

# Coffee Shop Analyzer

## TP Tolerancia a Fallas - Grupo 14

4 + 1 Views

Sistemas distribuidos I (75.64 / TA050)



Fecha de Entrega: 04/12/2025

# Índice

<b>Introducción.....</b>	<b>4</b>
<b>Casos de uso.....</b>	<b>4</b>
<b>Arquitectura de la solución.....</b>	<b>5</b>
1. Capa de Ingesta.....	5
2. Capa de Comunicación.....	5
2.1. Tipo de colas.....	5
3. Capa de Filtrado.....	6
4. Capa de Agregación.....	6
4.1. Características principales:.....	6
5. Capa de Consolidación.....	6
<b>Vista Lógica.....</b>	<b>7</b>
Componentes de la aplicación.....	7
Flujo de las tablas según query (DAG).....	8
<b>Vista de desarrollo.....</b>	<b>9</b>
Diagrama de paquetes.....	9
<b>Vista de procesos.....</b>	<b>11</b>
Comportamiento básico de filter router, joiner router y joiner worker.....	11
Pipeline de mensaje CleanUp.....	12
Pipeline de mensaje EOF.....	14
Flujo del filter router.....	15
Flujo del Joiner Worker.....	16
Flujo del Results Finisher.....	17
<b>Vista Física.....</b>	<b>18</b>
Diagrama de robustez.....	18
Diagrama de despliegue.....	19
<b>Cambios principales.....</b>	<b>20</b>
Evolución y Estructura de Mensajes (Protocolo V2).....	20
Envoltorio Universal (Envelope).....	20
Estandarización de Datos (DataBatch y TableData).....	20
Unificación de Resultados (QueryResult).....	21
Control de Flujo y Ciclo de Vida (EOF y CleanUp).....	21
Coordinación y Tolerancia a Fallos (Nuevos Mensajes).....	21
Mensaje TableMessage.....	21
Tipos principales de TableMessage.....	21
Estructura general de TableMessage.....	22
Características especiales.....	22
Alta Disponibilidad y Elección de Líder.....	22
El Rol del Líder (Watchdog Pattern).....	22
Algoritmo de Elección: Bully Algorithm.....	23
Detección de Fallos mediante Heartbeats.....	23
Recuperación Automática (Self-Healing).....	24

Configuración Parametrizable.....	24
<b>Métricas.....</b>	<b>25</b>
Especificaciones del dispositivo.....	25
Análisis de Escalabilidad por Volumen de Datos (Dataset Reducido vs. Completo).....	25
Dimensionamiento de los Datasets.....	26
Resultados Cuantitativos.....	26
Eficiencia Sostenida.....	26
Amortización de Overhead.....	26
Estabilidad en Joins.....	26
Estabilidad y Límites de Carga (Dataset Reducido vs. Completo).....	27
Puntos de Presión Arquitectónica y Evidencia (RabbitMQ):.....	27
Single-Client vs. Multi-Client.....	29
Escenario Ideal (Sin Errores): Escalabilidad Lineal.....	29
Escenario Crítico (Con Inyección de Fallos): Degradación Super-lineal.....	29
Conclusión sobre la Concurrencia.....	30
Robustez y Comportamiento bajo Estrés.....	30
<b>Lecciones aprendidas.....</b>	<b>31</b>
Simplificación de Filter Router.....	31
Protocolo Interno.....	33
Análisis de Impacto: Migración a Protobuf y Tolerancia a Fallos.....	33
Tabla de Resultados.....	34
Conclusiones Técnicas.....	34
<b>Conclusiones.....</b>	<b>35</b>
Impacto Crítico de la Optimización del Protocolo.....	35
Escalabilidad Lineal y Cuellos de Botella Físicos.....	35
Resiliencia y Auto-Recuperación.....	35

# Introducción

Este documento describe la arquitectura de una plataforma distribuida de procesamiento de datos, diseñada para el análisis de información de transacciones de una cadena de cafeterías ubicada en Malasia. La solución aborda la necesidad de procesar grandes volúmenes de datos para generar insights de negocio.

El sistema se enmarca en un entorno donde los datos transaccionales se encuentran distribuidos en múltiples archivos CSV que contienen información de ventas, clientes, productos y sucursales. Estos datos deben ser procesados de manera eficiente para responder consultas analíticas complejas que requieren filtrado, agregaciones y joins entre múltiples entidades. Para manejar esta complejidad, el sistema se ha diseñado siguiendo el patrón de arquitectura **pipe and filters** (tuberías y filtros). Este enfoque nos permite procesar los datos de manera eficiente a través de una serie de etapas independientes y desacopladas. Cada "filtro" realiza una tarea específica, como la limpieza, transformación, enriquecimiento o agregación de los datos, mientras que las "tuberías" se encargan de comunicar la información entre estos componentes. Este diseño modular no solo facilita el desarrollo y la escalabilidad, sino que también permite responder a consultas analíticas complejas que requieren filtrado, agregaciones y uniones (*joins*) entre múltiples entidades de manera ordenada y robusta.

## Casos de uso

Como usuario quiero poder obtener:

- Todas las transacciones realizadas durante 2024 y 2025 entre las 06:00 AM y las 11:00 PM con monto total mayor o igual a 75.
- Los productos más vendidos y los productos que más ganancias han generado, para cada mes en 2024 y 2025.
- El TPV por cada semestre en 2024 y 2025, para cada sucursal, para transacciones realizadas entre las 06:00 AM y las 11:00 PM.
- La fecha de cumpleaños de los 3 clientes que han hecho más compras durante 2024 y 2025, para cada sucursal.

# Arquitectura de la solución

La arquitectura del sistema se organiza en cinco capas especializadas, que en conjunto implementan un patrón pipe and filters. Este diseño segmenta el procesamiento en etapas independientes y desacopladas, permitiendo escalabilidad horizontal, reutilización de componentes y resiliencia frente a fallos. Cada capa funciona como un filtro que transforma los datos en lotes (batches), mientras que las tuberías de comunicación aseguran un flujo confiable entre etapas.

## 1. Capa de Ingesta

Punto único de entrada del sistema. Un orquestador recibe conexiones TCP de los clientes, gestiona el ciclo de vida de las consultas y coordina la preparación de los lotes de datos crudos.

- Funciones clave: control de protocolo, encolado inicial, enriquecimiento con metadata, filtrado de columnas innecesarias.
- Beneficio: centraliza la lógica de entrada, permitiendo aislar la complejidad de los clientes.

## 2. Capa de Comunicación

Implementada mediante un middleware de colas de mensajes, esta capa desacopla productores y consumidores.

- Funciones clave: asincronía, balanceo de carga, tolerancia a fallos.
- Beneficio: asegura que el sistema pueda absorber picos de tráfico y que cada componente evolucione de forma independiente.

### 2.1. Tipo de colas

- Queue: cola tradicional de RabbitMQ.
  - Los mensajes se envían directamente a una cola con nombre fijo (queue\_name).
  - Los consumidores leen los mensajes en orden FIFO.
  - La cola es durable (sobrevive reinicios del broker).
- Exchange: intercambio de tipo "topic".
  - Los mensajes se publican en un exchange y se enrutan a una o varias colas según la clave de enrutamiento (routing\_key).

- Las colas consumidoras pueden ser temporales y exclusivas (solo accesibles por una conexión y eliminadas al cerrarse).
- Permite patrones flexibles de distribución de mensajes (broadcast, filtrado por temas, etc.).

### 3. Capa de Filtrado

Un pool de workers genéricos aplica validaciones y transformaciones iniciales sobre los datos.

- Funciones clave: limpieza, normalización, aplicación de reglas de negocio.
- Beneficio: simplifica las etapas posteriores, reduciendo volumen y complejidad.

### 4. Capa de Agregación

La capa de agregación está implementada por la clase Aggregator, que actúa como worker especializado para el procesamiento distribuido de datos provenientes de distintas fuentes.

- Funciones clave: sumas, conteos, promedios y métricas intermedias.
- Beneficio: permite procesamiento paralelo y escalable, optimizando el tiempo de respuesta en consultas analíticas.

#### 4.1. Características principales:

- Procesamiento distribuido y especializado: cada instancia de Aggregator consume datos desde exchanges específicos, procesando mensajes según el tipo de consulta asociada.
- Estado en memoria: El worker mantiene el estado necesario para procesar los datos recibidos, permitiendo cálculos intermedios y agregaciones antes de reenviar los resultados.

### 5. Capa de Consolidación

Etapa final del pipeline, donde se combinan los resultados parciales y se realizan los joins entre tablas.

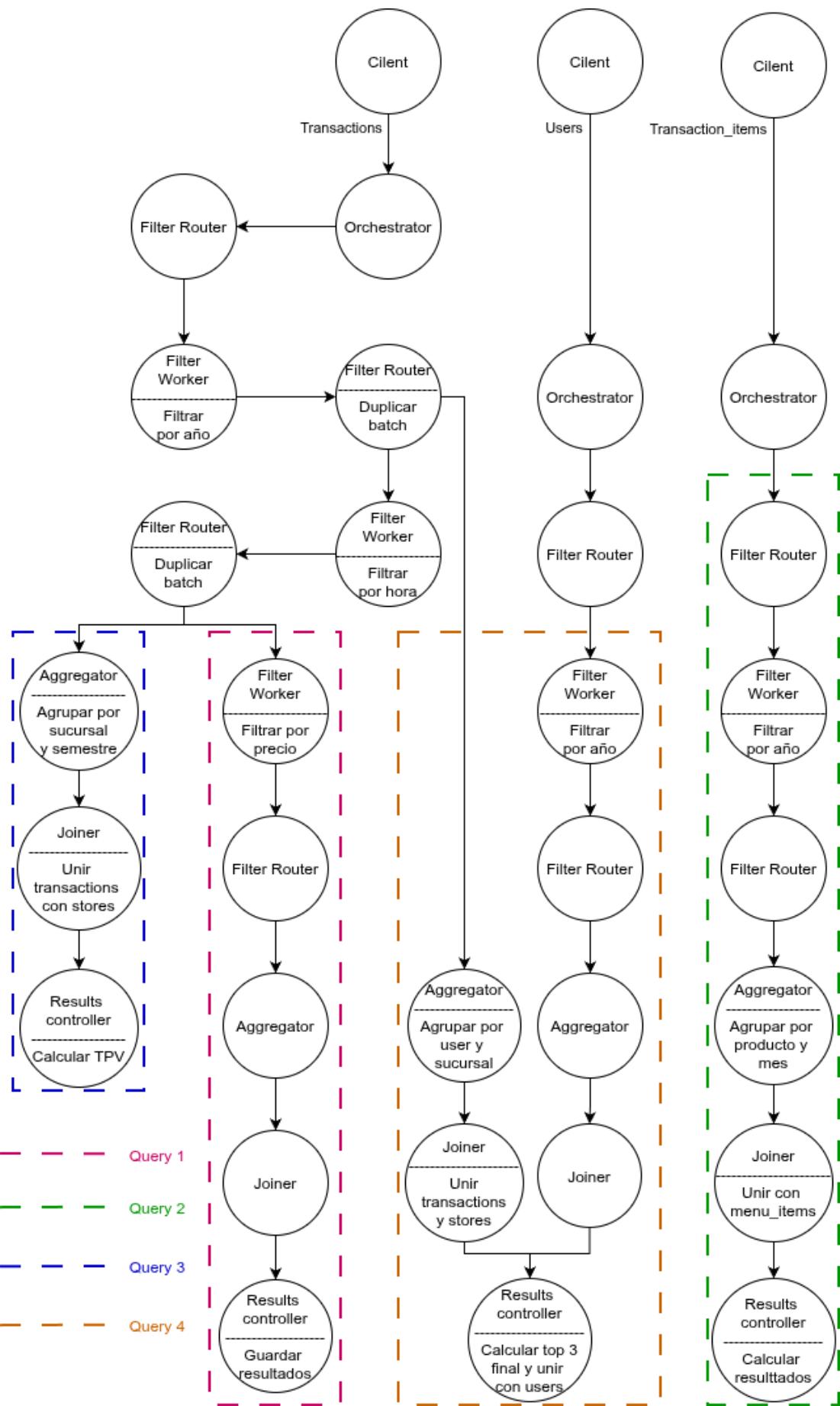
- Funciones clave: uniones tardías (late joins), consolidación de resultados, armado de la respuesta final.
- Beneficio: otorga flexibilidad para manejar datos heterogéneos y consultas complejas sin penalizar etapas previas.

