action: create
  document_name: string
  user_id: string
  width: usize
  height: usize
```

Edición de hoja

```
- topic: sheets:<sheet_id:{}>
- action: edition
  user_id: string
  op: insert
  position: usize
  value: string
  column: usize
  row: usize

- action: edition
  user_id: string
  op: delete
```

```
  start: usize
  end: usize
  column: usize
  row: usize
```

Sincronización de hoja

```
- topic: sheets:<sheet_id:{}>
- action: sync
  content: string
  users: {user1, user2, ...}
  width: usize
  height: usize
```

Respuestas al cliente

```
- respuesta: creación de documento
  topic: users:<user_id>
  response: creation
  id: string

- respuesta: lista de archivos
  topic: users:<user_id>
  response: files
  files: file1, file2, ...

- respuesta: creación de hoja
  topic: users:<user_id>
  response: sheet_creation
  sheet_id: string
  name: string
  width: usize
  height: usize
```

12.2. Persistencia de archivos

Los archivos se almacenan de forma periódica en la base de datos. Al iniciarse el microservicio, se recuperan estos documentos y se cargan nuevamente en memoria. Esta persistencia se implementó mediante un hilo secundario que ejecuta la tarea en intervalos configurables de tiempo.

13. Conclusiones

El desarrollo de este proyecto permitió implementar una solución distribuida inspirada en Redis, utilizando Rust como lenguaje base. Se construyó un clúster funcional con persistencia, replicación, y comunicación entre nodos, junto con un microservicio que coordina sesiones colaborativas y guarda el estado de los documentos. Además, se integró una aplicación gráfica para edición en tiempo real, aprovechando el sistema Pub/Sub del clúster.

Durante el proceso, fortalecimos la dinámica de trabajo colaborativo, aplicando buenas prácticas en el uso de Git, como el trabajo con ramas aisladas, revisiones por pull requests y protección de la rama principal. Esto permitió mantener una organización clara y resolver los desafíos técnicos de forma eficiente y coordinada.

14. Extras agregados

Se implementaron los extras de Shard PubSub y encriptacion en reposo.

14.1. Encriptación en reposo (`.rdb`)

Para proteger los datos sensibles almacenados en el sistema, se implementó un mecanismo de **encriptación en reposo** basado en el algoritmo **AES-128 en modo CBC** (*Cipher Block Chaining*) con **relleno PKCS7**. Este esquema se aplicó tanto a la persistencia binaria del estado general del sistema como al registro secuencial de comandos mutables.

14.1.1. Persistencia binaria (`.rdb`)

El sistema mantiene una estructura de almacenamiento clave-valor distribuida por *slots*, cuyo estado completo puede persistirse periódicamente a disco en formato binario. En esta instancia de guardado, cada valor almacenado es primero convertido a una cadena RESP y luego encriptado con AES-128-CBC utilizando una clave simétrica fija (`AES_KEY`) y un vector de inicialización (IV) aleatorio de 128 bits generado en cada operación. El formato final del bloque cifrado incluye:

- 2 bytes que indican la longitud total del mensaje (en *big endian*),
- 16 bytes correspondientes al IV,
- el mensaje encriptado (ciphertext).

Este bloque cifrado se serializa junto con su clave asociada y se escribe al archivo de persistencia binaria. Al restaurar el estado desde disco, cada valor es desencriptado utilizando el IV embebido, reconstruyendo así la instancia de almacenamiento original.

14.1.2. Archivo de operaciones (`.aof`)

Complementariamente, el sistema lleva un registro secuencial de operaciones mutantes mediante un archivo AOF (*Append-Only File*). Cada línea del AOF representa un comando RESP ejecutado por el usuario, como por ejemplo `SET clave valor`. Para evitar el almacenamiento de valores sensibles en texto plano, se encriptan únicamente los argumentos mutables definidos por los metadatos de cada comando. Esta encriptación se realiza con el mismo esquema AES-128-CBC, pero se codifica posteriormente en *hexadecimal* para garantizar compatibilidad textual con el formato del archivo.

Durante la restauración del sistema, estas líneas se parsean, se desencriptan los campos hexadecimales y se reejecutan en orden, permitiendo recuperar el estado del sistema aun si falló la persistencia binaria.

14.1.3. Integración y tolerancia a fallos

Ambos mecanismos se encuentran integrados en la lógica de restauración del sistema. Se intenta primero restaurar desde el archivo binario, y luego aplicar el AOF sobre el resultado. Si la restauración binaria falla, se inicializa una estructura vacía y se aplica únicamente el AOF. Esta estrategia garantiza robustez frente a errores en disco o fallos parciales durante el guardado.

En conjunto, esta implementación de encriptación en reposo permite cumplir con principios de seguridad como confidencialidad y durabilidad, sin comprometer el rendimiento ni la estructura modular del sistema.

14.2. Shard Pub/Sub: Diseño e Implementación

14.2.1. ¿Qué es Shard Pub/Sub?

Redis Cluster permite dos mecanismos de publicación y suscripción: el clásico **Pub/Sub** y la variante **Shard Pub/Sub**. En el modelo clásico, todos los mensajes publicados se envían a los suscriptores conectados al mismo nodo donde se publica el mensaje y a todos los otros nodos del cluster, lo cual implica una mayor carga en el trafico de red del cluster.

En este modelo, la suscripción se realiza a un canal que se mapea mediante el algoritmo de **hash slot** a un nodo específico, y sólo dicho nodo manejará las publicaciones en ese canal. Esto reduce el tráfico de sincronización entre nodos y mejora la escalabilidad.

14.2.2. Implementación de Comandos Shard Pub/Sub

Para dar soporte completo al sistema, se implementaron todos los comandos del estándar **Shard Pub/Sub**:

- `SPUBLISH <canal><mensaje>`
- `SSUBSCRIBE <canal1><canal2>...`
- `SUNSUBSCRIBE [canal1 canal2 ...]`
- `PUBSUB SHARDCHANNELS [patrón]`
- `PUBSUB SHARDNUMSUB [canal1 canal2 ...]`

El procesamiento de estos comandos sigue una lógica similar a la del sistema *Pub/Sub* clásico. Sin embargo, la gestión de suscripciones y publicaciones se realiza mediante estructuras separadas para evitar conflictos entre ambos mecanismos.

14.2.3. Estructura de Datos Interna

Se utiliza una estructura dedicada al manejo de canales **shardeados**, similar a la del `PubSub Broker` clásico, pero separada:

```
type ShardChannels = HashMap<Canal, HashMap<String, Sender<String>>>;
```

Cada cliente mantiene también un registro de los canales shardeados a los cuales está suscripto:

```
struct Client {
    ...
    shard_channels: HashSet<Canal>,
    ...
}
```

Esto permite realizar una gestión eficiente de las desuscripciones masivas y desconexiones, asegurando la correcta limpieza de referencias.

14.2.4. Distribución de Canales mediante Hash Slot

Al momento de procesar un comando `SPUBLISH`, se calcula el **hash slot** del canal mediante la función estándar de Redis, y se determina si el nodo actual es el responsable de dicho slot. Sólo si el canal pertenece al nodo se procesa la publicación y se reenvía el mensaje a los suscriptores. Ademas se propaga el comandos a todos los demas nodos que si manejan el solot correspondiente.

Este mismo criterio se aplica para `SSUBSCRIBE`: el cliente sólo puede suscribirse en el nodo dueño del slot correspondiente. Si el cliente intenta suscribirse en un nodo incorrecto, se devuelve un error o redirección.

14.2.5. Ventajas del Modelo Shard Pub/Sub

- Escalabilidad: los canales se procesan sólo por el nodo dueño del slot.
- Menor tráfico entre nodos: no se requiere replicar mensajes a todos los nodos.
- Eficiencia en el uso de memoria y CPU al centralizar suscripciones.
- Balanceo de carga natural mediante distribución de slots.

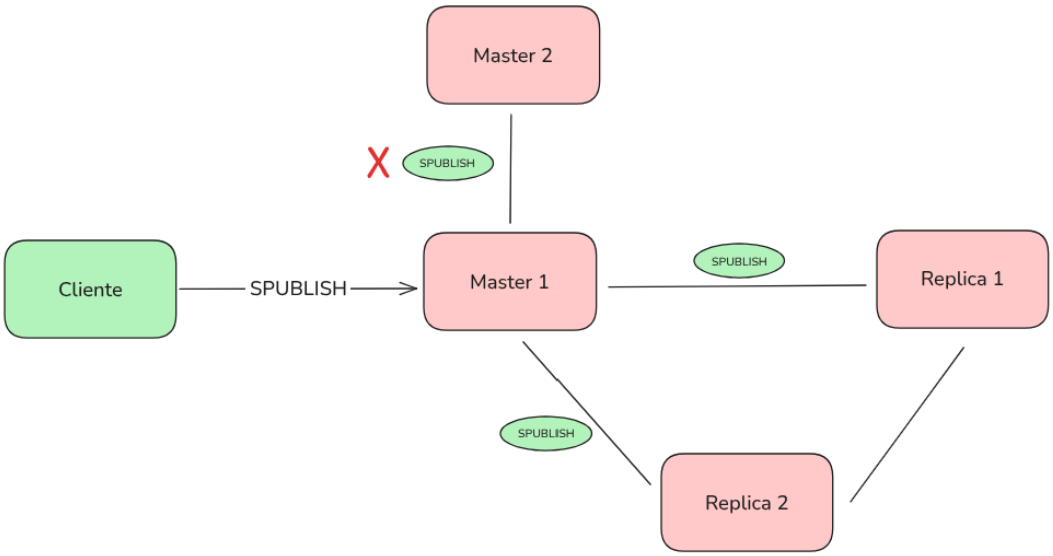


Figura 19: Ejemplo de flujo de `SPUBLISH`, se asume que `Master 1` maneja el valor del slot del canal a publicar

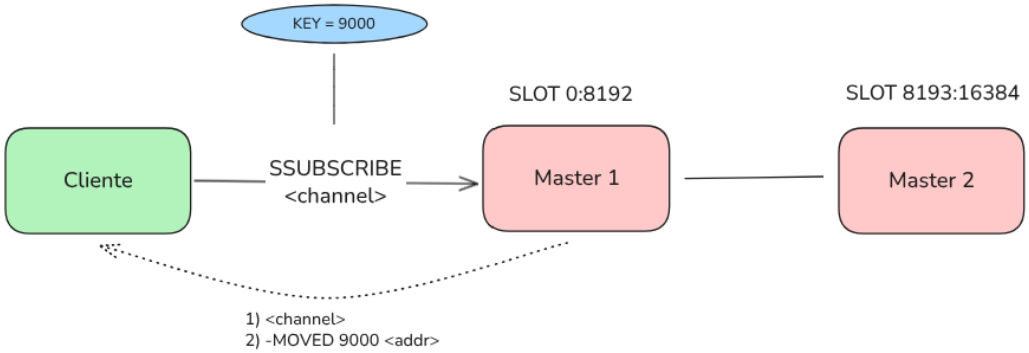


Figura 20: Ejemplo de flujo de `SSUBSCRIBE` donde el canal no corresponde con el `Master 1`

15. Docker y Docker-Compose

Docker es una herramienta de software que permite la contenerización de aplicaciones, lo que brinda beneficios como la portabilidad y la creación de entornos livianos que incluyen únicamente las dependencias necesarias para su ejecución. La portabilidad permite que la aplicación pueda

ejecutarse en distintos sistemas operativos, ya que el contenedor de Docker mantiene un entorno consistente en el que la aplicación seguirá funcionando correctamente.

Si bien esto resulta útil para una aplicación que no necesita comunicarse con otras réplicas del mismo software, en caso de que quisiéramos hacerlo, el proceso se vuelve un poco más complejo. Para resolver esta problemática, utilizaremos **Docker-Compose**.

Docker-Compose nos permite levantar múltiples contenedores de una manera más cómoda y realizar todas las configuraciones necesarias —que, de otro modo, tendríamos que especificar a Docker por línea de comandos— mediante un archivo en formato `yaml`. En nuestro caso, necesitaremos `docker-compose` para levantar varias instancias de nuestros nodos, cada una con una configuración distinta, así como también los microservicios.

15.1. Consideraciones

Para levantar nuestro clúster de Redis y los microservicios dentro de contenedores, debemos tener en cuenta lo siguiente:

- Los nodos deberán comunicarse mediante una red interna de Docker.
- Los nodos deberán poder ser accedidos desde fuera de Docker, para que nuestra aplicación gráfica pueda conectarse.
- Los microservicios —el de persistencia y el de IA— deberán poder conectarse a los nodos y reconectarse en caso de recibir un error del tipo `MOVED`.
- La red debe ser estable, ya que los nodos deben poder conectarse entre sí y detectar de forma eficiente si uno de ellos se cae o sufre microcortes.
- También se debe tener en cuenta que, si quisiéramos realizar una conexión entre distintas computadoras, los usuarios externos deberán poder conectarse a los nodos de Redis sin ningún tipo de problema.
- La creación del contenedor debe incluir únicamente las carpetas necesarias para lo requerido; es decir, no necesitamos que dentro del contenedor esté el *crate* de la UI, ya que eso agregaría un tiempo de compilación innecesario.

15.2. Creación del Dockerfile e Imagen de Docker

El primer paso consiste en la creación de la imagen de Docker. Para ello, contamos con el siguiente `Dockerfile`:

