.UBAfiuba
FACULTAD DE INGENIERÍA

# Taller de Programación I

# RustiDocs - 1C2025

Integrantes:


| Nombre | Padrón |
|---|---|
| Camila General | 105552 |
| Ramiro Mantero | 107163 |
| 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 de Redis distribuido utilizando el lenguaje Rust, con el objetivo de replicar las funcionalidades básicas de Redis Cluster. La arquitectura se organiza en tres niveles principales: nodos Redis, clientes y un microservicio de control y persistencia. Todos los componentes se comunican a través de sockets TCP utilizando el protocolo RESP, con soporte para Pub/Sub, comandos clave-valor y mecanismos de replicación.

Cada nodo del clúster puede actuar como maestro o réplica, y se le asigna un rango de hash slots que determina qué claves maneja. Los nodos se comunican entre sí para compartir información de estado, sincronizar datos y realizar detección de fallos. Cuando un nodo maestro falla, las réplicas que lo acompañan ejecutan un protocolo de promoción para reemplazarlo.

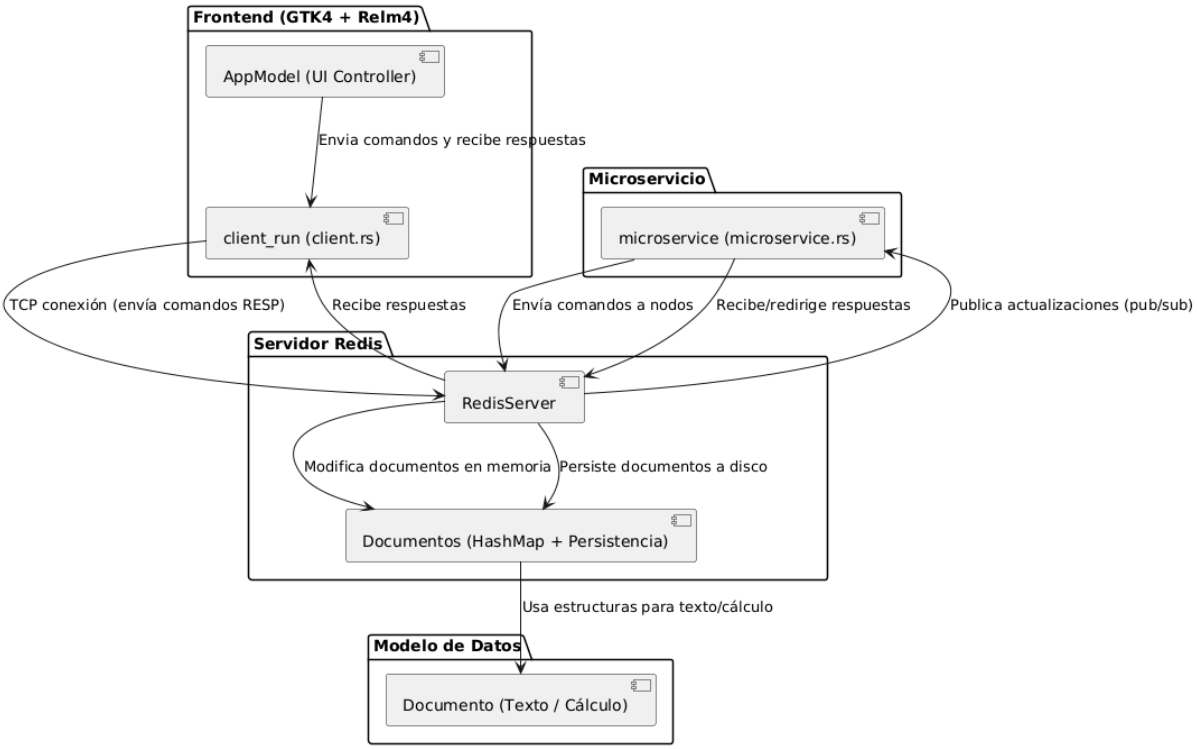
El sistema también incluye una capa de hashing basada en CRC16 para distribuir claves entre los distintos nodos. Este hashing se aplica al nombre de la clave, de acuerdo con las reglas del protocolo Redis Cluster, para determinar qué nodo debe manejar una operación dada.