# Vista Lógica

## Componentes de la aplicación

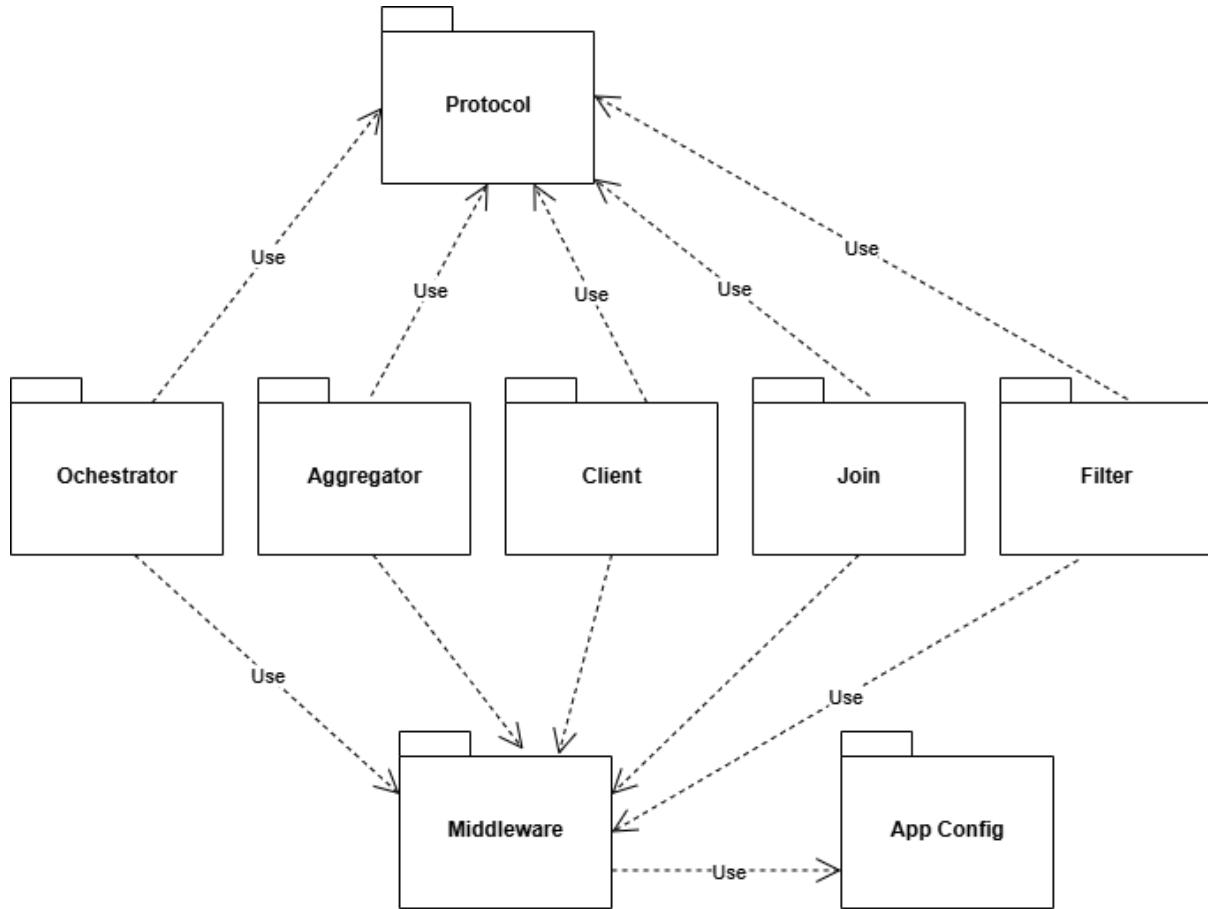
- **Cliente TCP:** levanta los archivos y los envía en batches.
- **Orquestador:** gestiona las conexiones con los clientes. Conoce el protocolo a utilizar (ej Query + Batches). Recibe los batches de data de parte de ellos. Prepara los batches de forma tal que estén listos para ser procesados por los distintos workers (escribe la metadata necesaria para que cada worker entienda que pasos ejecutar por ej). A futuro podría también tener data sobre en qué estado quedó la consulta (caso donde el cliente se desconecta y se vuelve a conectar). También recibe los resultados y se los pasa al cliente. Filtra las columnas que no se van a usar.
- **Middleware:** colas de mensajes para posibilitar la comunicación async entre componentes.
- **Filter Router:** deriva los mensajes según se necesite filtrar o pasar al paso de agregación.
- **Filter Worker:** procesan y transforman la información. Lo más importante es que cuentan con un “dispatcher” de filtros, donde el worker puede elegir que filtro usar basado en la metadata. Trabajan de a lotes.
- **Aggregator:** generan agregaciones parciales de los batches que llegan. Misma estrategia que los FilterWorker en un principio.
- **Joiner Router:** splitea los datos de las tablas segun cierto campo (primary o foreign keys), para enviar las porciones a distintos joiners, que se encargan de juntar distintas partes de los datos.
- **Joiner Worker:** join entre distintas tablas para tener todos los datos necesarios.}
- **Results Router:** distribuye al results worker adecuado dependiendo una key num query e id de cliente
- **Results Finisher:** agrupa los resultados que van a mandar los agregadores para llegar a, justamente, el resultado. También hace un join tardío entre transaction\_items\_menu\_items y users (por eficiencia).

## Flujo de las tablas según query (DAG)



# Vista de desarrollo

## Diagrama de paquetes



El diagrama de paquetes ilustra la arquitectura modular de la aplicación, la cual se descompone en los siguientes componentes principales de alto nivel: client, aggregator, filter, joiner, orchestrator, protocol y app\_config.

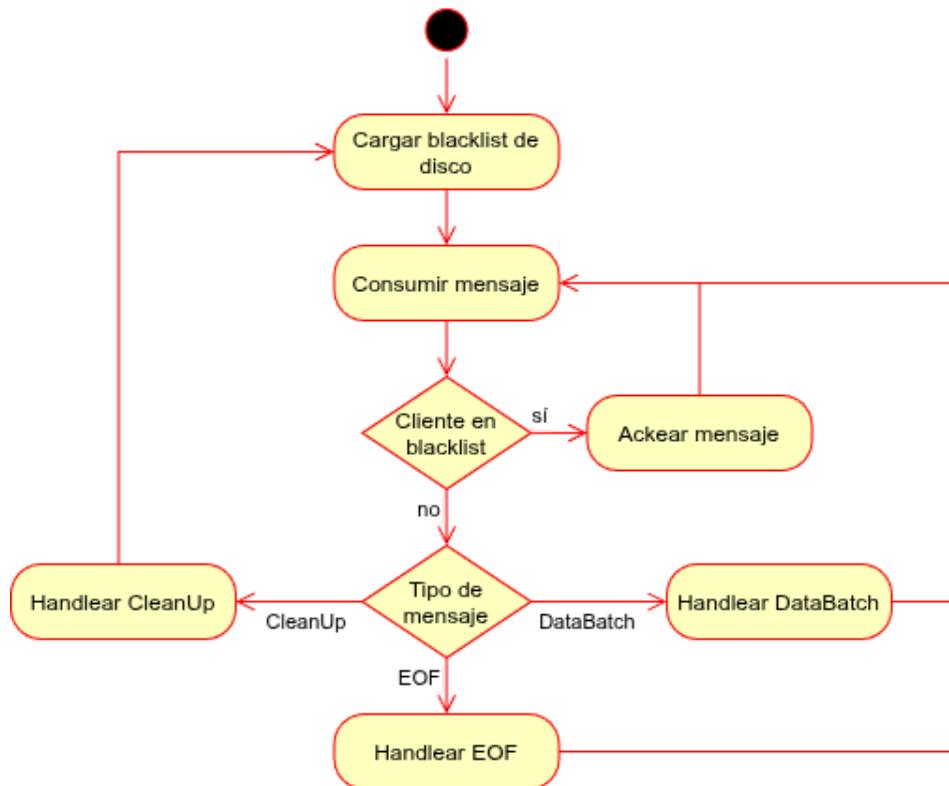
- Protocol: actúa como una librería base y central, conteniendo todas las estructuras de datos, constantes y modelos compartidos que se utilizan a lo largo del sistema. Este paquete es fundamental, ya que garantiza que todos los demás componentes se comuniquen utilizando un protocolo de datos coherente.
- Client: representa la aplicación de entrada. Su responsabilidad es interactuar con el usuario, leer los datos de origen y prepararlos para su envío al backend.
- Aggregator: contiene la lógica para el procesamiento distribuido y la agregación de datos. Se encarga de realizar cálculos intermedios, sumas, conteos y promedios sobre los datos recibidos.

- Filter: implementa la etapa de filtrado de datos, permitiendo seleccionar y transformar la información relevante antes de su procesamiento posterior.
- Joiner: se encarga de la unión y consolidación de datos provenientes de diferentes fuentes, preparando los resultados para su análisis final.
- Orchestrator: coordina el flujo de datos y la interacción entre los distintos módulos del sistema, gestionando el orden y la sincronización de las etapas de procesamiento. Cuenta con un pool de workers, un objeto que consume los resultados listos para entregar al cliente, y un objeto de manejo de mensajes.
- Middleware: proporciona las abstracciones para la comunicación asíncrona entre componentes, principalmente mediante RabbitMQ.
- App\_config: centraliza la configuración del sistema, permitiendo que los demás módulos obtengan parámetros como hosts, nombres de colas, exchanges y rutas de manera consistente y desacoplada.