```
1 FROM rust:1.88-bookworm
2 WORKDIR /app
3 COPY . /app
4 RUN mv Cargo.docker.toml Cargo.toml
5 RUN apt-get update &&
6 apt-get install -y libssl-dev pkg-config &&
7 cargo build --release
```

Para evitar que el `Dockerfile` se vuelva innecesariamente extenso, implementamos un pequeño *hack*: agregamos un archivo `Cargo.docker.toml` que modifica el `workspace` de nuestra aplicación en Rust. Además, incluimos un archivo `.dockerignore` que excluye las carpetas relacionadas con `tests`, `ui` y `tui`.

15.2.1. Creación del Yaml de Docker-Compose: Nodos

El Dockerfile que agregamos, es la base de todo. Lo siguiente que hicimos fue crear nuestro archivo **docker-compose.redis.yaml** que define un yaml para que podamos levantar los nodos.

```
1 services:
2  redis_node:
3    image: rusty_docs:latest
4    container_name: redis_node_0${NODE_ID:-1}
5    environment:
6      - NODE_ID=${NODE_ID:-1}
7      - API_KEY=${API_KEY:-default_api_key}
8    volumes:
9      - ./distributed_configs/redis_0${NODE_ID:-1}.conf:/config/node.conf
10      - ./distributed_configs/users.txt:/config/users.txt
11      - ./distributed_configs/data:/config/data
12    ports:
13      - "${NODE_PORT:-8088}:${NODE_PORT:-8088}"
14      - "${CLUSTER_PORT:-18088}:${CLUSTER_PORT:-18088}"
15    command: ["./target/release/redis_node", "/config/node.conf"]
16    extra_hosts:
17      - "host.docker.internal:host-gateway"
18    networks:
19      - redis_network
20
21 networks:
22  redis_network:
```

Dependiendo de que nodo estamos levantando, vamos a utilizar cierta configuración.

También creamos unos scripts que nos permite levantar multiples nodos de manera dinámica.

`cluster.sh`

