.UBAfiuba
FACULTAD DE INGENIERÍA

Taller de Programación I

RustiDocs - 1C2025

Integrantes:

Nombre	Padrón
Camila General	105552
Ramiro Mantero	107162
Valentina Moreno	107948
Franco Secchi	106803

Introducción

En este trabajo práctico se desarrolló un sistema de trabajo colaborativo en tiempo real que permite confeccionar documentos de texto y planillas de cálculo mediante la implementación de un Redis Cluster distribuido, utilizado tanto para el almacenamiento de información como para el intercambio de mensajes.

El sistema está compuesto por un conjunto de nodos que replican y distribuyen datos, permitiendo múltiples clientes concurrentes y tolerancia a fallos. Además, se desarrolló un microservicio de control y persistencia que actúa como cliente especial: escucha las publicaciones realizadas por los usuarios, responde con información de sesión y estado del documento, y guarda periódicamente el contenido en el clúster.

La arquitectura aborda conceptos clave de sistemas distribuidos como replicación, detección de fallos, comunicación entre nodos y persistencia, todo mediante un protocolo de publicación y suscripción compatible con Redis.

Investigación Inicial

La investigación comenzó con el estudio de los fundamentos de Redis Cluster y cómo adaptarlo a una implementación propia en Rust. Nos enfocamos en entender cómo Redis distribuye, replica y gestiona la consistencia de los datos entre múltiples nodos. Entre los conceptos clave que aprendimos y aplicamos se encuentran:

- Hashing y particionamiento: Analizamos cómo Redis utiliza hash slots para distribuir claves entre los nodos del clúster, garantizando un reparto uniforme y determinístico.

- Replicación maestro-réplica: Estudiamos el modelo de Redis donde cada nodo maestro puede tener uno o más réplicas, e implementamos este esquema con detección de fallos, monitoreo de actividad, y un mecanismo para la promoción automática de réplicas en caso de caída del nodo maestro.

- Detección de fallos y consenso parcial: Incorporamos ideas similares al gossip protocol para que los nodos compartan sus conocimientos del estado del clúster. Además, diseñamos una lógica de consulta entre réplicas para confirmar la inactividad de un maestro antes de iniciar una promoción.

- Persistencia y durabilidad: Investigamos mecanismos de persistencia utilizados por Redis, como AOF (Append-Only File) y RDB (Redis Database Backup). En nuestro sistema, optamos por un enfoque mixto, implementando una persistencia en archivos divididos por rango de slots, garantizando durabilidad sin sobrecargar el rendimiento.

- Pub/Sub distribuido: Analizamos cómo Redis maneja canales de publicación/suscripción, y replicamos este comportamiento para permitir que múltiples clientes colaboren en la edición de documentos. Cada documento tiene asociado un canal, y los nodos publican mensajes a suscriptores en tiempo real.

- Encriptado en tránsito: Implementamos un algoritmo XOR modificado para proteger los datos intercambiados entre nodos.

- Protocolo RESP (REdis Serialization Protocol): Para garantizar compatibilidad con herramientas cliente y mantener una estructura clara de comunicación, utilizamos el protocolo RESP, el mismo que emplea Redis oficialmente. RESP permite representar comandos y respuestas en un formato textual simple pero estructurado, que es fácilmente interpretable tanto por humanos como por clientes automatizados.

Arquitectura General del Sistema

El sistema desarrollado implementa un clúster distribuido inspirado en Redis Cluster, utilizando el lenguaje Rust. La arquitectura se organiza en tres componentes principales: nodos Redis, clientes y un microservicio de control. La comunicación entre ellos se realiza a través de sockets TCP usando el protocolo RESP, y se brinda soporte para comandos clave-valor, pub/sub, redirección de claves y persistencia distribuida.

Cada nodo puede actuar como maestro o réplica, y se le asigna un rango de *hash slots* que determina qué claves gestiona. Los nodos intercambian información de estado, replican datos y ejecutan un protocolo de promoción en caso de fallo del maestro.