Todos los paquetes principales (client, aggregator, filter, joiner, orchestrator) dependen de protocol para la definición de datos y mensajes, y utilizan middleware para la comunicación. Además, app\_config es utilizado por los módulos para acceder a la configuración centralizada. Esta arquitectura asegura que, aunque los componentes son independientes y modulares, todos operan sobre una base común y se comunican de forma coherente y eficiente.

## Vista de procesos

Comportamiento básico de filter router, joiner router y joiner worker



Estos tres son los componentes más complejos, junto con el results finisher. El resto de componentes tienen un flujo muy parecido, pero no tienen blacklist. Además de esto, los componentes results router y results finisher no tienen handlers para mensajes EOF, ya que el finisher utiliza el campo status y el campo number de los batches para saber cuándo terminaron de llegar todos los batches de cierta tabla para cierta query.

## Pipeline de mensaje CleanUp

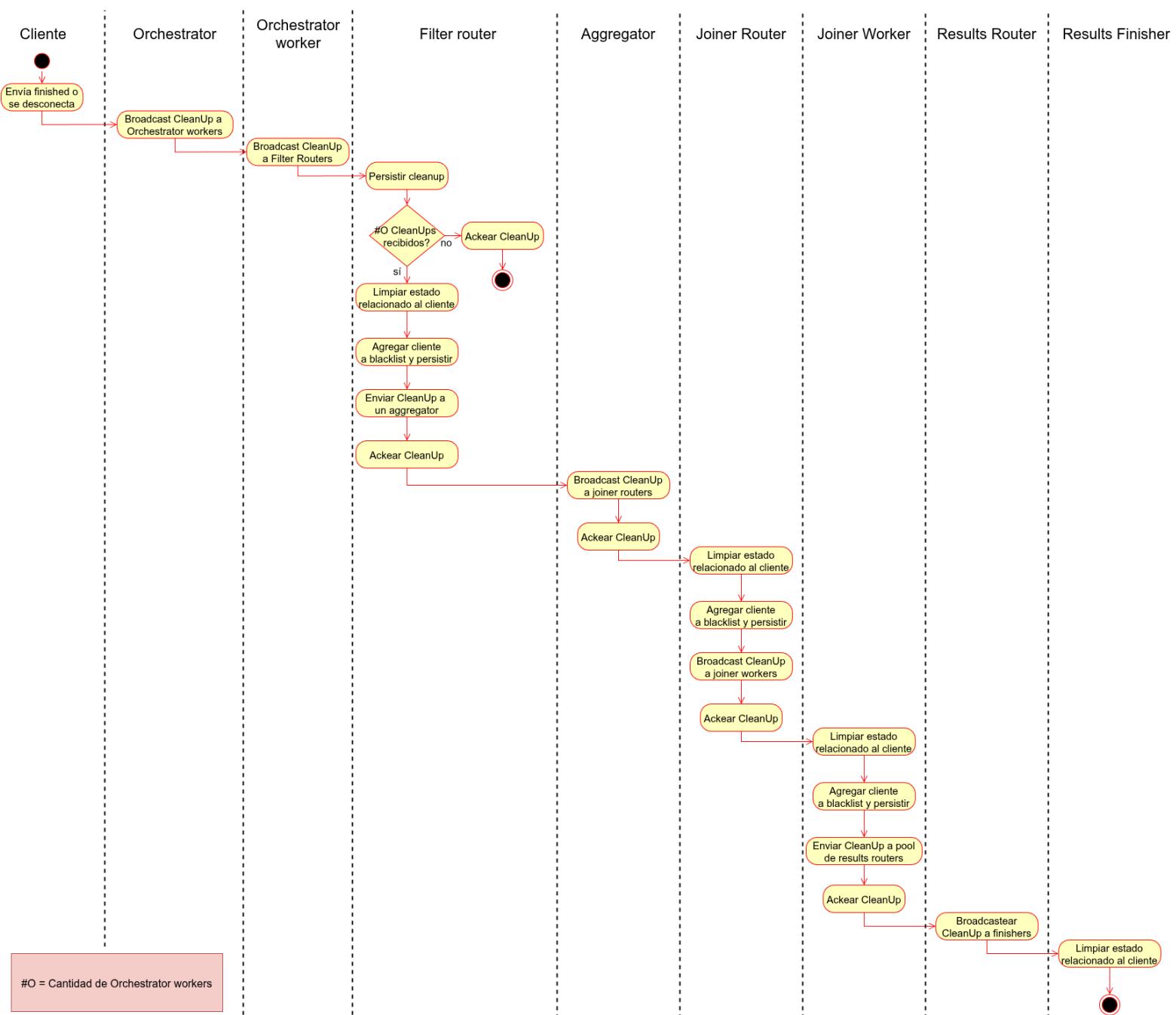
Para entender qué es la blacklist, necesitamos conocer el mensaje CleanUp y comprender su flujo a lo largo del sistema. Este mensaje está pensado para garantizar la liberación de recursos y la consistencia del estado en el sistema distribuido. Anteriormente, la desconexión inesperada de un cliente podía dejar residuos en memoria en los distintos nodos; este mensaje asegura un cierre ordenado mediante la propagación de una señal de "limpieza" a través del pipeline. Además, como veremos más adelante, la gestión de los mensajes EOF, que antes se utilizaban para limpiar el estado de los clientes al final de la run, se simplificó mucho, por lo que ahora este mensaje es utilizado también para borrar los datos relacionados al cliente al final del procesamiento.

El flujo de este proceso se estructura de la siguiente manera:

1. **Detección e Inyección (Orchestrator):** el proceso inicia en el Orquestador cuando se detecta el fin de la interacción con un cliente, ya sea por una finalización exitosa (envío de las 4 queries esperadas) o por una desconexión abrupta del socket. En ese momento, se inyecta una tarea de limpieza (`__cleanup__`) en las colas de todos los workers del orquestador para asegurar que ningún hilo quede pendiente.
2. **Sincronización y Barrera (Filter):** los workers procesan la tarea y notifican a la siguiente etapa (Filter-Router). Aquí se implementa una barrera de sincronización: el router espera recibir la confirmación de limpieza de todos los workers del orquestador antes de proceder. Una vez sincronizado, añade al cliente a una blacklist (para ignorar tráfico residual), limpia su estado local y propaga el mensaje hacia la capa de agregación.
3. **Propagación en Cascada (Aggregator & Joiner):**
  - **Agregación:** El Aggregator actúa como pasarela, rebotando inmediatamente el mensaje hacia la capa de Joiners.
  - **Ruteo de Joins:** El Joiner-Router, al recibir la primera señal de limpieza, bloquea al cliente (`blacklist`) y realiza un *broadcast* a todos los shards de los Joiner-Workers para todas las tablas involucradas, asegurando que ninguna partición retenga datos innecesarios.
4. **Consolidación Final (Results Layer):**

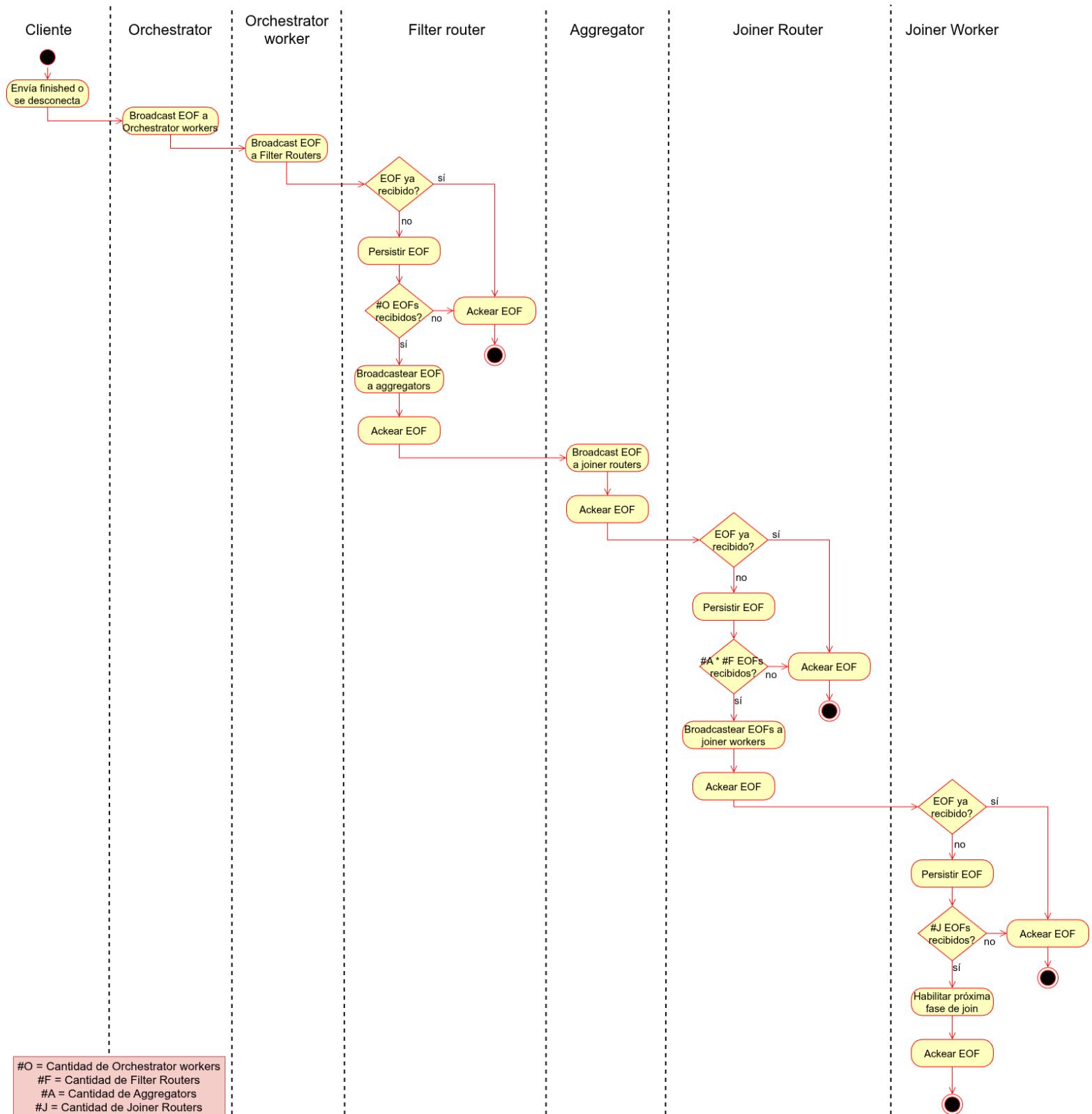
Finalmente, los Joiner-Workers limpian su memoria local. Para evitar una tormenta de mensajes hacia la etapa final, solo el shard 0 de cada worker es responsable de reenviar la señal al Results-Router, el cual se encarga de difundir la orden de cierre a los finalizadores (Results-Finisher), concluyendo así el ciclo de vida de la sesión.