A su vez, se incorpora un microservicio de control y persistencia, que se suscribe a los canales de documentos y responde a eventos del Pub/Sub. Este microservicio no tiene interfaz gráfica, pero mantiene un seguimiento del estado de colaboración en cada documento y se encarga de persistir los contenidos de manera periódica en las estructuras internas del clúster.

Finalmente, los clientes pueden conectarse a cualquier nodo y enviar comandos RESP estándar. Si el nodo al que se conectan no es responsable de la clave solicitada, este responde con un mensaje de redirección tipo ASK, de acuerdo al comportamiento esperado en Redis Cluster.

Esta arquitectura busca balancear escalabilidad, tolerancia a fallos y modularidad, priorizando un diseño comprensible y mantenible.

**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.

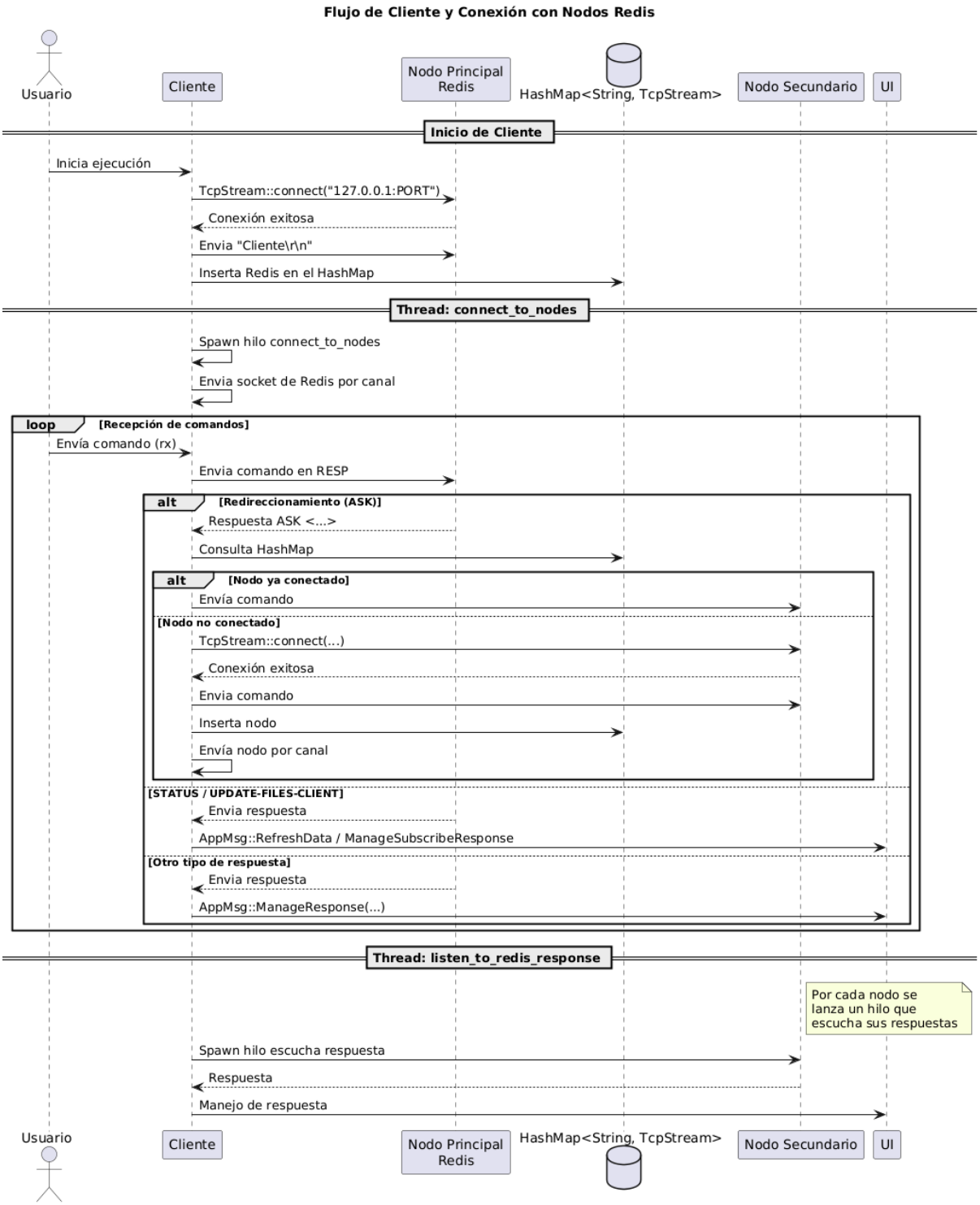



**Flujo de la comunicación Cliente/Cluster**

El diagrama muestra cómo funciona el cliente en un sistema distribuido basado en Redis. Al iniciar, el cliente se conecta al nodo principal y envía un mensaje de identificación. Esta conexión se guarda en un HashMap junto con otras posibles conexiones a nodos.