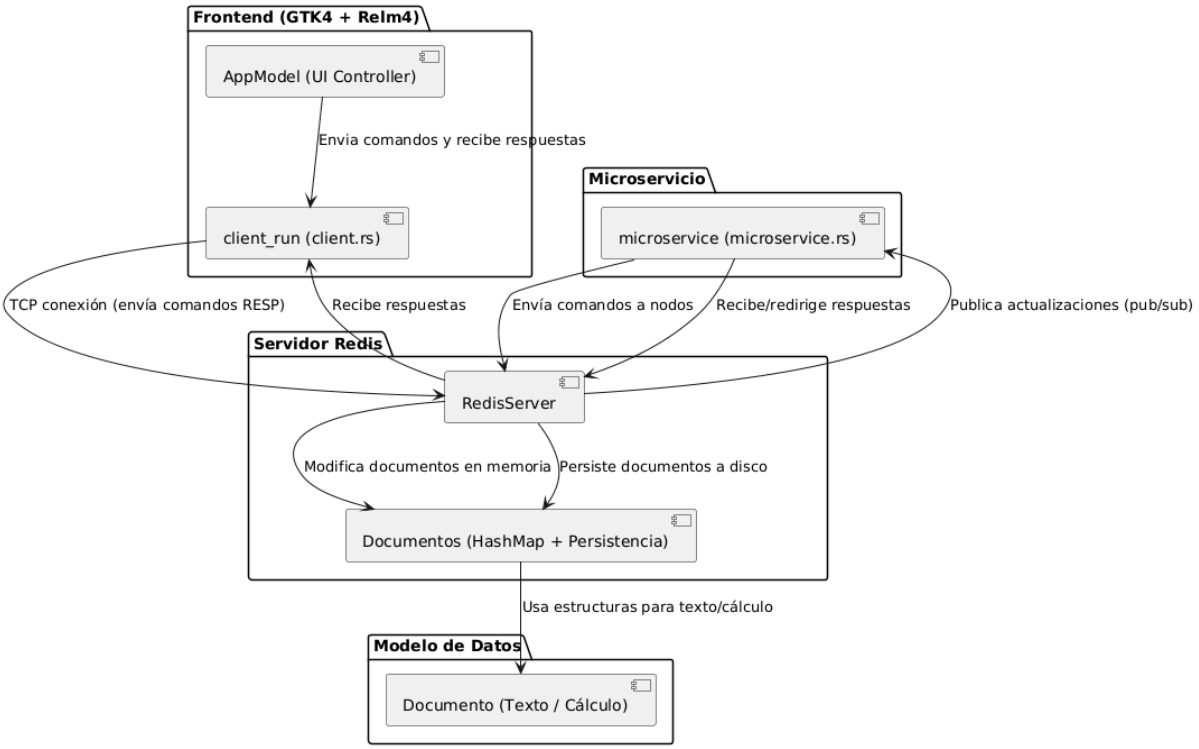
El sistema implementa una función de hash basada en CRC16 que determina, a partir del nombre de la clave, en qué nodo debe ejecutarse una operación. Cuando un cliente se conecta a un nodo y envía un comando sobre una clave que ese nodo no maneja, este responde con un mensaje de redirección (tipo `ASK`) para que el cliente reenvíe el comando al nodo correcto.

Además, se incorpora un microservicio de colaboración y persistencia que escucha los canales pub/sub asociados a cada documento. Cuando un cliente publica un mensaje (por ejemplo, una edición en un documento), el microservicio lo recibe, actualiza su propia copia local del archivo, y lo mismo hacen todos los demás clientes suscriptos al canal correspondiente. Es decir, la propagación del cambio ocurre de forma distribuida y en tiempo real.

Cada tres segundos, el microservicio toma su versión actualizada del documento y la persiste en Redis mediante un comando `SET`, asegurando que los datos se mantengan consistentes en el clúster. Esta estrategia permite una colaboración eficiente, con sincronización parcial local y escritura periódica centralizada.

La arquitectura está diseñada para favorecer la escalabilidad, la tolerancia a fallos y la separación clara de responsabilidades entre los distintos componentes del sistema.

Diagrama de Componentes
El siguiente diagrama describe la estructura general del sistema, detallando las principales unidades funcionales y las relaciones entre ellas. El frontend, desarrollado con GTK4 y Relm4, interactúa con el backend a través de una capa de cliente que mantiene una conexión TCP con el servidor Redis. Este servidor central administra los documentos en memoria y en disco, y colabora con un microservicio encargado de enviar y reenviar comandos entre distintos nodos del clúster Redis. A su vez, los documentos se modelan internamente mediante estructuras especializadas que permiten representar tanto texto como hojas de cálculo.