Veámoslo gráficamente:



Podemos observar, gracias al gráfico, que la idea del mensaje CleanUp es limpiar todo el estado relacionado a un cliente cuando se produce su desconexión, por la razón que sea. Para evitar posibles race conditions con mensajes que lleguen luego del CleanUp, se agregan a los clientes cuyo estado se limpió a la blacklist, y siempre se descartan los mensajes de clientes que están en la blacklist. Los client\_id son eliminados de la lista luego de 10 minutos.

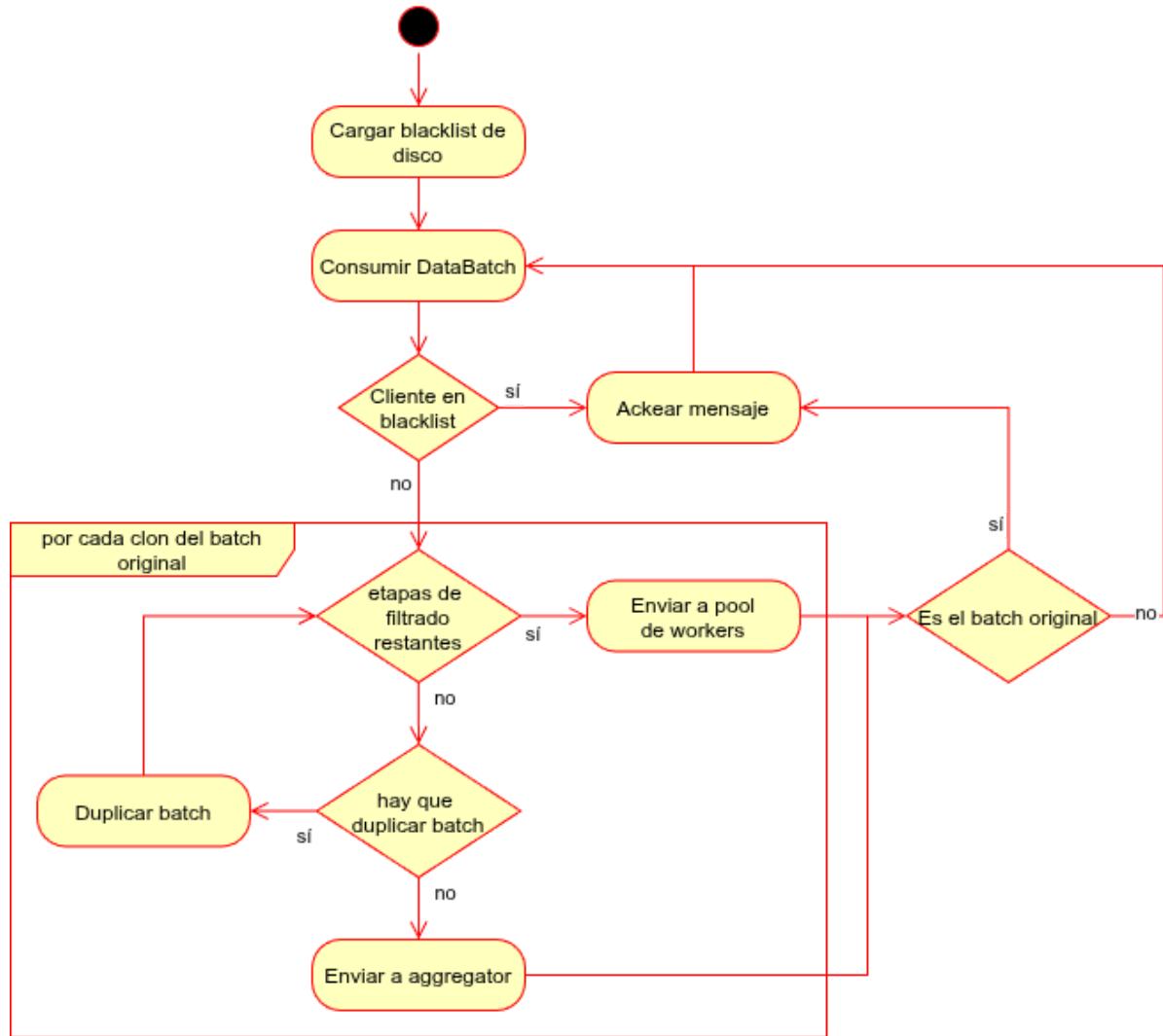
# Pipeline de mensaje EOF



Con el pipeline del CleanUp y el EOF podemos empezar a observar un patrón que se repite a lo largo del procesamiento de todos los mensajes: el ack se realiza recién cuando el mensaje se terminó de procesar por completo. Esto es indispensable para evitar pérdidas

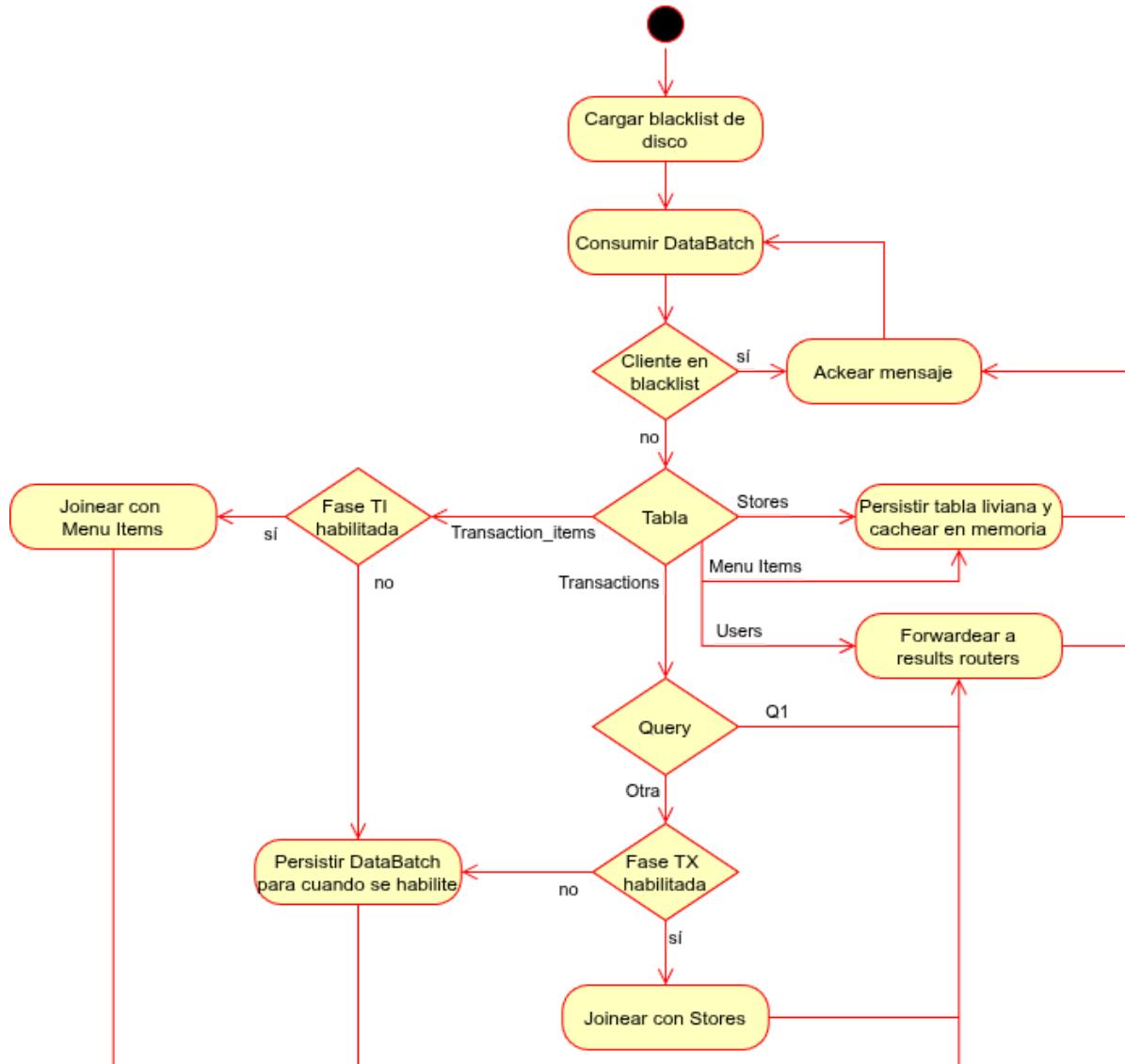
de mensajes ante la caída de un componente: ante la falta de un ack, el middleware reentrega el mensaje, evitando la pérdida del mismo.

## Flujo del filter router



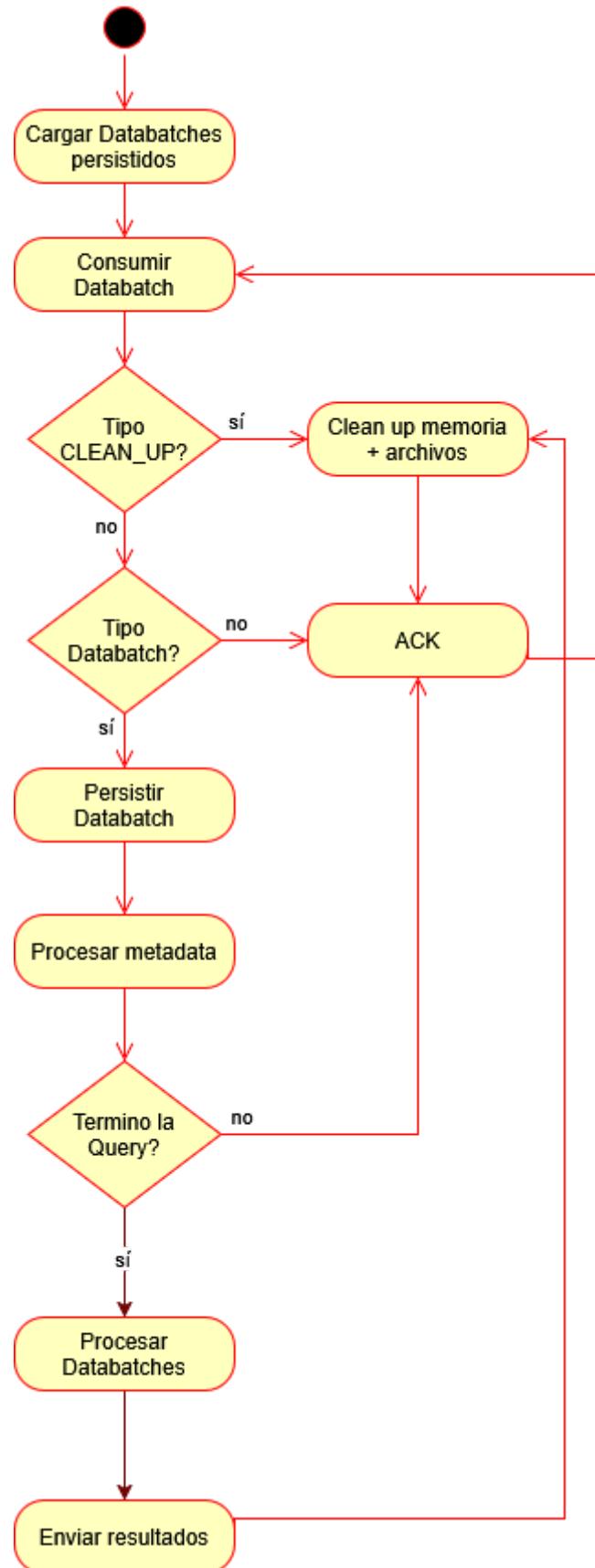
Básicamente el filter router primero se fija si quedan pasos de filtrado para esa combinación de query\_ids y tabla; si quedan pasos, envía el batch al pool de filter workers; si no quedan pasos, se fija si el batch debe duplicarse (ver DAG para entender la lógica detrás de esto), y por cada clon repite el proceso. Si el batch (o el clon) no debe filtrarse ni duplicarse, se envía a algún aggregator. Una vez que todos los clones del batch fueron flusheados, se ackea el batch original.

