FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico
Coffee Shop Analysis

[TA050] Sistemas Distribuidos
Segundo Cuatrimestre de 2025

Alumno	Padrón
FÁBREGAS, Camilo	103740
FERNÁNDEZ, Julio Mateo	107491
HUTTIN, Facundo Tomás	107854

Índice

1. Introducción

El presente documento de arquitectura describe la solución propuesta para el problema *Coffee Shop Analysis*, cuyo propósito consiste en diseñar un sistema distribuido capaz de procesar un conjunto de consultas sobre datos de ventas pertenecientes a una cadena de cafeterías.

El sistema opera sobre datos correspondientes al período comprendido entre julio de 2023 y junio de 2025, que incluye información de transacciones, clientes, sucursales y productos. Dichos datos constituyen la base para los escenarios de uso que orientan y justifican las decisiones de diseño.

La arquitectura propuesta se fundamenta en el patrón *Pipelines and Filters*, en el cual un flujo continuo de datos provenientes del cliente es procesado de manera incremental por distintos nodos especializados. Cada nodo corresponde a un *worker* que aplica un filtro o transformación específicos, alineados con las consultas definidas en el dominio del problema.

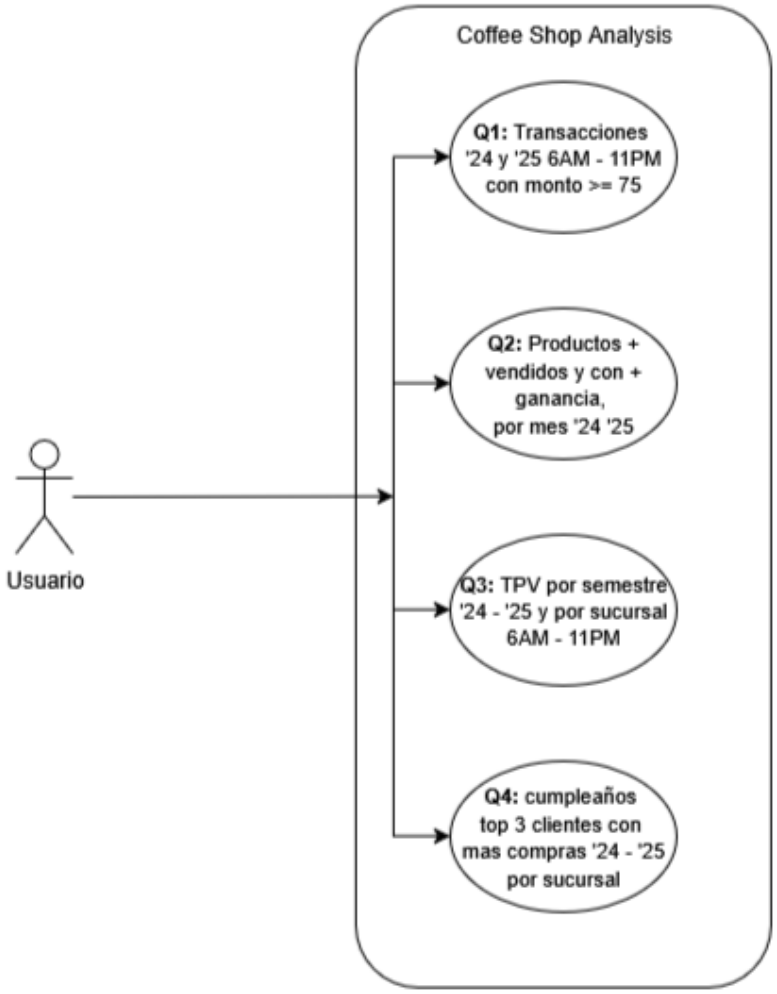
El documento organiza la descripción de la solución siguiendo el Modelo de Vistas 4+1, abarcando las vistas lógica, de procesos, de desarrollo y física, así como los escenarios y casos de uso que permiten validar la coherencia global de la arquitectura.

2. Vista de Escenarios

El sistema contempla cuatro casos de uso principales, cada uno asociado a una de las *queries* definidas para el proyecto. Estas consultas serán resueltas utilizando el conjunto de datos provisto por el cliente. A continuación se presentan los cuatro casos de uso considerados.