```
1 #!/bin/bash
2 ports=($(seq 8088 8096))
3 for i in {0..8}; do
4    NODE_ID=$((i + 1))
5    NODE_PORT=${ports[$i]}
6    echo $NODE_ID
7    echo $NODE_PORT
8    CLUSTER_PORT=$((10000 + NODE_PORT))
9   ./node.sh $NODE_ID $NODE_PORT $CLUSTER_PORT
10 done
```

`node.sh`

```
#!/bin/bash
1 NODE_ID=$1
2 if [ -z "$NODE_ID" ]; then
3    echo "Error: No se proporcionó el ID del nodo."
4    exit 1
5 fi
6 NODE_PORT=$2
7 if [ -z "$NODE_PORT" ]; then
8    echo "Error: No se proporcionó el puerto externo."
9    exit 1
```

```
10 fi
11 CLUSTER_PORT=$3
12 if [ -z "$CLUSTER_PORT" ]; then
13    echo "Error: No se pudo calcular el puerto de métricas."
14    exit 1
15 fi
17 gnome-terminal -- bash -c "
18    echo 'Levantando redis_node_$NODE_ID en puerto $NODE_PORT...';
19    NODE_ID=$NODE_ID \
20    NODE_PORT=$NODE_PORT \
21    CLUSTER_PORT=$CLUSTER_PORT \
22    docker compose -f docker-compose.redis.yaml --project-name redis_node_0$NODE_ID up --build |
23      read -p "Presiona Enter para cerrar la terminal..."
24    "
```

Con eso ya definido, podemos levantar nuestro Cluster de Redis, dentro de contenedores de Docker.

15.2.2. Creación del Yaml de Docker-Compose: Microservicios

Por otro lado, los microservicios son algo más simples de levantar, a continuación dejamos el yaml creado para ellos

`docker-compose.microservices.yaml`

```
1 version: "3.9"
2 services:
3  documents_handler:
4    image: rusty_docs:latest
5    command: ["./target/release/documents-handler-service",
6 "redis_address=host.docker.internal:8088", "save_timer=30000"]
7    extra_hosts:
8      - "host.docker.internal:host-gateway"
9    networks:
10      - redis_network
11  llm_service:
12    image: rusty_docs:latest
13    environment:
14      - API_KEY=${API_KEY}
15    command: ["./target/release/microservice_llm", "host=host.docker.internal", "port=8088",
16 "api_key=${API_KEY}"]
17    extra_hosts:
18      - "host.docker.internal:host-gateway"
19    networks:
20      - redis_network
21 networks:
22  redis_network:
```

15.3.

Comandos para levantar el ambiente

15.3.1. Crear la imagen de Docker

Para compilar todos los componentes y generar la imagen `rusty_docs:latest`, ejecutar:

```
make build_docker_image
```

15.3.2. Levantar el Cluster de Redis

Una vez creada la imagen, se puede levantar el cluster completo con:

```
make run_docker_cluster
```

Levantar nodos individuales

En caso de querer levantar un nodo de forma individual:

```
NODE_ID=$NODE_ID NODE_PORT=$NODE_PORT CLUSTER_PORT=$CLUSTER_PORT make run_docker_single_node
```

- `NODE_ID`: número de ID según la configuración en `distributed_configs/`
- `NODE_PORT`: puerto donde los clientes se conectarán al nodo
- `CLUSTER_PORT`: puerto utilizado para comunicación entre nodos

15.3.3. Levantar los microservicios

Los microservicios de AI y Persistencia son esenciales para el funcionamiento de la aplicación, ya que brindan funcionalidades clave a la interfaz gráfica.

Para levantarlos, ejecutar:

```
make run_docker_microservices
```

15.3.4. Detener los contenedores

En caso de que queramos detener los contenedores de los nodos, podemos utilizar el comando