Luego, se lanza un hilo que escucha las respuestas de cada nodo conectado. A medida que el usuario envía comandos, el cliente los convierte al formato RESP y los envía. Si Redis responde con una redirección (ASK), el cliente verifica si ya está conectado al nodo destino: si no lo está, crea la conexión y reenvía el comando.

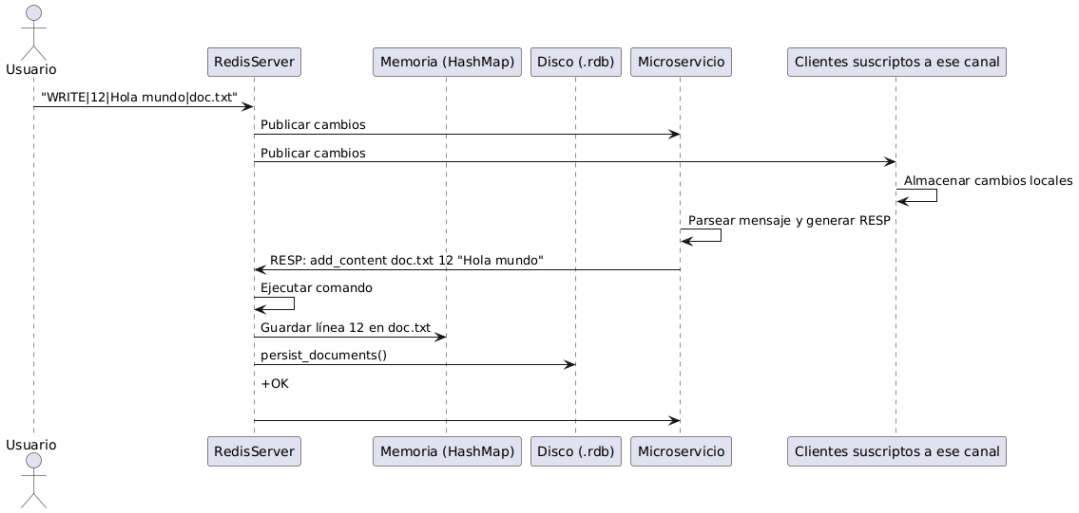
Cada nodo tiene un hilo que lee sus respuestas y las envía a la interfaz de usuario. Así, el cliente puede interactuar con múltiples nodos Redis de forma dinámica y paralela.




### Flujo Cliente/Cluster/Microservicio

En el siguiente diagrama se muestra cómo se procesa una operación de escritura enviada por el usuario, involucrando tanto al servidor Redis como al microservicio y a los clientes suscritos al documento afectado. Al recibir el comando `WRITE`, el servidor Redis se encarga de propagar la actualización al microservicio y a los suscriptores del canal correspondiente. El microservicio convierte la instrucción al formato RESP y la reenvía al servidor para su ejecución. Redis actualiza el contenido en memoria, persiste los datos en disco y confirma la operación. Este flujo garantiza la

consistencia entre los distintos nodos del sistema, manteniendo sincronizados los clientes que visualizan el mismo documento.