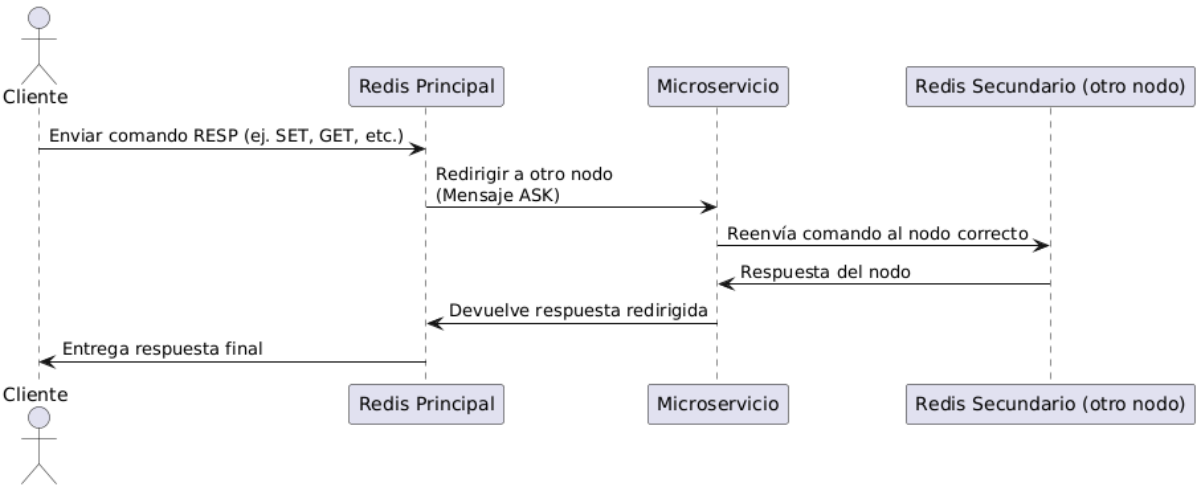


Flujo de la comunicación Cliente/Cluster

El siguiente diagrama representa el flujo de comunicación entre un cliente, un nodo principal de Redis, un microservicio intermediario y un nodo secundario dentro del clúster Redis distribuido. Cuando el cliente envía un comando al nodo principal, este puede determinar que la clave no le pertenece y responde con un mensaje de redirección (ASK).

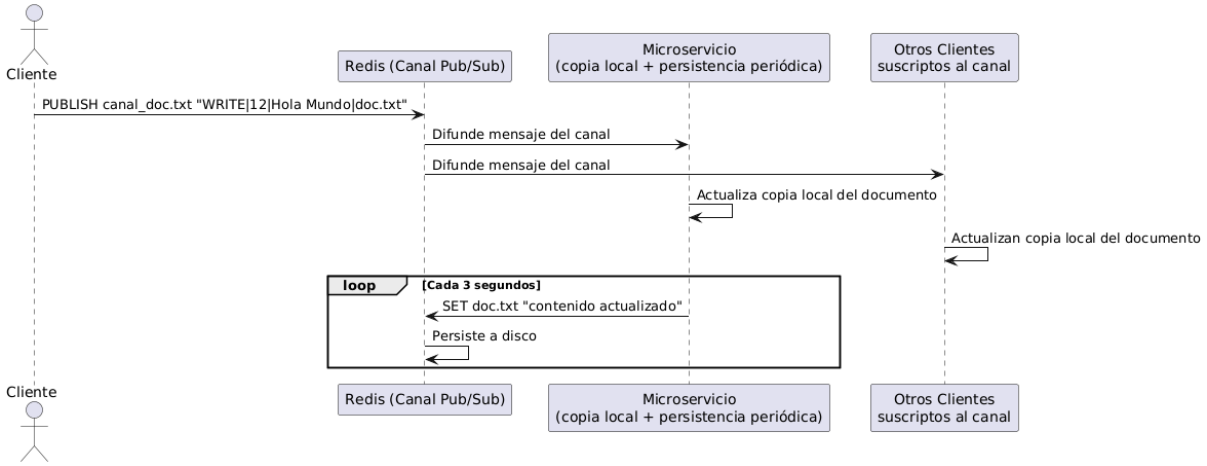
El microservicio recibe esa indicación y se encarga de reenviar el comando al nodo correcto. Una vez procesado, la respuesta viaja de vuelta al microservicio, luego al nodo principal y finalmente al cliente.

De esta forma, el cliente no necesita conocer la estructura interna del clúster: la redirección y el reenvío se resuelven de forma transparente mediante el microservicio.



Flujo Cliente/Cluster/Microservicio

El siguiente diagrama muestra el flujo de comunicación entre un cliente, el microservicio y el resto de los clientes suscriptos a un documento, utilizando el mecanismo de canales Pub/Sub de Redis. Cuando un cliente realiza una operación de escritura sobre un archivo, publica un mensaje en el canal correspondiente. Redis propaga este mensaje tanto al microservicio como a todos los demás clientes suscriptos a ese canal. Cada uno de ellos actualiza su copia local del documento en memoria, asegurando consistencia entre las réplicas distribuidas. Luego, el microservicio es responsable de persistir el estado actualizado del documento enviando, cada tres segundos a Redis el nuevo contenido. Esto permite que la versión persistida en disco se mantenga sincronizada con las copias locales de los nodos participantes.