```
make kill_docker_cluster
```

16. Nueva interfaz grafica

Para esta entrega se incorporaron nuevas funcionalidades en la aplicación cliente, con el objetivo de mejorar la experiencia de edición colaborativa, enriquecer las capacidades de los documentos soportados y ampliar la interacción con servicios externos.

16.1. Integración con Inteligencia Artificial

Se implementó la integración con un microservicio de lenguaje natural (LLM) provisto por el backend, que permite modificar documentos de texto a través de comandos enviados por los usuarios.

- **Edición localizada:** el usuario puede seleccionar una posición específica del documento e ingresar un *prompt* que solicite una modificación puntual (por ejemplo, reescribir una frase, traducir un párrafo, resumir una oración, etc.). El resultado se inserta directamente en el documento, en la posición indicada. Esta acción se realiza mediante clic derecho sobre el punto deseado, lo cual despliega un *tooltip* con la opción `Preguntar a la IA`. El resultado es insertado directamente en la posición seleccionada.

- **Edición global del documento:** se añadió además la posibilidad de enviar un *prompt* que actúe sobre el contenido completo del documento, como reescribirlo en un estilo determinado, traducirlo a otro idioma o realizar una mejora general. En este caso, se envía al microservicio únicamente el identificador del documento y el prompt, y es el backend quien se encarga de recuperar el contenido y devolver el texto modificado. Esta funcionalidad puede activarse haciendo clic en el botón del header denominado `Preguntar a la IA sobre el documento`.

16.2. Soporte para Markdown

La vista de edición de documentos (`RustyDocsUI`) fue adaptada para soportar sintaxis de Markdown. Esto permite a los usuarios aplicar formato enriquecido (títulos, listas, énfasis, enlaces, etc.) directamente desde la interfaz, manteniendo la estructura subyacente como texto plano y sin afectar la sincronización colaborativa.

16.3. Exportación de documentos

Se incorporó un botón para exportar el contenido del documento en formato `.md`, que permite al usuario guardar localmente una copia del documento con su contenido actual. Esta funcionalidad permite respaldar versiones del documento o integrarlo fácilmente en otras plataformas que utilicen Markdown.

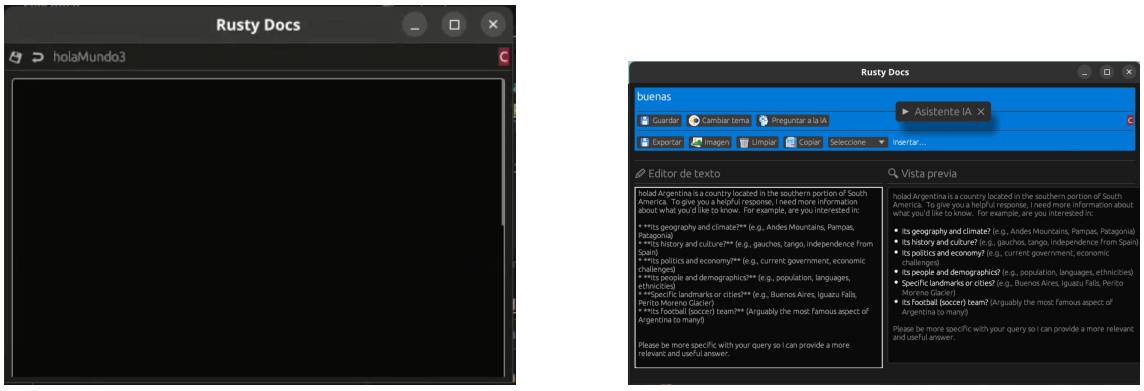