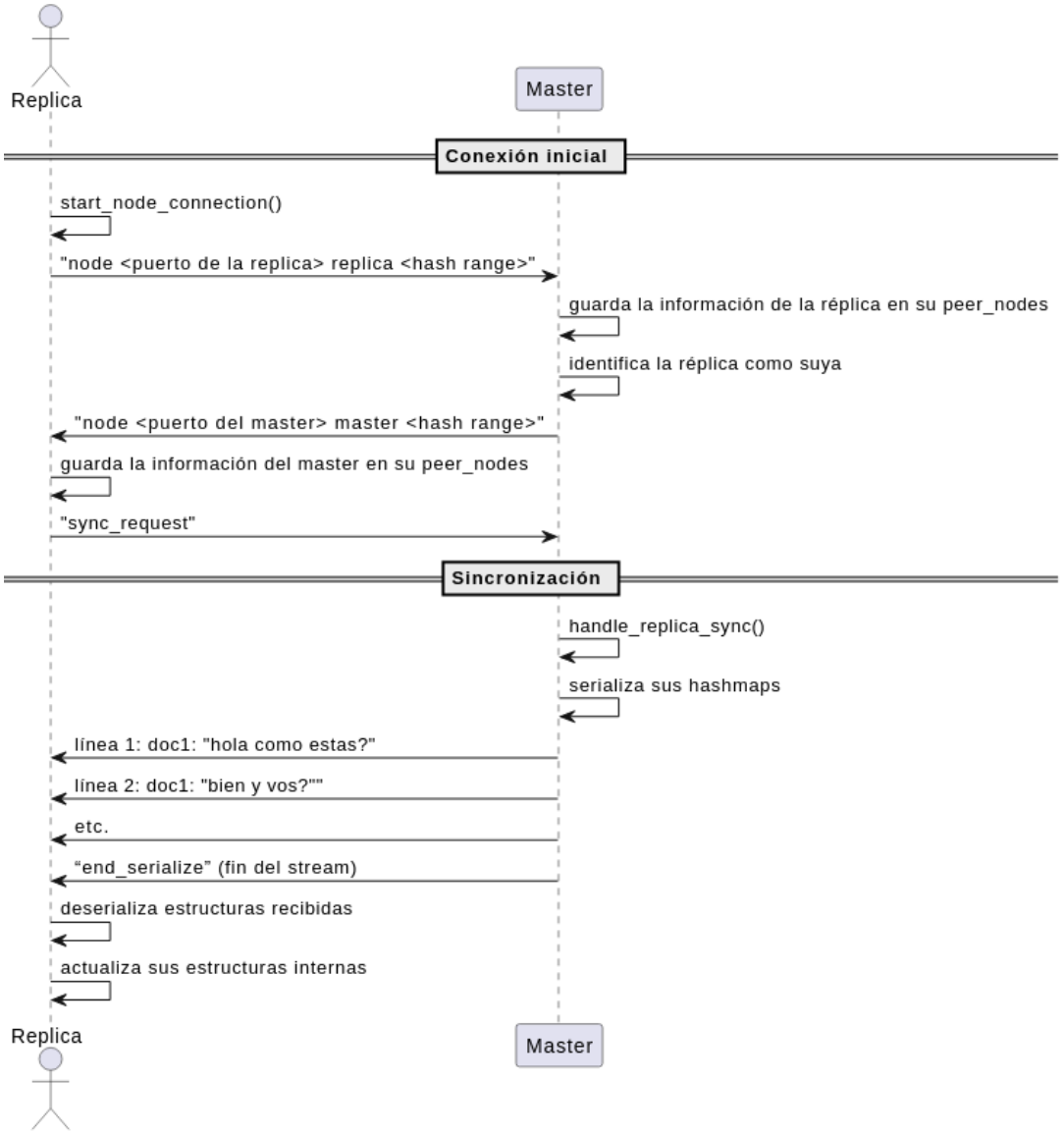

**Sincronización de Réplicas**

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.