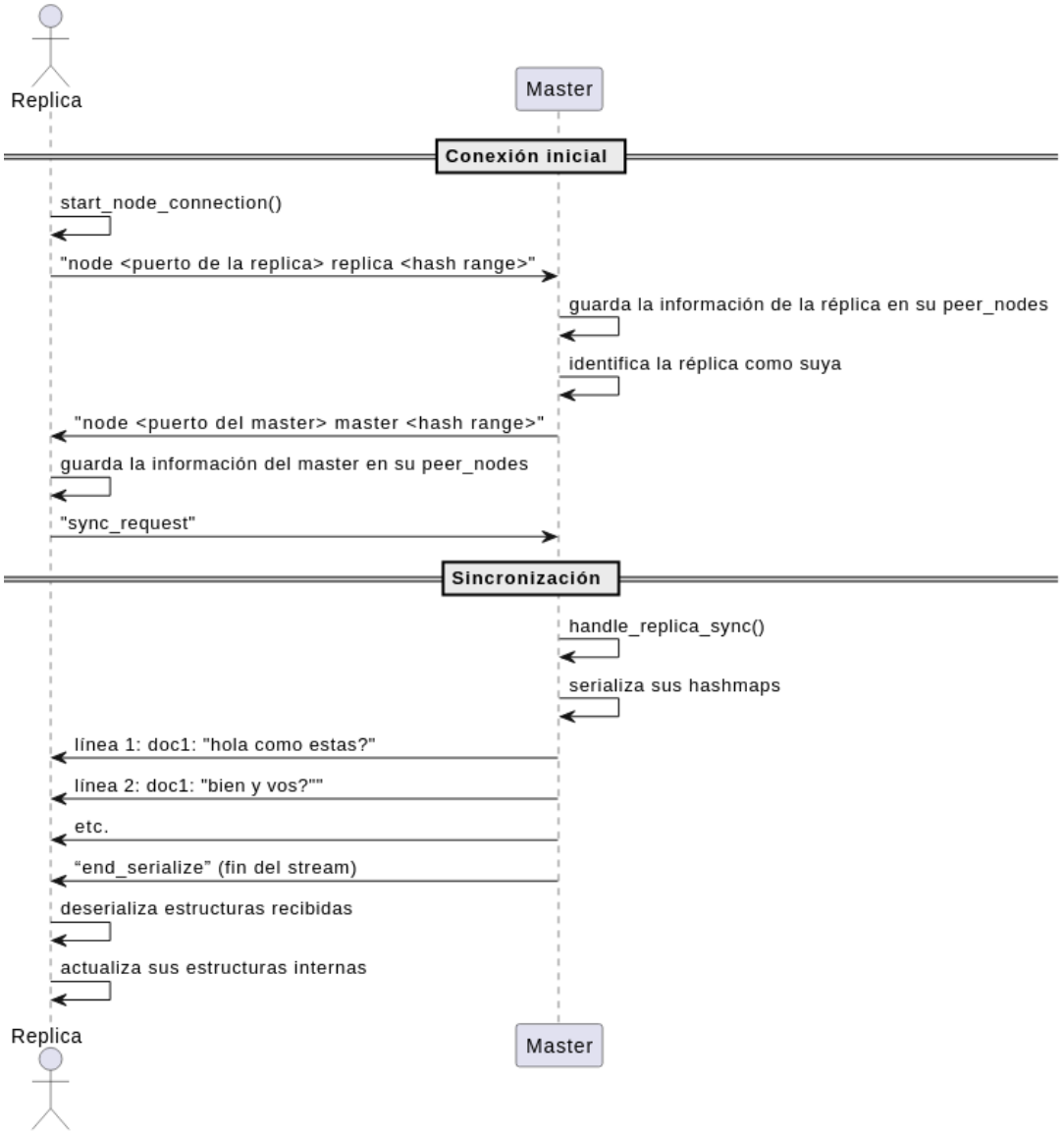
Sincronización de Réplicas

El siguiente diagrama de secuencia ilustra el procedimiento de sincronización entre un nodo master y una réplica que se conecta al clúster por primera vez. Inicialmente, la réplica intenta establecer conexión con todos los nodos del sistema. Al lograr conectarse con el nodo correspondiente a su rango de hash, intercambian información mediante el mensaje node, donde comunica su puerto, rol (réplica o master) y el intervalo de hash que manejan.

Al recibir esta información, el nodo maestro registra a la nueva réplica dentro de su estructura *peer_nodes* y la añade a su lista interna de réplicas. Por otro lado, la réplica identifica al nodo como su propio master, e inicia una solicitud de sincronización con el mensaje *sync_request*.

A partir de este punto, el maestro procede a serializar sus estructuras compartidas, como los hashmaps de las listas y los sets, transmitiéndolos por el stream TCP línea por línea. Para indicar el fin de esta transmisión, se envía un mensaje delimitador especial (*end_replica_sync*). La réplica receptora interpreta este marcador, deserializa la información y la incorpora en su propio estado local.

Este mecanismo asegura que toda réplica se integre al clúster con una copia del estado actual del master, facilitando la coherencia de datos y la preparación para asumir responsabilidades en caso de fallas futuras.