1. Transacciones (id y monto) realizadas durante 2024 y 2025 entre las 06:00 AM y las 11:00 PM con monto total mayor o igual a 75.
2. Productos más vendidos (nombre y cantidad) y productos que más ganancias han generado (nombre y monto), para cada mes en 2024 y 2025.
3. TPV (*Total Payment Value*) por cada semestre en 2024 y 2025, para cada sucursal, para transacciones realizadas entre las 06:00 AM y las 11:00 PM.
4. Fecha de cumpleaños de los 3 clientes que han hecho más compras durante 2024 y 2025, para cada sucursal.

2.1. Casos de Uso del Sistema



3. Vista Lógica

3.1. Componentes del sistema

A continuación se enumeran los componentes *protagonistas* del sistema junto con una breve explicación de sus responsabilidades, con el objetivo de establecer la estructura conceptual sobre la cual se organiza la arquitectura.

- **Actores**
  - **Cliente**: Actor externo que interactúa con el sistema enviando las consultas y los archivos *.csv* correspondientes a transacciones, productos, sucursales y usuarios. También recibe los resultados generados por el procesamiento distribuido.
- **Objetos de Borde (*Boundary Objects*)**
  - **Server Gateway**: Punto de entrada del sistema encargado de recibir las consultas y los archivos enviados por el cliente, y de devolver los resultados generados.
- **Entidades de Filtrado**
  - **Cleaner**: Encargado del filtrado inicial de las columnas relevantes para cada consulta, permitiendo que solo la información necesaria avance hacia las etapas posteriores.
  - **Filtrador por Tiempo**: Filtra los registros en función de un rango temporal definido, reenviando únicamente los registros válidos para la consulta correspondiente.
  - **Filtrador por Monto**: Filtra los registros de transacciones según un umbral de monto específico.
- **Entidades Lógicas**
  - **Agrupadores y Reductor**: Los agrupadores generan agregaciones parciales y, una vez procesados todos los registros, el Reductor produce la agregación final.
  - **Ordenadores**: Ejecutan los ordenamientos requeridos por cada consulta antes del envío del resultado.
  - ***Toppers***: Obtienen el *top N* de un conjunto de registros recibido.
  - ***Joiners***: Recolectan y combinan registros provenientes de diferentes conjuntos de datos según el tipo de consulta. Cada *joiner* constituye el último eslabón de la cadena de procesamiento y envía los resultados al *Server Gateway*.
- **Entidades de Control**
  - ***Worker State Manager (WSM)***: Responsable de coordinar el procesamiento distribuido, manteniendo la sincronización entre réplicas mediante la asignación de posiciones y la preservación del estado.
  - **Coordinador**: Secuencia los mensajes asignándoles una posición global, permitiendo a los nodos conocer el progreso y la cantidad restante de información a procesar.