## Flujo del Joiner Worker



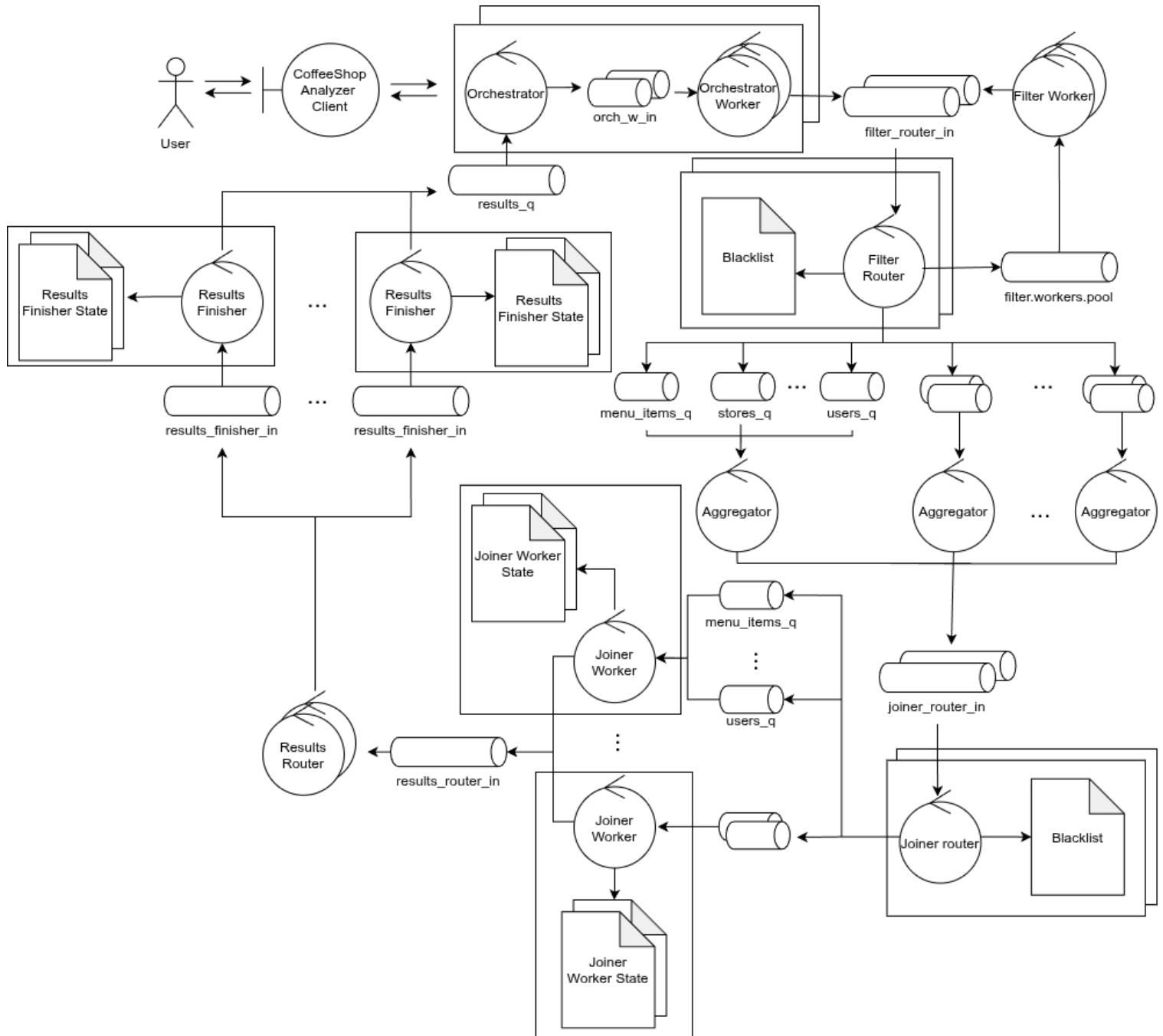
Como se pudo observar en el pipeline del EOF, las fases TX y TI se habilitan cuando llegan todos los EOFs de la tabla liviana correspondiente (menu items para transaction items, y stores para transactions).

## Flujo del Results Finisher

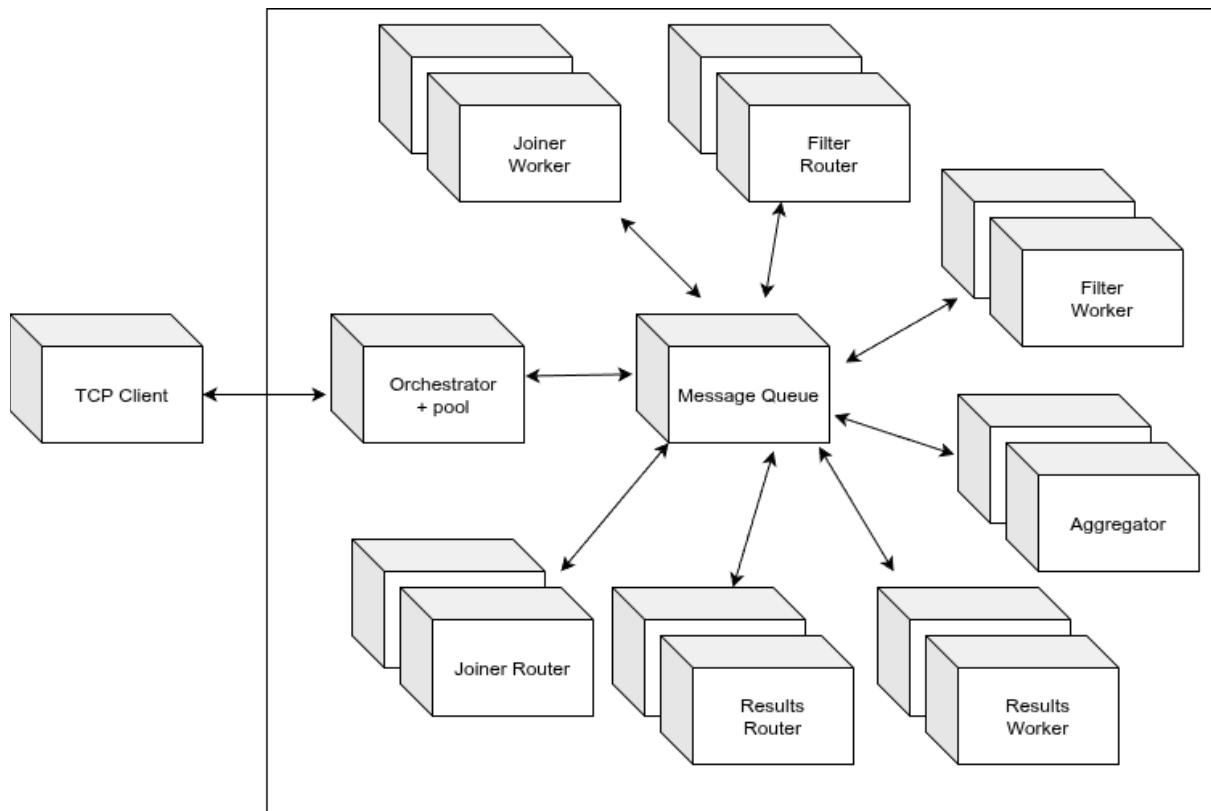


# Vista Física

## Diagrama de robustez



## Diagrama de despliegue



Todos los nodos se corren en containers separados, y todos ellos tienen réplicas. El único container que no está replicado es el del orquestador. Sin embargo, dentro del procesamiento del orquestador, tenemos el filtrado de columnas y armado de batches con metadata. Ese proceso es llevado a cabo por un pool de workers. Si bien no se replica el container del orquestador, sí se replican los workers del pool, mejorando la performance general del sistema al hacer que el envío de archivos sea más rápido.

# Cambios principales

Esta sección detalla los cambios críticos introducidos para garantizar la resiliencia del sistema frente a la versión anterior. El análisis se centra en dos ejes: la extensión del protocolo de mensajes y la nueva lógica de control de nodos, la cual integra roles jerárquicos (Líder) y monitoreo de salud (Heartbeat).

## Evolución y Estructura de Mensajes (Protocolo V2)

La migración a **Protocol Buffers** implicó una reingeniería de los mensajes del sistema. Se abandonaron las estructuras ad-hoc y débilmente tipadas en favor de un esquema estricto que facilita la extensibilidad y la validación. A continuación, se detallan las principales categorías de mensajes y sus modificaciones:

### Envoltorio Universal (Envelope)

Se introdujo un contenedor raíz inexistente en la versión anterior. Antes, el sistema debía inferir el tipo de mensaje o leer bytes crudos; ahora, todo tráfico de red viaja encapsulado en un Envelope.

- **Función:** Provee una capa de abstracción única para el manejo de sockets.
- **Campos:** Utiliza un campo oneof (polimorfismo de Protobuf) que puede contener un DataBatch, EOF, TableData, QueryResult o mensajes de coordinación. Incluye un MessageType explícito para un ruteo rápido sin necesidad de deserializar todo el payload.

### Estandarización de Datos (DataBatch y TableData)

La transmisión de lotes de datos sufrió el cambio estructural más significativo, separando la lógica de enrutamiento de la estructura de los datos.