Detección de Fallos y Promoción

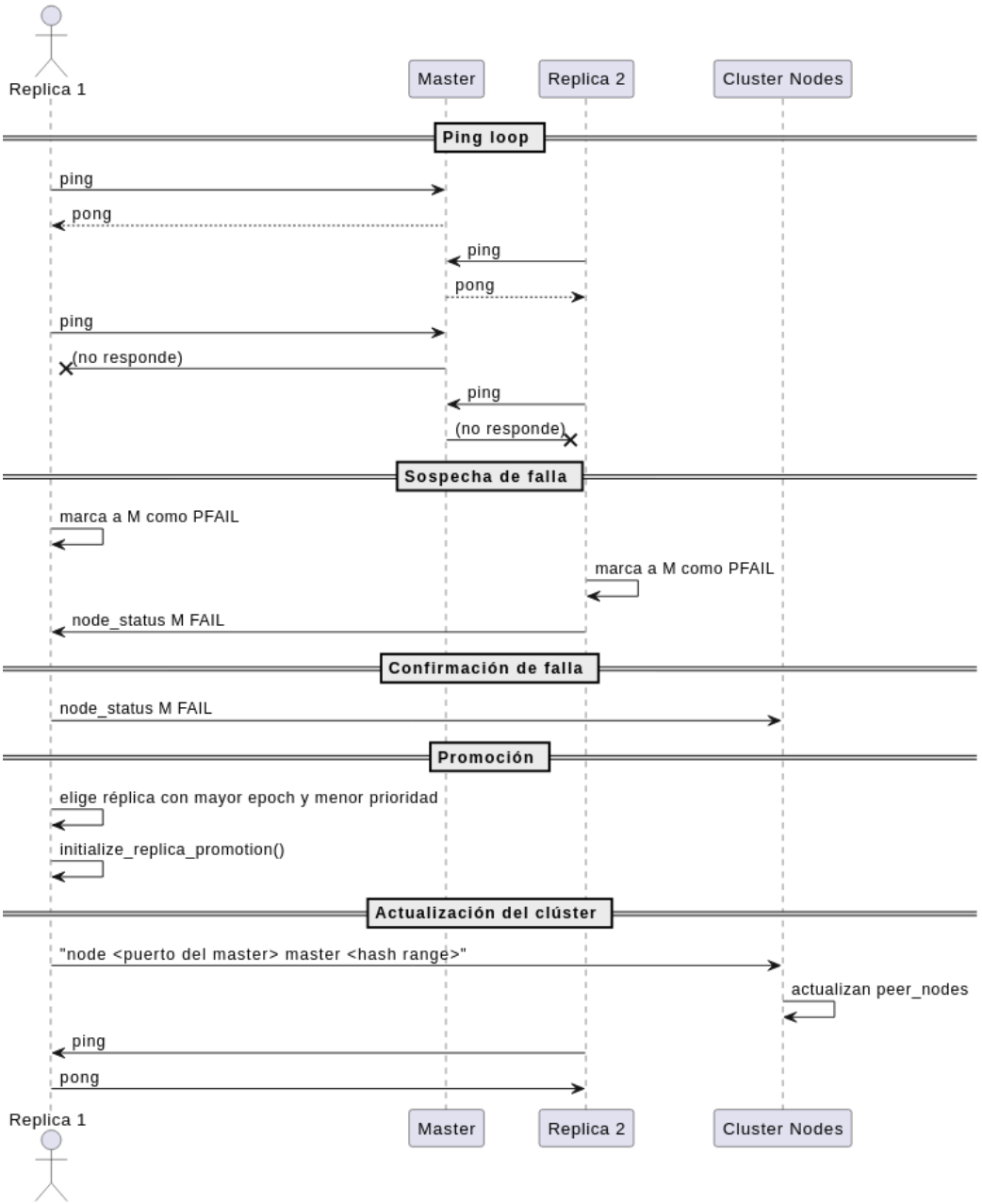
Este diagrama representa el proceso de detección de falla del nodo maestro y la posterior promoción de una réplica dentro del clúster Redis desarrollado. En el diseño propuesto, cada nodo mantiene una supervisión activa del resto utilizando el protocolo gossip para mandar pings periódicos. Este proceso es gestionado por la función *ping_to_nodes*, que intenta establecer comunicación regular con el nodo maestro correspondiente al rango de hash compartido.

Cuando un nodo detecta que otro ha dejado de responder, se marca como *PFAIL*, indicando que puede estar inactivo, y le avisa al resto del cluster a través de un mensaje de *node_state.* De ahí espera una confirmación externa: si recibe un mensaje *node_status* de otro nodo informando que también considera a ese nodo como inactivo, lo marca como *FAIL*. Luego, ese estado de *FAIL* también se propaga al resto del clúster a través del *node_status*.

En el caso de que el nodo marcado como FAIL sea un maestro, se inicia automáticamente el proceso de promoción de una réplica. La réplica a promover se elige de la siguiente manera: Se consideran

todas las réplicas del maestro fallido, se selecciona la que tenga el mayor epoch (número de versión lógica que representa la antigüedad y validez del nodo), y en caso de empate de epoch, se selecciona la réplica con mayor prioridad, la cual se define en la configuración del nodo.