3.2. Arquitectura multi-cliente y Gateway

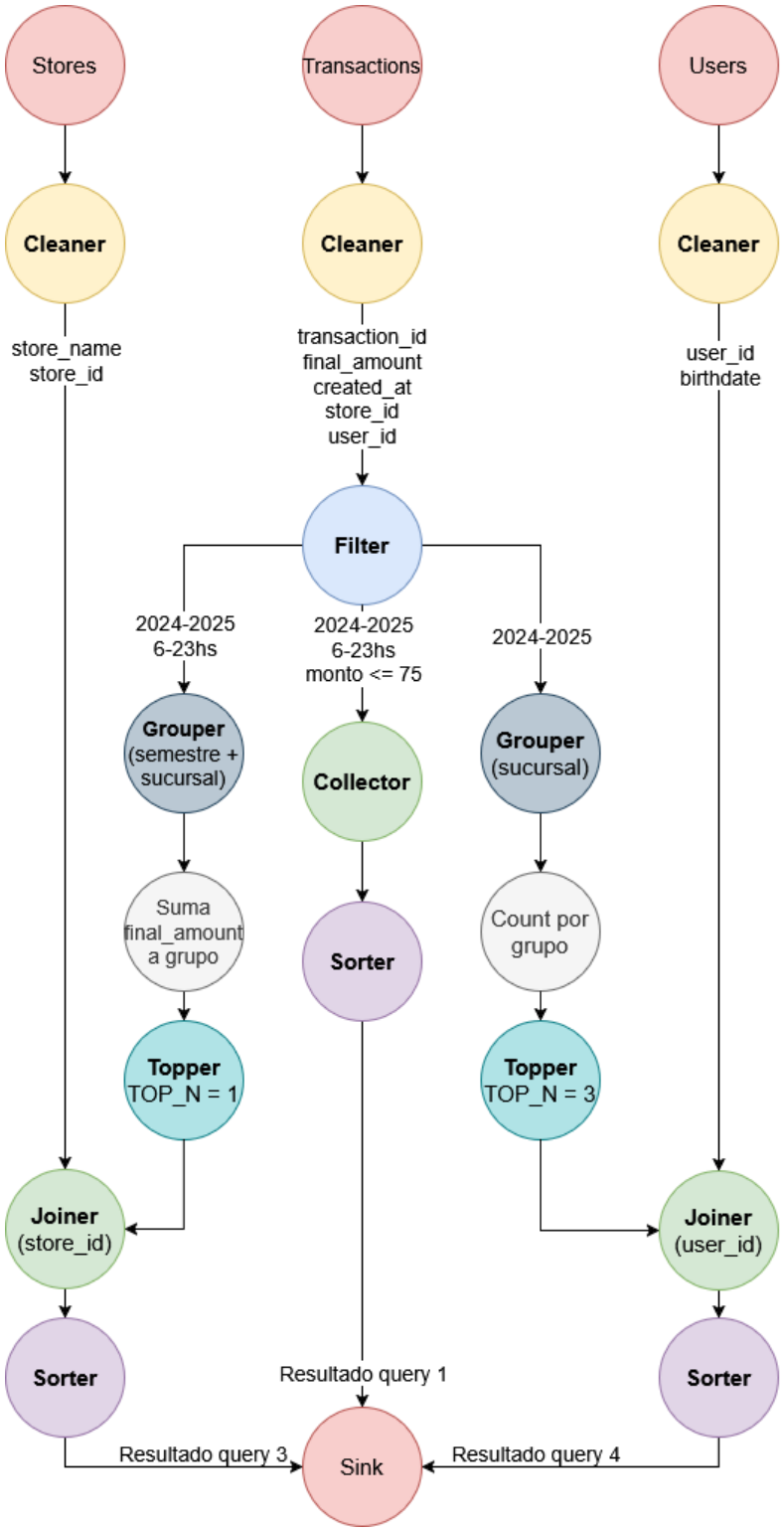
El sistema adopta una arquitectura orientada a múltiples clientes concurrentes mediante un **Gateway** que actúa como punto de entrada único. Este componente gestiona de manera aislada cada conexión entrante, permitiendo escalar horizontalmente en función de la carga y de los recursos disponibles.

El Gateway mantiene un registro de las solicitudes activas, asociando cada `request_id` con su correspondiente canal de comunicación. Esto habilita el enrutamiento correcto de las respuestas a cada cliente y asegura que los resultados sean entregados de forma ordenada y consistente.