**DataBatch (Ruteo):** Se eliminaron campos redundantes o arbitrarios (table\_ids, metadatos libres). Ahora se enfoca estrictamente en el transporte: conserva client\_id y query\_ids (ahora tipados como **Enums**, no enteros) e incorpora filter\_steps para orquestar el paso por los filtros de manera dinámica.

**TableData (Payload):** Se reemplazaron las clases específicas por tabla (NewMenuItem, NewStore, etc.) por una estructura genérica TableData.

- **Cambio clave:** En lugar de clases rígidas, ahora se envía un **Esquema (TableSchema)** junto con filas genéricas (Row). Esto permite agregar nuevas tablas o columnas sin modificar el código del protocolo, solo ajustando la configuración.
- **Estado:** Los estados del lote (CONTINUE/EOF/CANCEL) se estandarizaron en el enum TableStatus.

## Unificación de Resultados (QueryResult)

Se eliminó la proliferación de clases para resultados (QueryResult1, QueryResult2, QueryResultError).

- **Diseño Genérico:** Se implementó un único mensaje QueryResult capaz de transportar la respuesta de cualquier consulta. Utiliza, al igual que los datos de entrada, un esquema y filas genéricas.
- **Manejo de Errores:** Se eliminaron los códigos de error explícitos en favor de un diseño más limpio; si la query falla, el sistema lo maneja a nivel de infraestructura o timeout, simplificando el contrato de respuesta.

## Control de Flujo y Ciclo de Vida (EOF y CleanUp)

Los mensajes de control se enriquecieron para soportar mejor la observabilidad y el cierre ordenado de recursos.

- **EOFMessage:** Se hizo estrictamente tipado. El tipo de tabla ya no es un *string* propenso a errores, sino un **Enum TableName**. Se eliminó información redundante (como batch\_number) y se añadió un campo trace para permitir el seguimiento del mensaje a través de los nodos distribuidos.
- **CleanUpMessage (Nuevo):** Se creó este mensaje específico para gestionar la desconexión de clientes y la liberación de memoria (blacklists), cubriendo una carencia del protocolo anterior. Incluye client\_id y trace de propagación.

## Coordinación y Tolerancia a Fallos (Nuevos Mensajes)

Para soportar la nueva arquitectura tolerante a fallos, se crearon mensajes de infraestructura que no existían en la versión manual:

- **Algoritmo de Elección (Bully):** Se incorporaron Election, ElectionAnswer y Coordinator para gestionar dinámicamente la elección de líderes en el clúster.
- **Detección de Fallos (Heartbeating):** Se añadieron Heartbeat (con timestamp de envío) y HeartbeatAck (con timestamp de recepción) para monitorear la salud de los nodos y calcular latencias de red.

## Mensaje TableMessage

Es el único mensaje del protocolo viejo que sigue utilizándose en el envío del cliente al sistema y en la vuelta del resultado del orquestador al cliente.

### Tipos principales de TableMessage

- **NewMenuItems:** lote de nuevos productos del menú.
  - **Claves requeridas:** product\_id, name, category, price, is\_seasonal, available\_from, available\_to.

- **NewStores:** lote de nuevas tiendas.
  - **Claves requeridas:** store\_id, store\_name, street, postal\_code, city, state, latitude, longitude.
- **NewTransactionItems:** lote de ítems de transacciones.
  - **Claves requeridas:** transaction\_id, item\_id, quantity, unit\_price, subtotal, created\_at.
- **NewTransactions:** lote de transacciones.
  - **Claves requeridas:** transaction\_id, store\_id, payment\_method\_id, user\_id, original\_amount, discount\_applied, final\_amount, created\_at.
- **NewUsers:** lote de usuarios.
  - **Claves requeridas:** user\_id, gender, birthdate, registered\_at
- **EOFMessage:** mensaje de fin de tabla, indica que se terminó de enviar todos los datos de un tipo específico.
  - **Claves requeridas:** table\_type (string que especifica la tabla).

Estructura general de TableMessage

[opcode:1][length:4][nRows:4][batchNumber:8][status:1][rows...]

Características especiales

- El mensaje EOF (EOFMessage) tiene siempre una sola fila virtual con metadatos y estado BatchStatus.EOF.
- Los TableMessage de datos contienen una lista de objetos (rows) según el tipo de tabla.

## Alta Disponibilidad y Elección de Líder

Para garantizar la resiliencia del sistema distribuido, se implementó un mecanismo de **Alta Disponibilidad** basado en la elección dinámica de líderes. Cada grupo de servicios (Orquestadores, Routers, Workers y Agregadores) forma un clúster autónomo donde los nodos colaboran para detectar fallos y autogestionar su recuperación.

La arquitectura se sostiene sobre tres pilares: el Algoritmo de Elección (Bully), el monitoreo de salud (Heartbeating) y la recuperación automática (Self-Healing).

El Rol del Líder (Watchdog Pattern)

En cada grupo de réplicas, se elige un único nodo como **Líder**. Si bien todos los nodos procesan datos en paralelo (workers), el Líder asume una responsabilidad administrativa adicional: actúa como un *Watchdog* (perro guardián).

- **Responsabilidad:** es el único autorizado para vigilar la salud de sus pares (Followers) y ejecutar comandos de recuperación (como reiniciar contenedores Docker) si detecta que uno ha caído.
- **Beneficio:** Evita condiciones de carrera donde múltiples nodos intentarían reiniciar al mismo servicio caído simultáneamente.

### Algoritmo de Elección: Bully Algorithm

Se implementó el **Algoritmo de Bully** para determinar qué nodo asume el liderazgo. Este algoritmo favorece al nodo con el identificador (ID) más alto.

- **Dinámica de Estados:** un nodo puede estar en estado follower, **candidate** o **leader**.
- **Proceso de Elección:**
  1. Si un nodo detecta la ausencia del líder actual, se autoproclama **candidate** y dispara `start_election()`.
  2. Envía mensajes de **election** a todos los nodos con ID mayor al suyo.
  3. Si nadie con mayor jerarquía responde, el nodo se autoproclama **leader** y envía un mensaje **coordinator** al resto para informar su nuevo rol.
  4. Si recibe un mensaje de un nodo superior, el candidato aborta y vuelve a ser **follower**.
- **Renuncia Ordenada (Graceful Resignation):** si un líder debe apagarse (ej. SIGTERM por despliegue), invoca `graceful_resign()`, forzando una elección inmediata antes de salir para minimizar el tiempo de inactividad.

### Detección de Fallos mediante Heartbeats

La salud del sistema se monitorea mediante un intercambio constante de latidos (*heartbeats*) sobre UDP/TCP, configurado dinámicamente según el rol del nodo.

- **Flujo (Follower → Leader):** los Followers envían activamente un mensaje **heartbeat** al Líder cada **heartbeat\_interval\_seconds**.
- **Confirmación:** el Follower espera un ACK. Si no lo recibe tras un tiempo (TIMEOUT), incrementa un contador de fallos.
- **Detección de Líder Caído:** si un Follower acumula **heartbeat\_max\_misses** sin respuesta, asume que el líder ha muerto e inicia una elección.

- *Nota de Diseño:* Se incluye un jitter y un tiempo de cooldown para evitar que todos los followers inicien elecciones simultáneamente, lo que saturaría la red.

## Recuperación Automática (Self-Healing)

Esta es la capa de acción correctiva. El módulo **FollowerRecoveryManager** se activa exclusivamente en el nodo Líder.

- **Monitoreo Activo:** el Líder revisa periódicamente cuándo fue el último latido recibido de cada Follower.
- **Criterio de Muerte:** si un nodo no ha reportado salud por más de **follower\_down\_timeout\_seconds** (y ya pasó su periodo de gracia de inicio), se considera caído.
- **Resurrección:** el Líder se comunica con el daemon de Docker y ejecuta un docker restart sobre el contenedor afectado.
- **Protección:** para evitar bucles infinitos de reinicio en contenedores defectuosos, se limita la cantidad de intentos (**max\_restart\_attempts**) y se imponen tiempos de espera entre reinicios.

## Configuración Parametrizable

Todo el sistema de tolerancia a fallos es altamente configurable mediante variables de entorno inyectadas al inicio, permitiendo ajustar la sensibilidad del sistema sin recomilar:

- **Tiempos:** intervalos de latido, timeouts de elección y periodos de gracia al inicio (**startup\_grace\_seconds**) para permitir que los procesos pesados arranquen antes de ser monitoreados.
- **Puertos:** definición de puertos específicos de elección para aislar el tráfico de control del tráfico de datos.

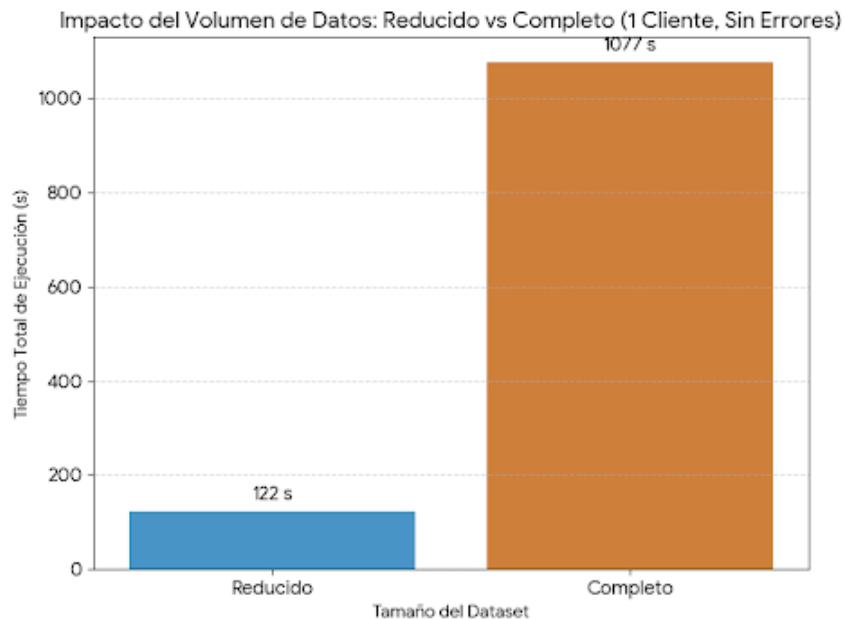
# Métricas

Las siguientes secciones detallan el análisis de diversos ejes del sistema. Las pruebas fueron realizadas en una computadora de las siguientes características:

## Especificaciones del dispositivo

- **Procesador:** Intel(R) Core(TM) i5-10400 CPU @ 2.90GHz (6 núcleos, 12 hilos lógicos)
- **RAM instalada:** 16,0 GB (2 módulos de 8 GB DDR4 a 2667 MHz)
- **Almacenamiento:**
  - Disco principal (Sistema): 480 GB (WD Green SSD)
- **Tipo de sistema:** Sistema operativo de 64 bits, procesador x64
- **Gráficos:** Intel UHD Graphics 630 (Integrada)
- **Placa Base:** MSI B560M PRO-E (MS-7D22)

## Análisis de Escalabilidad por Volumen de Datos (Dataset Reducido vs. Completo)



Para evaluar la estabilidad del sistema bajo condiciones de carga intensiva, se comparó el tiempo de procesamiento del dataset reducido frente al dataset completo, utilizando el protocolo nuevo y un único cliente.