La réplica seleccionada actualiza su rol a Master, utilizando la función *initialize_replica_promotion*. Luego, envía un nuevo mensaje node con su información actualizada como nuevo maestro, lo cual permite a los demás nodos ajustar su estructura interna (*peer_nodes*) y reconocer a la réplica promovida como autoridad en ese rango de hash. De esta manera se logra una recuperación automática y coordinada ante fallos, garantizando alta disponibilidad y consistencia en el sistema distribuido.



Conclusión

El desarrollo de este proyecto representó un gran desafío técnico y conceptual, que nos permitió poner en práctica múltiples conocimientos adquiridos durante la carrera. Nos obligó a comprender a fondo no sólo los principios detrás de bases de datos distribuidas, sino también aspectos complejos de programación concurrente, comunicación entre procesos y diseño de sistemas resilientes.

Uno de los mayores aprendizajes fue entender cómo interactúan los distintos componentes de un sistema distribuido en tiempo real, y cómo pequeñas decisiones en el diseño afectan el comportamiento global. La implementación manual de protocolos como RESP y de mecanismos como la promoción de réplicas o la sincronización de nodos nos brindó una perspectiva concreta sobre cómo funcionan sistemas en producción, muchas veces invisibles desde la abstracción de un cliente.

Más allá de lo técnico, también adquirimos experiencia valiosa en planificación, lectura de especificaciones ambiguas y toma de decisiones de diseño. Enfrentar errores de concurrencia, problemas de bloqueo mutuo y desafíos en la sincronización de información entre nodos nos forzó a razonar más allá del código, poniendo en juego habilidades de depuración y comunicación efectiva.

En definitiva, este proyecto no solo fortaleció nuestras habilidades como programadores, sino que también nos dejó herramientas prácticas para abordar futuros desafíos en el desarrollo de software distribuido y sistemas críticos.

Anexo - Evaluación Final

Docker

Modificaciones de la estructura original para el soporte de Docker

Con el objetivo de facilitar la ejecución y despliegue del sistema distribuido, se integró soporte para Docker en el proyecto.

El proyecto fue reestructurado para que cada componente (nodos del servidor, cliente, microservicios, etc.) tenga su propio workspace de Cargo, lo que permite compilar, testear y contenerizar cada parte por separado, mientras que los módulos y utilidades compartidas se movieron a un workspace común para reutilización entre los diferentes servicios.

Cada microservicio y nodo del servidor tiene su propio Dockerfile, lo que permite definir entornos de ejecución aislados para cada uno. Estos se juntan a través de un único docker-compose.yml que permite levantar todos los contenedores en conjunto y conectarlos fácilmente.

Estructura general de un contenedor Docker

Cada contenedor compila su propio workspace Rust con *cargo build --release* y expone un puerto específico configurado mediante variables de entorno.

Estructura del docker-compose

El archivo docker-compose.yml define los contenedores para los nodos del servidor y los microservicios, usando una red interna para que los nodos se puedan comunicar por nombre de contenedor (por ejemplo, node0:4000). También define volúmenes persistentes para almacenar archivos .rdb y logs. Para evitar errores relacionados con demasiadas conexiones o archivos abiertos, se aumentó el límite de recursos del sistema.

Cambios en el código para soportar Docker

Se realizaron varios ajustes en el código para que sea compatible con entornos dockerizados:

Primero, en lugar de usar 127.0.0.1:<puerto> para conectarse entre nodos, se utiliza node<n>:<puerto>, donde node<n> es el nombre del contenedor, permitiendo que Docker resuelva correctamente el nombre a una IP interna. El cliente sigue utilizando 127.0.0.1:<puerto> ya que no se corre con Docker si no que con la máquina host. Para soportar esto, los servidores pasaron a escuchar en 0.0.0.0:<puerto> en lugar de 127.0.0.1 para aceptar conexiones tanto desde otros contenedores como desde el host.

Además, se configuraron volúmenes para que los logs (server.log) y archivos de persistencia (.rdb) se puedan guardar fuera de los contenedores y no se pierdan al reiniciar.