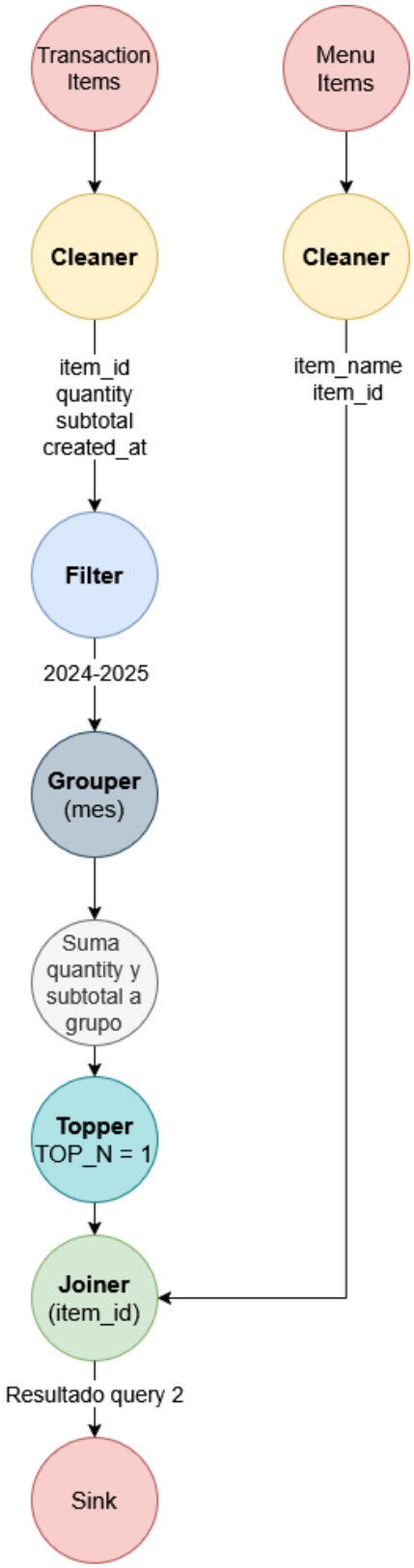
3.3. Flujo de datos

Los diagramas presentados a continuación ilustran el flujo de datos a través del sistema. El procesamiento se inicia en la *source* de datos, donde se eliminan las columnas no relevantes. Posteriormente, se aplican los filtros específicos de cada *query*, seguidos de las etapas de agrupamiento, unión y ordenamiento necesarias para producir el resultado final.

3.3.1. Flujo de datos por consultas 1, 3 y 4



3.3.2. Flujo de datos por consulta 2

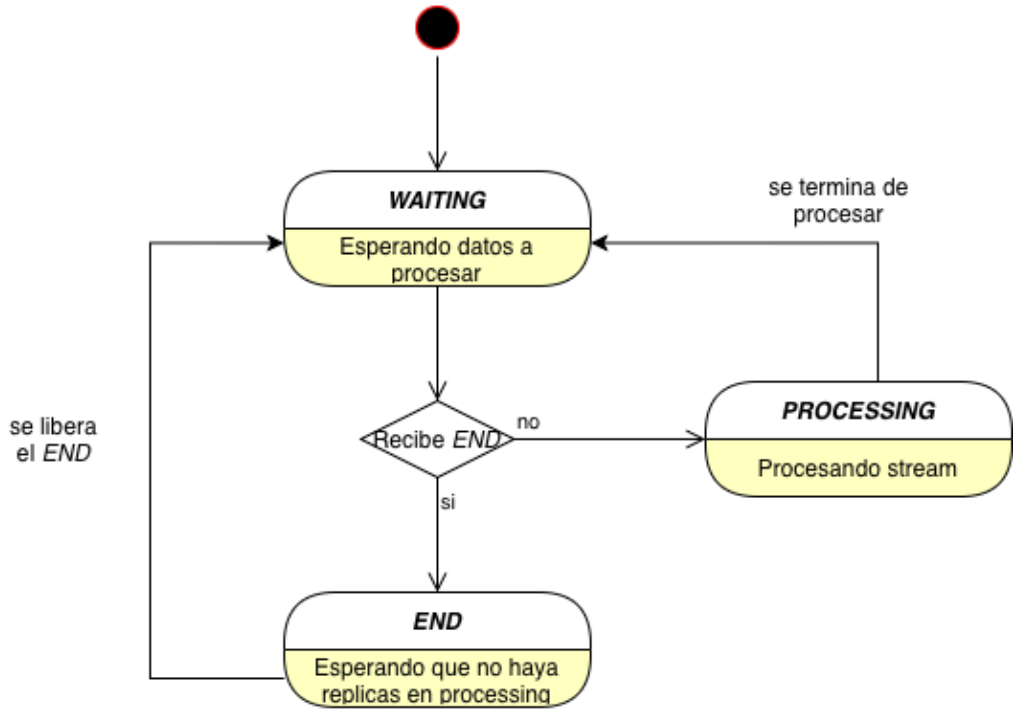


3.4. Manejo de estados en sincronización de mensajes

El *Worker State Manager* (*WSM*) es el componente encargado del mecanismo de sincronización entre réplicas. Cuando múltiples contenedores procesan un *streaming* de datos cuyo origen es una misma cola de entrada, se debe garantizar que el mensaje de *END* —el cual determina que todos los *batchs* con información fueron procesados— se envíe siempre al final.

Para lograr esto, se cuenta con dos herramientas clave. En primer lugar, antes de comenzar a procesar los datos, estos atraviesan un secuenciador que asigna números a cada uno de los *batchs*. De esta forma se puede identificar qué elemento falta procesar. Una vez asignados los números, cada réplica se conecta por *TCP* a su *WSM*. (Para cada conjunto de réplicas existe un *WSM* propio. Por ejemplo, los *cleaners* de transacciones poseen uno, mientras que los *filters* de *amount* poseen otro).