16.4. Coloreado local de celdas en hojas de cálculo

En la interfaz de edición de hojas de cálculo (`RustyExcelUI`) se incorporó la capacidad de modificar el color de fondo de las celdas. Esta modificación es local, es decir, no se propaga a otros usuarios ni se almacena en Redis. Su objetivo es brindar herramientas visuales para organización, sin alterar el contenido persistente del documento.

16.5. Modo claro/oscuro

Se implementó un selector de tema para cambiar entre modo claro y modo oscuro, disponible en ambas interfaces (`RustyDocsUI` y `RustyExcelUI`). Esta funcionalidad mejora la accesibilidad visual y la experiencia de usuario en distintos entornos de iluminación.

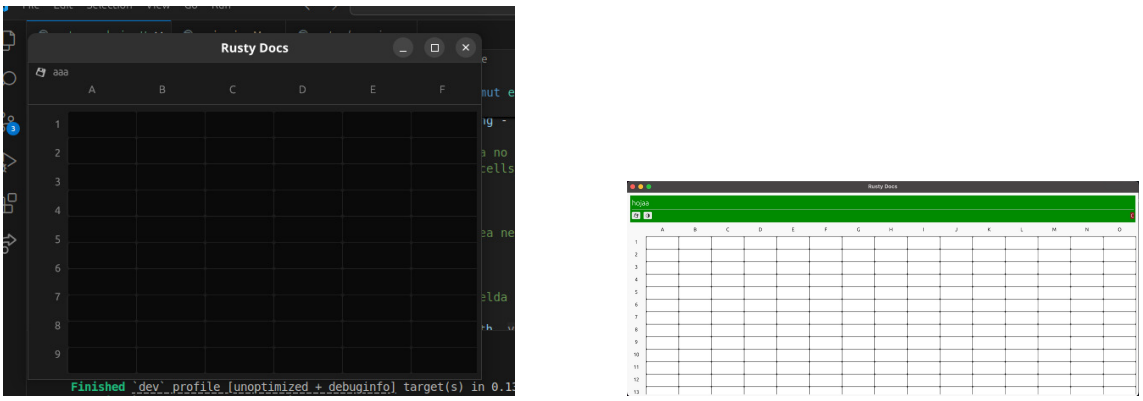
16.6. Antes y después de la interfaz del Docs



(a) Estado anterior (b) Estado posterior

Figura 21: Comparación entre el estado original y el modificado de la interfaz de Docs.

16.7. Antes y después de la interfaz del Excel



(a) Estado anterior (b) Estado posterior

Figura 22: Comparación entre el estado original y el modificado de la interfaz.

17. Microservicio LLM

Este apartado describe el diseño e implementación del microservicio LLM, encargado de procesar solicitudes de generación o modificación de texto utilizando un proveedor externo de inteligencia artificial (IA). El microservicio se comunica exclusivamente mediante canales `Redis PubSub`, asegurando un bajo acoplamiento entre clientes y otros microservicios. Se implementó un modelo de procesamiento multithread para garantizar la concurrencia de múltiples peticiones en paralelo.

17.1. Estructura de Comunicación

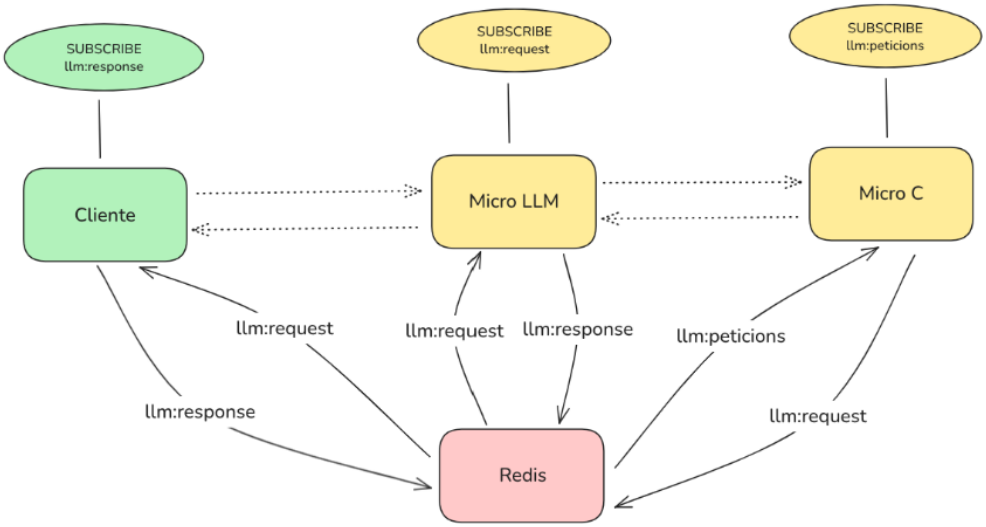


Figura 23: Comunicación entre entidades mediante canales Redis

El sistema se compone de tres actores principales:

- **Cliente**: publica solicitudes de procesamiento en el canal `llm:request`.

- **Microservicio LLM**: procesa la solicitud, interactúa con la IA y publica el resultado en el canal de respuesta correspondiente.
- **Microservicio de control y persistencia**: gestiona y provee el contenido de los documentos para solicitudes globales.

Las solicitudes contienen toda la metadata necesaria para procesar la petición y se codifican como JSON. El canal de respuesta se especifica dinámicamente en el mismo mensaje.

17.2. Formato de Mensajes

Ejemplo de solicitud de cliente:

```
{
  "requestId": "doc789_user123",
  "docId": "doc789",
  "type_request": "local/global",
  "prompt": "Resum  el contenido del documento completo.",
  "response_channel": "res:doc789"
}
```

Respuesta generada por el proveedor IA:

```
{
  "candidates": [
    {
      "content": {
        "parts": [{ "text": "Texto ejemplo devuelto por la IA." }],
        "role": "model"
      },
      "finishReason": "STOP"
    }
  ],
  "modelVersion": "gemini-1.5-flash",
  "responseId": "resp_global_summary_001"
}
```

Respuesta final enviada al cliente:

```
{
  "status": "ok/err",
  "text": "Texto ejemplo devuelto por la IA."
}
```

Petición del LLM al microservicio de control:

```
{
  "requestId": "doc789_user123",
  "docId": "doc789",
  "response_channel": "llm:request",
  "request_type": "get"
}
```

Respuesta del microservicio de control:

```
{
  "requestId": "doc789_user123",
  "docId": "doc789",
  "type_request": "global",
  "text": "Contenido completo del documento..."
}
```

17.3. Diseño Interno del Microservicio LLM

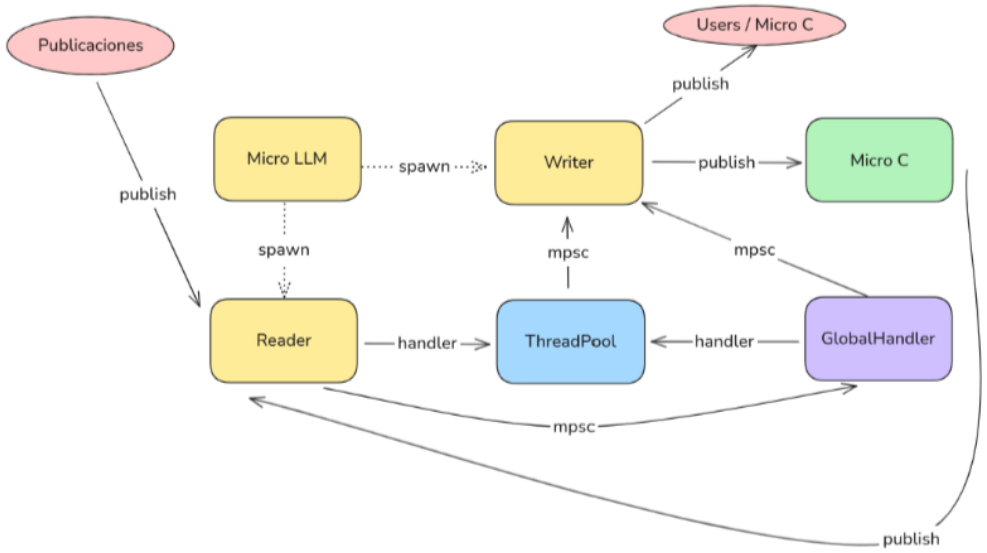


Figura 24: Componentes internos del microservicio LLM

El microservicio LLM se compone de los siguientes componentes concurrentes:

- **Reader Thread**: Se conecta al canal `llm:request`, escucha las solicitudes entrantes y las encola para su procesamiento.
- **Writer Thread**: Publica las respuestas generadas por la IA en el canal de respuesta indicado.
- **Handler Global**: Para solicitudes globales, solicita el texto completo al microservicio de control y encola el trabajo una vez recibido.
- **Thread Pool**: Gestiona la ejecución concurrente de múltiples tareas de generación utilizando hilos reusables.

Flujo:

1. El hilo `reader` recibe solicitudes publicadas en `llm:request`.
2. Dependiendo del tipo:
   - `local`: se ejecuta directamente en el thread pool.
   - `global`: se solicita primero el texto al microservicio de control.
   - `doc_text`: la respuesta del texto llega y se envía al pool.
3. El prompt es preparado y enviado al proveedor IA.
4. La respuesta es publicada por el `writer` en el canal correspondiente.

17.4. Modelo de Thread Pool

Para poder manejar múltiples solicitudes en paralelo, se implementó un modelo de ejecución basado en un `ThreadPool`, inspirado en el diseño propuesto por el libro oficial de Rust. Un `ThreadPool` es una abstracción que permite reutilizar un conjunto fijo de hilos de ejecución (`workers`) para procesar tareas concurrentes, evitando la sobrecarga de crear y destruir hilos constantemente.

Cada solicitud entrante se empaqueta en una tarea (`Job`) que se envía a través de un canal hacia los workers del pool. Estos workers se encuentran en espera (`blocking wait por Mutex del sender`) de nuevas tareas para ejecutar. El sistema garantiza sincronización segura mediante primitivas como `Mutex` y `Arc`, y además proporciona tolerancia a fallos a través de un mecanismo de recuperación automática en caso de `panic!`.

Definición del pool y mecanismos de recuperación:

```
type Job = Box<dyn FnOnce() + Send + 'static>;

struct Shared {
    receiver: Mutex<Receiver<Message>>,
    live: AtomicUsize,
    max: usize,
}

pub struct ThreadPool {
    handles: Vec<Option<JoinHandle<()>>>,
    sender: Sender<Message>,
    shared: Arc<Shared>,
}

struct Sentinel {
    shared: Arc<Shared>,
}

impl Drop for Sentinel {
    fn drop(&mut self) {
        self.shared.live.fetch_sub(1, Ordering::SeqCst);
        if std::thread::panicking() {
            self.shared.spawn_worker(); // Reemplazo del worker ca do
        }
    }
}
```

El struct `Sentinel` cumple el rol de "vigilante"de cada worker. Si un hilo entra en `panic!`, el `Drop` del `Sentinel` detecta esta condición y automáticamente lanza un nuevo worker para reemplazar al caído, manteniendo constante la cantidad de hilos activos definida inicialmente en `max`. La contabilidad de hilos activos se maneja con `AtomicUsize` para evitar condiciones de carrera.

Flujo general de ejecución

1. Se crea un `ThreadPool` con una cantidad fija de hilos (`max`).
2. Cada hilo se lanza ejecutando un bucle que intenta recibir y ejecutar tareas del canal compartido (`receiver`).
3. Las tareas se envían al pool mediante el canal `sender`, encapsuladas como funciones `FnOnce()`.
4. Si un hilo falla (entra en `panic!`), el `Sentinel` asociado detecta la condición y lanza un nuevo hilo para reemplazarlo.
5. El sistema continúa funcionando con la misma cantidad de hilos, sin necesidad de intervención externa.

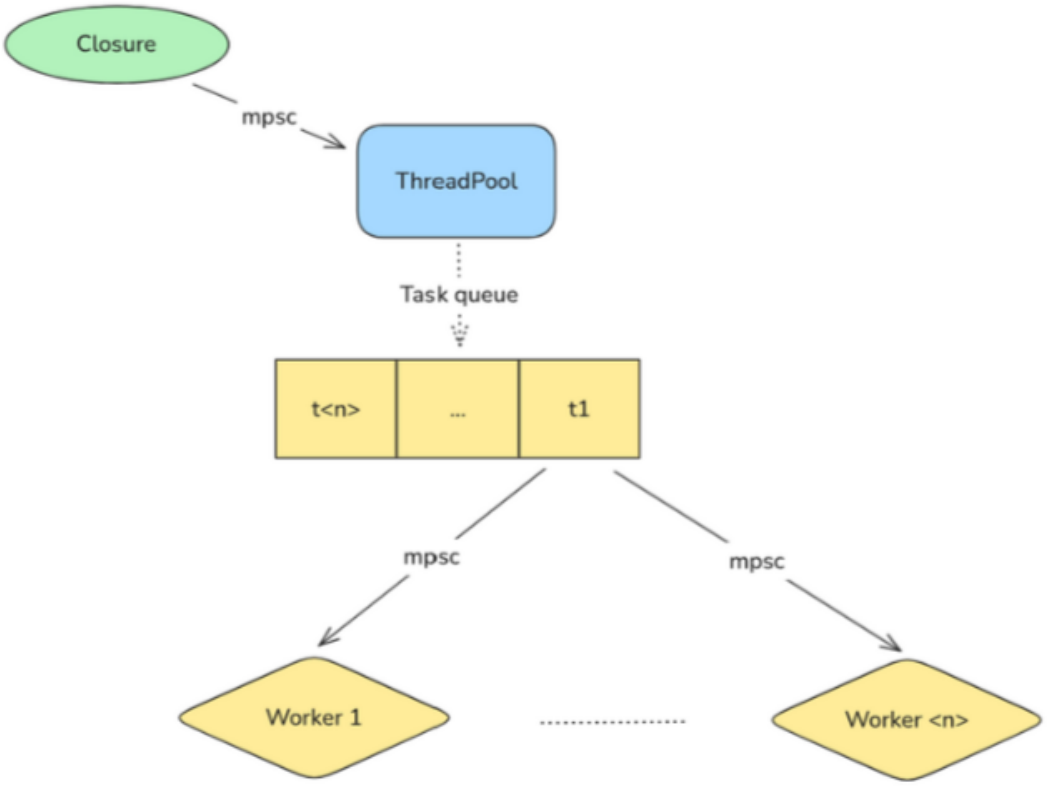


Figura 25: Flujo de procesamiento multithread con manejo de errores

Este enfoque proporciona un sistema robusto y eficiente, capaz de manejar múltiples solicitudes concurrentes de forma segura y tolerante a fallos. En el contexto del microservicio de control, este modelo permite procesar en paralelo comandos entrantes desde Redis, garantizando una rápida respuesta incluso bajo alta demanda. Al evitar la sobrecarga asociada a la creación frecuente de hilos y asegurar la recuperación automática ante fallos, el diseño favorece la estabilidad continua del sistema. Esto es particularmente relevante al coordinar la interacción con el LLM, donde las latencias pueden ser variables y la disponibilidad del servicio resulta crítica para sostener flujos de trabajo asincrónicos y distribuidos.

18. Costos de uso

Para desarrollar la aplicación, se utilizó **Gemini** como proveedor de inteligencia artificial, en particular la versión **Gemini 1.5 Flash**. Esto permitió desarrollar y probar el trabajo bajo un *free trial*, pero permitiendo una sencilla evolución a un plan pago si así se deseara. En particular, se decidió trabajar con este proveedor y esta versión por su alta velocidad y baja latencia, manteniendo costos bajos. De esta manera, se pudo analizar la rapidez, eficacia y el correcto funcionamiento de los algoritmos y el microservicio LLM implementado.

Se calcula ahora el costo estimado de mantenimiento de la solución presentada, medido en tokens (de entrada y salida de la IA) por mes.

En primer lugar, se plantea el análisis para la carga correspondiente al uso de la aplicación dentro de FIUBA, para calcular el costo aproximado por facultad. La misma cuenta con alrededor de 10.000 estudiantes. Suponiendo 1 documento mensual por alumno, se tiene un estimado de 10.000 documentos.

Se calculan documentos de aproximadamente 2.000 palabras (aproximadamente 3.000 tokens). Con 3 consultas a la IA (dos particulares y una sobre todo el documento), se rondan los 3.150

tokens de entrada por documento. Si se consideran 3 tokens de salida por cada uno del prompt, en total serán:

$$3{,}150 \times 3 = 9{,}450 \text{ tokens de salida por consulta.}$$

Luego, en un mes se estiman (considerando que los *tokens* se calculan como: **usuarios** $\times$ **documentos** $\times$ **tokens por documento**):

$$\text{Tokens de entrada} = 10{,}000 \times 1 \times 3{,}150 = 31{,}500{,}000$$
$$\text{Tokens de salida} = 10{,}000 \times 1 \times 9{,}450 = 94{,}500{,}000$$

Considerando los costos del modelo Gemini 1.5 Flash:

- US$ 0,075 por millón de tokens de entrada,
- US$ 0,30 por millón de tokens de salida,

El costo total mensual aproximado es:

$$\frac{0{,}075 \times 31{,}500{,}000}{1{,}000{,}000} + \frac{0{,}30 \times 94{,}500{,}000}{1{,}000{,}000} = \textbf{US\$ 30,71}.$$

Si deseáramos escalar nuestra propuesta a distintas universidades del mundo, podemos estimar el costo amplificando este resultado. Por ejemplo:

- Para 5 universidades: US$ 30,71 $\times$ 5 = **US$ 153,55**.
- Para 15 universidades: US$ 30,71 $\times$ 15 = **US$ 460,65**.

19. Conclusiones Finales

El proyecto desarrollado a lo largo del cuatrimestre permitió diseñar e implementar un sistema distribuido inspirado en Redis Cluster y todos los otros sistemas agregados que permitieron el desarrollo del el proyecto **Rusti Docs**.

El trabajo en equipo permitió consolidar tanto habilidades técnicas como colaborativas, destacándose el uso eficiente de Git y GitHub, junto con flujos de trabajo estructurados basados en ramas, revisiones y control de versiones. En síntesis, el proyecto cumplió plenamente con sus objetivos, logrando desarrollar un sistema distribuido robusto e integrando, además, un microservicio de inteligencia artificial. El resultado final fue una solución sólida, bien integrada y coherente tanto en su funcionamiento como en su organización interna.