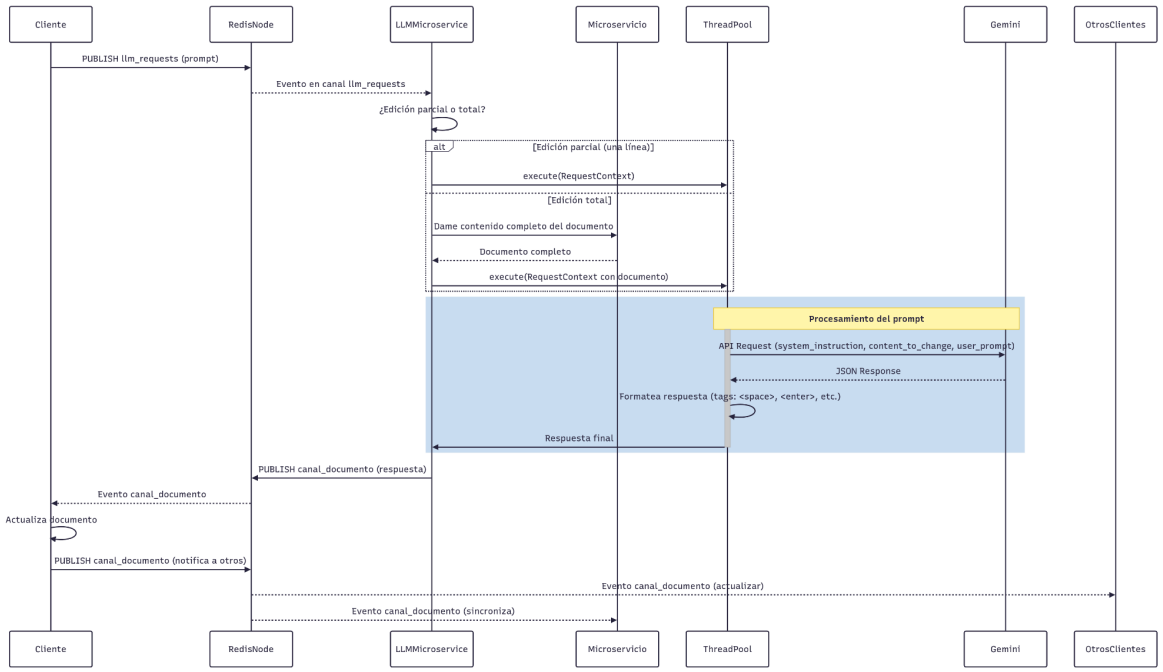
Microservicio de Inteligencia Artificial (LLM)

Objetivo

El microservicio de AI tiene como objetivo asistir al usuario durante la edición del documento, generando o corrigiendo contenido textual a partir de prompts enviados por los clientes. Su integración permite funcionalidades como corrección ortográfica, traducción, reescritura, generación de texto, entre otras.

Flujo de funcionamiento

1. El usuario ingresa un **prompt** desde la interfaz gráfica.
2. El prompt se envía al **canal Pub/Sub del documento** correspondiente.
3. El **microservicio de AI**, suscrito a ese canal, detecta el mensaje y realiza una petición al proveedor externo (Gemini).
4. Una vez obtenida la respuesta, el microservicio publica el texto generado en el mismo canal.
5. El **cliente que originó el prompt** recibe la respuesta y:
    - Inserta el texto automáticamente en la posición actual del cursor.
    - Publica el documento actualizado al canal, permitiendo que los demás clientes y el microservicio de control sincronicen su estado.



Consideraciones de diseño

- **Ubicación de inserción**: el texto generado se inserta en la **posición actual del cursor**, sin borrar contenido preexistente.

- **No se implementó confirmación (Aceptar/Rechazar)**, ya que no era requerida por el enunciado ni por los docentes.

- En caso de prompts generales (ej. “traducí el documento completo”), el microservicio de AI solicita el contenido al microservicio de control mediante Pub/Sub.

- El **microservicio de AI no escribe directamente en Redis**, respetando la separación de responsabilidades: los clientes son quienes publican las versiones definitivas del documento.

Análisis de costos

Considerando el uso estimado del sistema en un entorno real:

- **Cantidad esperada de prompts diarios**: 500

- **Longitud promedio de cada prompt**: 200 tokens

- **Longitud promedio de respuesta esperada**: 500 tokens

- **Total de tokens procesados por día**:
  500 * (200 + 500) = 350.000 tokens/día

Según la estructura de precios pública de Gemini:

- El precio estimado para **700.000 tokens (input + output)** diarios ronda los **USD 0.70 a USD 1.20 por día**, lo cual da un **costo mensual estimado entre USD 21 y USD 36**.

Conclusión: **el sistema es económicamente viable**, con escalabilidad controlada, y puede ajustarse a límites presupuestarios mediante restricciones de uso en UI.