## Dimensionamiento de los Datasets

A partir de los archivos fuente, se calcularon los volúmenes totales a procesar:

- **Dataset Reducido:** ~357 MB (Principalmente transaction\_items 166MB y transactions 97MB).
- **Dataset Completo:** ~3.2 GB (Crecimiento masivo en transaction\_items a 1.9GB y transactions a 1.2GB).
- **Observación:** La tabla de dimensiones users se mantuvo constante (94MB), aislando la prueba al crecimiento de las tablas transaccionales (Hechos).

## Resultados Cuantitativos

Versión del Dataset	Tiempo Total (s)	Minutos (aprox)	Factor de Incremento
Reducido	122 s	~2 min	Base (1x)
Completo	1077 s	~18 min	8.83x

## Eficiencia Sostenida

Los resultados evidencian una **escalabilidad lineal casi perfecta**. Mientras que el volumen de datos aumentó **9.1 veces**, el tiempo de procesamiento solo aumentó **8.8 veces**.

## Amortización de Overhead

El hecho de que el factor de tiempo (8.8x) sea menor al factor de datos (9.1x) indica que los costos fijos de inicialización del sistema (conexiones TCP, elecciones de líder, *handshakes* de Protobuf) se diluyen eficazmente en cargas de trabajo largas.

## Estabilidad en Joins

Entendido. Ajustamos el texto para reflejar la realidad: el sistema demuestra estabilidad en cargas medias (Dataset Reducido), pero **satura y colapsa** bajo carga masiva (Dataset Completo) debido al cuello de botella de I/O, verificado mediante las métricas de RabbitMQ.

Aquí tienes la versión corregida para el informe, con un tono técnico que justifica la falla por saturación de recursos físicos (disco/red) y no lógica:

## Estabilidad y Límites de Carga (Dataset Reducido vs. Completo)

El análisis comparativo revela dos comportamientos diametralmente opuestos dependiendo del volumen de datos, exponiendo los límites físicos de la infraestructura:

### 1. Dataset Reducido (Escenario Estable):

Con un volumen de 166 MB en tablas transaccionales, la estrategia de particionado (sharding) funcionó de manera efectiva. La gestión de memoria y la persistencia en disco operaron dentro de los márgenes de latencia aceptables, evitando bloqueos y manteniendo el flujo constante de mensajes.

### 2. Dataset Completo (Escenario de Colapso):

Al escalar a 1.9 GB, el sistema de Joiners alcanzó su punto de saturación, provocando inestabilidad operativa. A diferencia del escenario reducido, el volumen masivo de datos superó la capacidad de throughput de disco (Disk I/O) requerida para manejar los Late Joins.

## Puntos de Presión Arquitectónica y Evidencia (RabbitMQ):

El monitoreo de las colas de RabbitMQ permitió identificar la secuencia que llevó a la degradación del servicio:

- **Saturación en Joiners y Persistencia:** la arquitectura obliga a los *Joiner Workers* a persistir en disco los lotes que no pueden unir inmediatamente. Con el dataset completo, la escritura en disco se volvió el factor dominante, ralentizando drásticamente el consumo de mensajes.
- **Result Workers:** de manera similar, la etapa de consolidación (Results Worker) sufrió cuellos de botella al intentar agrupar y persistir resultados parciales de gran volumen
- **Efecto en las Colas:** se observó un crecimiento en las colas de entrada de estos servicios. Al no poder procesar los mensajes a la velocidad de llegada, se generó un efecto de *backpressure* que saturó la memoria del middleware y elevó la latencia de red.

Esto pudo verificarse mediante el monitoreo de colas de RabbitMQ.

Vamos a ver algunas métricas de colas de RabbitMQ con envío del dataset completo de 1 cliente.

Vamos a ver el pico más alto registrado para los finishers:

/	<b>finisher_input_queue_0</b>	classic	D	running	0	0	0	21/s	24/s	24/s
/	<b>finisher_input_queue_1</b>	classic	D	running	0	0	0	21/s	23/s	23/s
/	<b>finisher_input_queue_2</b>	classic	D	running	0	5,250	5,250	12/s	0.80/s	23/s
/	<b>finisher_input_queue_3</b>	classic	D	running	0	0	0	21/s	24/s	23/s

Por ejemplo, acá vemos la finisher\_input\_queue\_2 con 5250 mensajes acumulados de procesamiento de la query2. También vemos que las velocidades se degradan, ya que momentos antes puede visualizarse lo siguiente.

/	<b>finisher_input_queue_0</b>	classic	D	running	0	0	0	0.00/s	0.00/s	0.00/s
/	<b>finisher_input_queue_1</b>	classic	D	running	0	0	0	0.00/s	0.00/s	0.00/s
/	<b>finisher_input_queue_2</b>	classic	D	running	0	4,273	4,273	52/s	51/s	19/s
/	<b>finisher_input_queue_3</b>	classic	D	running	0	0	0	0.00/s	0.00/s	0.00/s

Vemos que incluso con carga alta, las velocidades eran mayores.

En cuanto al joiner, observamos que las colas de transactions suelen estar en valores altos durante toda la ejecución del programa, ya que transactions es la segunda tabla que el cliente envía y se utiliza en todas las queries –excepto la query 2–.

Los picos más altos registrados son:

/	<b>join.transactions.shard.00</b>	classic	D	running	3,430	100	3,530	20/s	7.4/s	7.4/s
/	<b>join.transactions.shard.01</b>	classic	D	running	3,409	100	3,509	19/s	5.0/s	5.0/s
/	<b>join.transactions.shard.02</b>	classic	D	running	3,347	100	3,447	21/s	4.6/s	4.6/s
/	<b>join.transactions.shard.03</b>	classic	D	running	3,462	100	3,562	19/s	7.8/s	7.8/s

Unos segundos después, las velocidades empiezan a degradarse, tal como sucede con el results finisher.

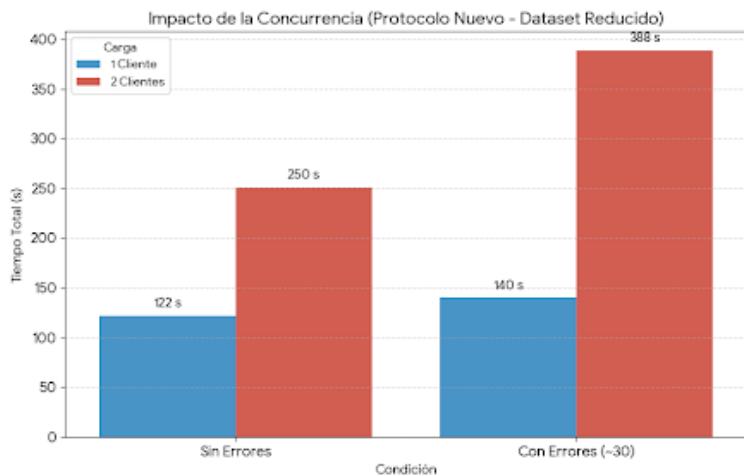
/	<b>join.transactions.shard.00</b>	classic	D	running	3,418	100	3,518	0.00/s	2.6/s	2.6/s
/	<b>join.transactions.shard.01</b>	classic	D	running	3,397	100	3,497	0.00/s	2.4/s	2.4/s
/	<b>join.transactions.shard.02</b>	classic	D	running	3,400	100	3,500	0.00/s	4.2/s	4.2/s
/	<b>join.transactions.shard.03</b>	classic	D	running	3,433	100	3,533	0.00/s	2.6/s	2.6/s

Esto marca una gran diferencia con componentes como el agregador, donde los picos más altos registrados son los siguientes:

/	<b>agg.transactions.00</b>	classic	D	running	0	73	73	6.8/s	7.0/s	6.4/s
/	<b>agg.transactions.01</b>	classic	D	running	0	36	36	7.6/s	7.0/s	5.0/s
/	<b>agg.transactions.02</b>	classic	D	running	0	90	90	5.6/s	5.4/s	6.2/s
/	<b>agg.transactions.03</b>	classic	D	running	0	61	61	5.2/s	4.2/s	5.6/s

Lo cual no representa una diferencia con la versión reducida.

## Single-Client vs. Multi-Client



Se evaluó el comportamiento del sistema al duplicar la carga de trabajo simultánea (de 1 a 2 clientes), analizando cómo la concurrencia afecta los tiempos de respuesta bajo distintas condiciones de red.

### Escenario Ideal (Sin Errores): Escalabilidad Lineal

En condiciones normales de red, el sistema exhibe un comportamiento predecible y justo.

- **1 Cliente:** 122 s.
- **2 Clientes (Promedio):** ~250 s.
- **Factor de Crecimiento:** **2.05x**.
- **Interpretación:** El tiempo prácticamente se duplica, lo que indica una **asignación justa de recursos (Fair Scheduling)**. Los workers procesan las tareas de ambos clientes de manera alternada sin overhead significativo. El sistema está "saturado" pero es eficiente; no pierde tiempo en cambios de contexto excesivos.

### Escenario Crítico (Con Inyección de Fallos): Degradación Super-lineal

Al introducir una tasa alta de errores (20-30 errores por sesión), la concurrencia penaliza el rendimiento más allá de la simple suma de cargas.

- **1 Cliente (30 errores):** 140 s.
- **2 Clientes (20/30 errores):** ~388 s.
- **Factor de Crecimiento:** **2.77x**.

- **Interpretación:** El incremento supera el factor lineal de 2x. Este **0.77x adicional** representa el costo de la **contención de recursos**.
  - Al haber fallos, se disparan retransmisiones y duplicados.
  - Esto incrementa el tráfico en RabbitMQ y la competencia por el disco (I/O) en los nodos de persistencia (Joiners/Results).
  - El sistema pasa de un modelo "cooperativo" a uno "competitivo", donde las operaciones de recuperación de un cliente bloquean momentáneamente el flujo del otro.

## Conclusión sobre la Conurrencia

El Protocolo Nuevo maneja la concurrencia de manera robusta. En ausencia de fallos, garantiza que el rendimiento no se degrade (el throughput total del sistema se mantiene constante, solo se reparte entre dos). Sin embargo, bajo condiciones de alta tasa de error, la latencia aumenta de forma no lineal debido a la saturación de I/O provocada por la gestión simultánea de flujos de datos y lógica de recuperación.