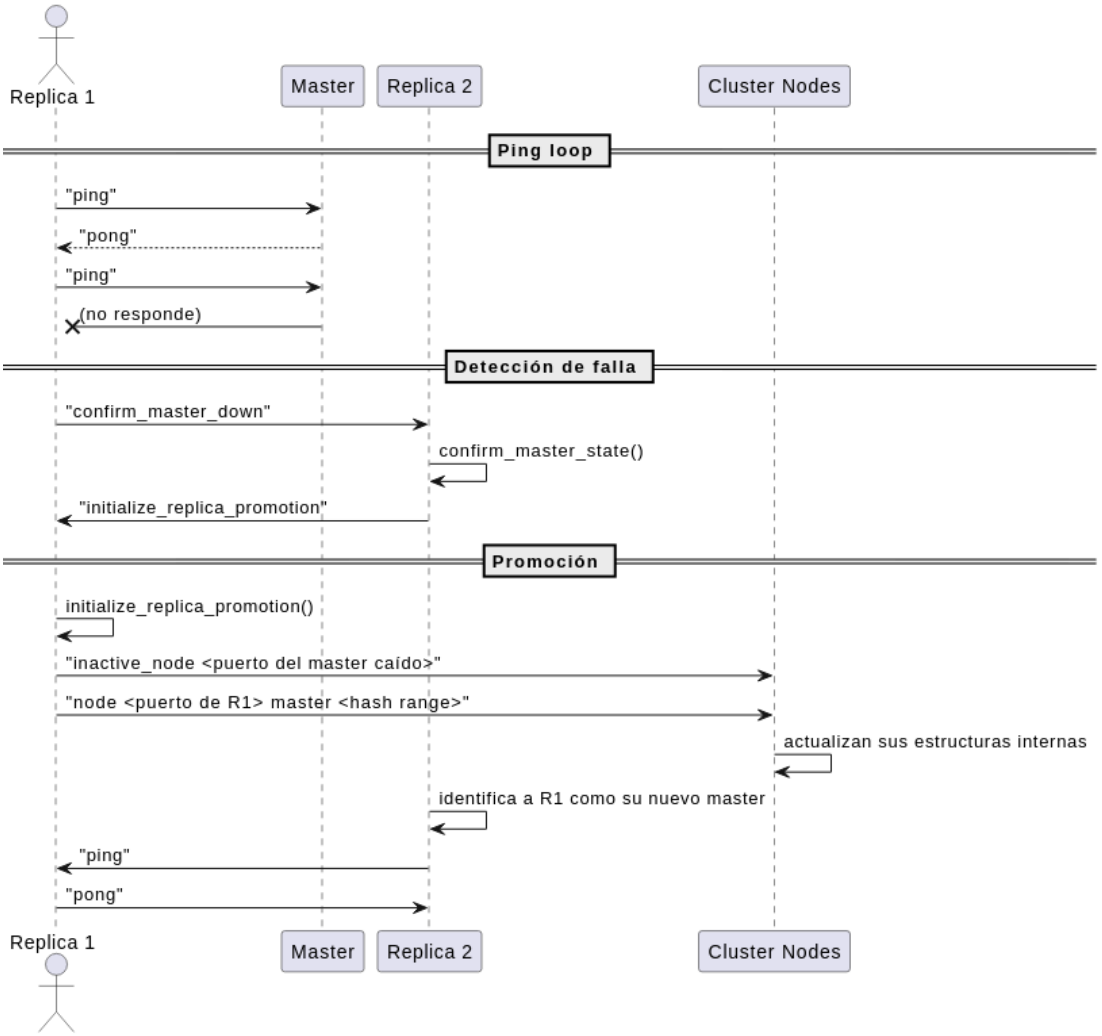



**Detección de Fallos y Promoción**

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 réplica mantiene una supervisión activa del maestro utilizando el protocolo gossip para mandar pings periódicos. Este proceso es gestionado por la función *ping_to_master*, que intenta establecer comunicación regular con el nodo maestro correspondiente al rango de hash compartido.

Cuando una réplica detecta que el maestro ha dejado de responder, inicia un proceso de verificación enviando un mensaje *confirm_master_down* a la otra réplica del mismo rango de hash. Esta segunda réplica verifica el estado del maestro en su propia estructura de nodos y responde indicando si también considera que el maestro está inactivo. Si ambas coinciden en que el maestro ha fallado, la réplica inicial procede a su promoción a nuevo nodo maestro utilizando la función *initialize_replica_promotion*. Si no se encuentra otra réplica, la primera réplica continua con el proceso de promoción sin confirmación externa.

En esta etapa, la réplica actualiza su rol local y comunica al resto del clúster que el antiguo maestro se encuentra inactivo a través de un mensaje *inactive_node*. 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.




## Mensajes Cliente - Microservicio

## 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.