Sincronización y consistencia

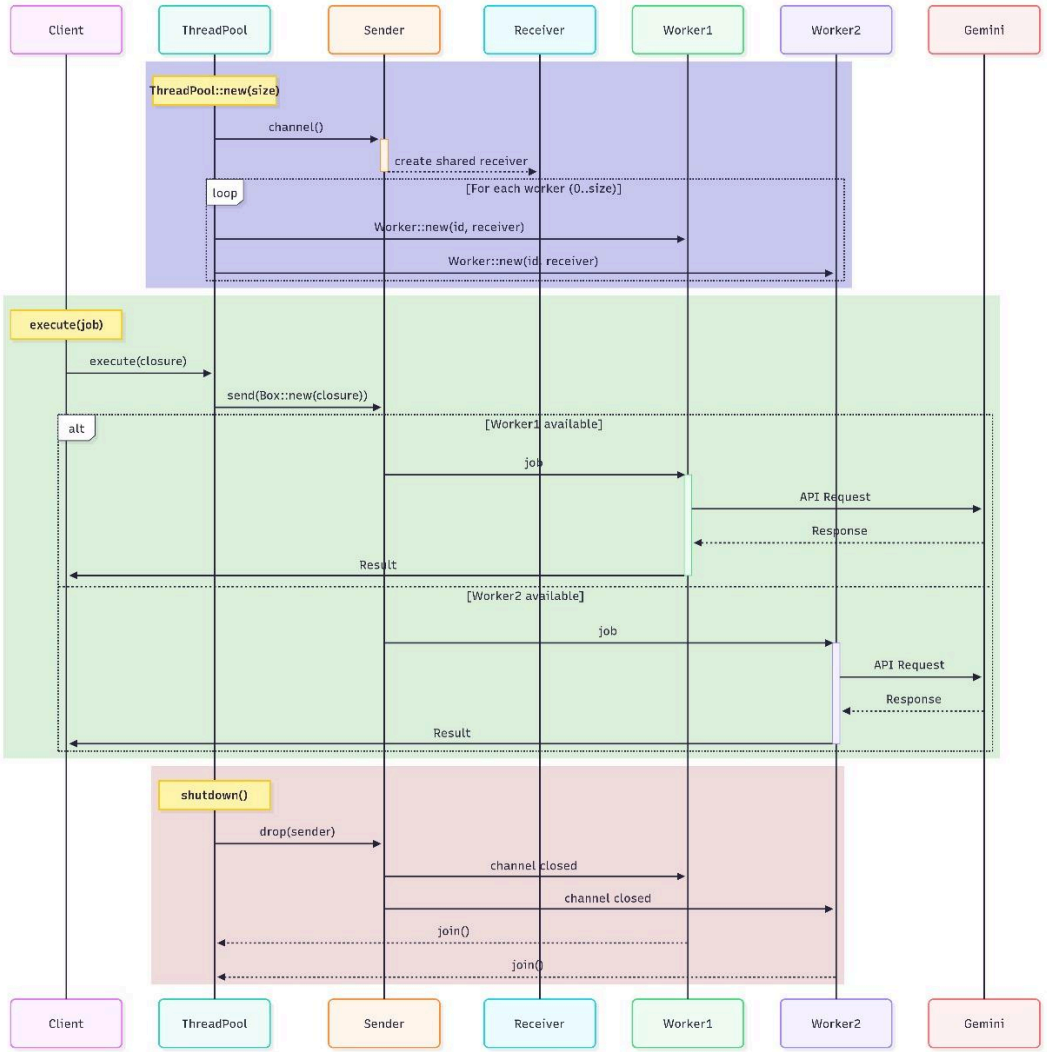
Durante el procesamiento del prompt, el cliente bloquea temporalmente el cursor para evitar inconsistencias. No se implementa un sistema de resolución de conflictos entre ediciones simultáneas, por lo que se asume que el usuario será quien decida cuándo sincronizar el contenido final tras recibir la respuesta del modelo.

Multithreading y concurrencia

El microservicio de Inteligencia Artificial fue diseñado para procesar múltiples prompts de usuarios de manera concurrente, sin bloquear el flujo general del sistema. Para lograr esto, se implementó un **pool de hilos (thread pool)**.

Funcionamiento del thread pool:

- El microservicio mantiene un conjunto fijo de hilos trabajadores.
- Cada mensaje recibido desde el canal Redis Pub/Sub es enviado a una **cola de trabajo**, desde la cual los hilos activos toman tareas de forma concurrente.
- Cada hilo procesa el prompt, realiza la petición HTTP al proveedor Gemini, espera la respuesta y la publica en el canal Redis del documento correspondiente.
- Este enfoque permite que varios prompts sean procesados simultáneamente, sin bloquear la recepción de nuevos mensajes o congestionar la red.



Beneficios del enfoque:

- **Paralelismo real**: múltiples solicitudes pueden estar en curso al mismo tiempo, optimizando la latencia percibida por los usuarios.

- **Control de recursos**: al definir un número fijo de hilos, se evita el consumo excesivo de CPU y memoria en momentos de alta carga.

- **Aislamiento de errores**: un fallo en el procesamiento de una tarea no afecta al resto de los procesos.

- **Escalabilidad modular**: la arquitectura permite aumentar o reducir el tamaño del pool según la capacidad del entorno de despliegue.

Justificación de diseño

Decisión	Justificación
Inserción automática del texto generado	Agiliza la experiencia del usuario; evita una etapa adicional innecesaria
Publicación del documento solo por el cliente	Centraliza la lógica de control y evita condiciones de carrera
Comunicación por Pub/Sub	Desacopla los servicios, favorece la escalabilidad y trazabilidad del flujo
Multithreading	Permite manejar múltiples pedidos simultáneamente sin latencia perceptible