## Robustez y Comportamiento bajo Estrés

Si bien el procesamiento del Dataset Completo con un único cliente se mantuvo estable (validando la implementación de Protobuf y RabbitMQ para sesiones largas), la introducción de **conurrencia (múltiples clientes simultáneos)** reveló los verdaderos límites de estrés de la infraestructura, afectando la configuración de *liveness*.

1. **Contención de Recursos (Resource Contention):** a diferencia del flujo secuencial de un solo cliente, la carga multicliente generó una alta competencia por los recursos de E/S en los nodos con estado (Joiner y Results Finisher). Al tener múltiples hilos intentando persistir lotes en disco simultáneamente, la latencia de escritura aumentó drásticamente, llevando a los CPUs al límite por el costo de *context switching* y espera de I/O.
2. **Inanición de Procesos y Falsas Elecciones:** esta saturación por concurrencia provocó momentos de **inanición de procesos (process starvation)**. Los hilos encargados del protocolo de *Heartbeat* (cuya prioridad debería ser alta) quedaron encolados esperando ciclos de CPU o desbloqueo de disco. El sistema interpretó este silencio no como una saturación, sino como la caída del nodo, disparando mecanismos de "**Falsas Elecciones**" de líder.
3. **Resiliencia ante la Inestabilidad:** se observó que, incluso habiendo duplicado preventivamente los *timeouts* de elección, la latencia inducida por

la carga concurrente superó estos márgenes en momentos críticos. Sin embargo, el sistema demostró una notable **capacidad de auto-recuperación**: tras dispararse las elecciones espurias, el clúster fue capaz de reorganizar el liderazgo, estabilizarse una vez liberada la presión puntual y continuar el procesamiento hasta finalizar la tarea sin pérdida de datos.

## Lecciones aprendidas

### Simplificación de Filter Router

Anteriormente, el cliente enviaba mensajes EOF para todas las tablas del dataset, permitiendo eliminar el estado del cliente en los componentes inmediatamente después de la transmisión. Si bien esto parecía ventajoso en teoría, en la práctica generaba una sobrecarga de mensajes y aumentaba innecesariamente la complejidad del Filter Router.

Este componente se veía obligado a mantener un seguimiento (en memoria y disco) de los batches pendientes de envío al Aggregator. Esto ocurría porque, debido a la lógica de filtrado, los batches de Transactions y Transaction Items no fluyen linealmente, sino que requieren múltiples pasadas por los filter workers. Esta arquitectura incrementaba el consumo de recursos, ralentizaba el procesamiento y multiplicaba los posibles puntos de falla, comprometiendo la tolerancia a errores.

Al reevaluar el problema, determinamos que los únicos mensajes EOF críticos son los de las tablas livianas (Stores y Menu Items), ya que son indispensables para que el joiner worker habilite la fase de join. La diferencia temporal entre limpiar el estado tras un EOF o esperar al mensaje de CleanUp es marginal; sin embargo, optar por lo segundo reduce drásticamente la cantidad de mensajes y la complejidad del sistema.

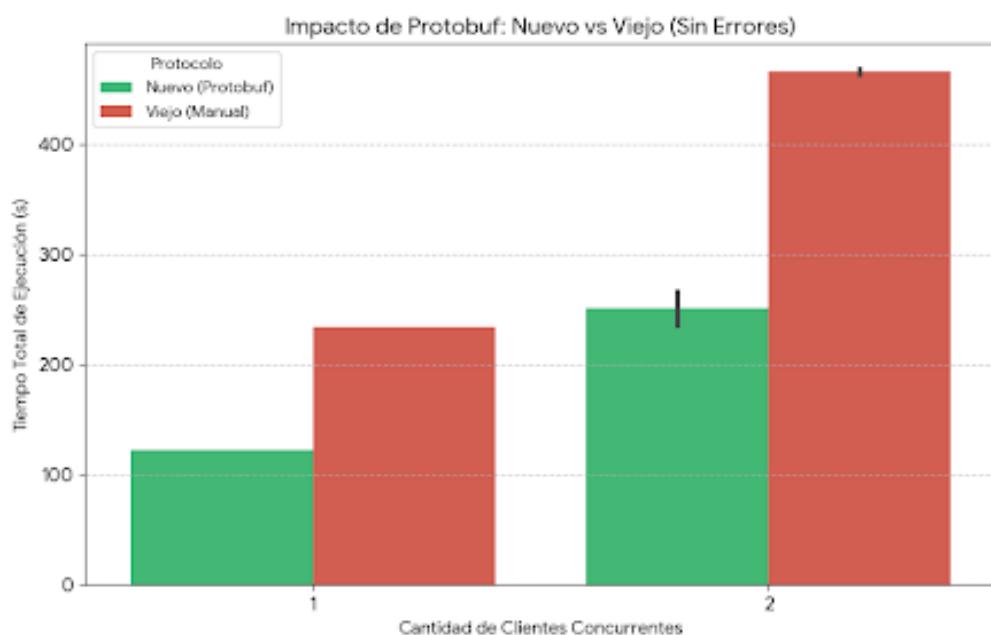
Esto simplifica el Filter Router porque, al ser Transactions y Transaction Items las únicas tablas que circulan por los workers, se garantiza que para el resto de las tablas el EOF llegue a los aggregators una vez finalizado el flujo real, eliminando la necesidad del mecanismo de seguimiento anterior.

A modo de comparación, veamos el diagrama de actividades del filter router bajo el esquema previo (el diagrama actual se puede ver en la vista de procesos):



## Protocolo Interno

El protocolo propietario utilizado inicialmente introducía una alta rigidez en el sistema. La modificación del esquema de datos, tal como la adición de nuevos campos, incrementaba la fragilidad del código y el riesgo de errores en tiempo de ejecución. Con el objetivo de garantizar la integridad de los datos y facilitar la mantenibilidad, se migró la capa de comunicación a **Protobuf**. Esta decisión arquitectónica no solo resolvió los problemas de robustez al establecer un contrato de datos estricto, sino que, como evidencian las métricas, redujo drásticamente los tiempos de latencia gracias a la eficiencia del formato binario frente al modelo anterior (las métricas fueron tomadas con el mismo dispositivo especificado en la sección de Métricas).



## Análisis de Impacto: Migración a Protobuf y Tolerancia a Fallos

Se comparó el desempeño de la versión antigua del sistema (protocolo manual) contra la nueva versión (basada en Protobuf y con tolerancia a fallos). Los resultados demuestran que la optimización en la capa de serialización compensa y supera significativamente el overhead introducido por los mecanismos de confiabilidad.

Tabla de Resultados

Carga (Clientes)	Protocolo Viejo (Manual)	Protocolo Nuevo (Protobuf)	Mejora de Performance
<b>1 Cliente</b>	234 s	<b>122 s</b>	<b>~48% de reducción de tiempo</b>
<b>2 Clientes</b>	466 s	<b>251 s</b>	<b>~46% de reducción de tiempo</b>

## Conclusiones Técnicas

- Eficiencia de Protobuf:** El cambio a Protocol Buffers ha permitido reducir el tiempo de ejecución casi a la mitad. Esto se debe a:
  - Serialización Binaria:** Protobuf genera mensajes mucho más compactos que el protocolo manual anterior, reduciendo drásticamente el uso de ancho de banda en la red
  - Menor Costo de CPU:** El proceso de serializar/deserializar (marshalling/unmarshalling) es computacionalmente más eficiente, liberando recursos en los *Workers* y el *Middleware* para procesar lógica de negocio.
- Costo de la Tolerancia a Fallos:** Generalmente, agregar ACKs, reintentos y persistencia añade latencia. Sin embargo, en este caso, la ganancia de rendimiento obtenida por Protobuf fue tan alta que **absorbió completamente este costo**, resultando en un sistema que es a la vez **más robusto y más rápido**.
- Escalabilidad Sostenida:** La mejora se mantiene constante al duplicar la carga de clientes, lo que indica que el nuevo protocolo escala correctamente y alivia la presión sobre el middleware de colas.

# Conclusiones

El desarrollo e implementación de **Coffee Shop Analyzer** ha permitido validar la eficacia de una arquitectura distribuida basada en el patrón *Pipe and Filters* para el procesamiento de grandes volúmenes de datos, cumpliendo satisfactoriamente con los requisitos de tolerancia a fallos y alta disponibilidad.

A lo largo de este trabajo, se destacan tres pilares fundamentales que definen el éxito y los límites de la solución:

## Impacto Crítico de la Optimización del Protocolo

La transición de un protocolo de texto propietario a Protocol Buffers demostró ser la decisión arquitectónica más influyente del proyecto. Más allá de las mejoras en mantenibilidad y tipado estricto, las métricas revelaron que la eficiencia de la serialización binaria permitió reducir los tiempos de procesamiento en aproximadamente un 48%. Esta ganancia de rendimiento fue decisiva, ya que permitió "amortizar" el costo computacional adicional introducido por los mecanismos de confiabilidad (ACKs, reintentos y persistencia), logrando un sistema que es, paradójicamente, más robusto y más rápido que su versión original no tolerante a fallos.

## Escalabilidad Lineal y Cuellos de Botella Físicos

El sistema exhibió una escalabilidad notable bajo condiciones de carga controlada. Las pruebas de volumen evidenciaron un comportamiento lineal casi perfecto: ante un incremento de 9.1x en el volumen de datos (del dataset reducido al completo), el tiempo de ejecución creció en un factor de 8.8x. Esto confirma que la sobrecarga de gestión del clúster es mínima y que la lógica de negocio escala correctamente.

Sin embargo, las pruebas de estrés (Dataset Completo y concurrencia Multi-Cliente) expusieron los límites físicos de la infraestructura. Se identificó que el sistema es **I/O Bound** en las etapas de *Join* y *Consolidación*. La saturación de escritura en disco en estos nodos provocó fenómenos de *inanición de procesos (starvation)*, donde la latencia de I/O bloqueó momentáneamente los hilos de control (Heartbeats), desencadenando "falsas elecciones" de líder. Este hallazgo subraya que, en sistemas de alto throughput, la optimización lógica debe ir acompañada de una infraestructura de almacenamiento adecuada.

## Resiliencia y Auto-Recuperación

A pesar de los desafíos de saturación y las elecciones espurias bajo estrés extremo, el sistema demostró una alta resiliencia. Los mecanismos de Self-Healing,

orquestados mediante el algoritmo Bully y el patrón Watchdog, lograron estabilizar el clúster automáticamente sin intervención humana. Incluso en los escenarios más críticos, donde la competencia por recursos fue máxima, el sistema fue capaz de reorganizarse, garantizar la consistencia de los datos y finalizar el procesamiento de todas las consultas.

En definitiva, la solución presentada no solo resuelve la problemática de negocio planteada, sino que constituye una plataforma robusta capaz de sobrevivir a fallos de nodos, caídas de red y picos de carga, priorizando siempre la integridad del resultado final sobre la velocidad pura, tal como lo exige un sistema de misión crítica.