Una vez establecida la conexión, el *client* envía al servidor actualizaciones de estado, indicando si se encuentra en estado *WAITING*, *PROCESSING* o *END*. En los casos de *PROCESSING* y *END*, este estado verboso es acompañado del *request id* correspondiente y de la posición del mensaje en la secuencia, lo cual permite identificar los mensajes faltantes. Se observa en el diagrama, como las replicas atraviesan los estados mencionados.



El *WSM*, a medida que recibe estas actualizaciones, registra en disco todos los números que ya fueron procesados, con el fin de determinar cuáles faltan.

Cuando una réplica desencola el mensaje de *END*, se consulta al *WSM* para verificar si todos los mensajes anteriores ya fueron procesados y notificados. En caso afirmativo, el *WSM* libera el *END* y el sistema continúa con el siguiente paso. En caso contrario, la réplica permanece esperando, realizando un *poll* con *exponential backoff* hasta que las demás réplicas terminen de procesar, para finalmente poder liberar el *END*.

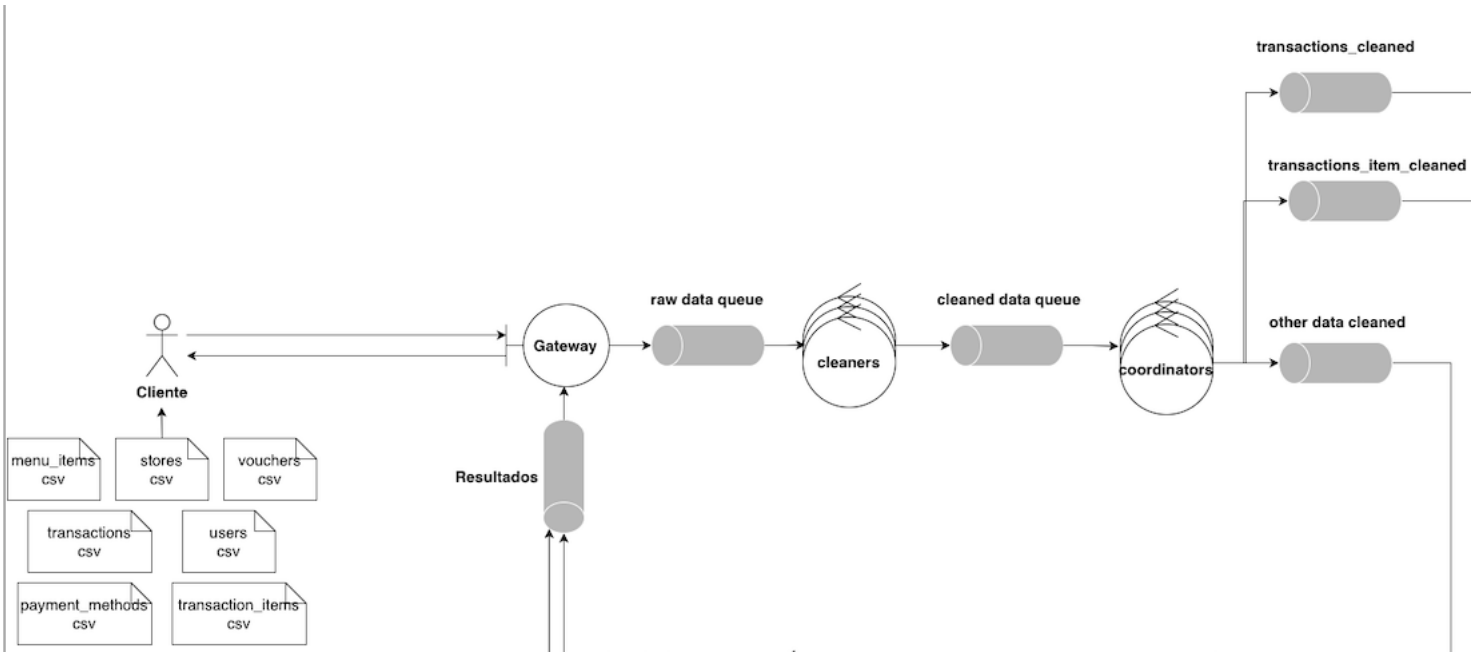
4. Vista Física

4.1. Arquitectura del Sistema (Diagrama de Robustez)

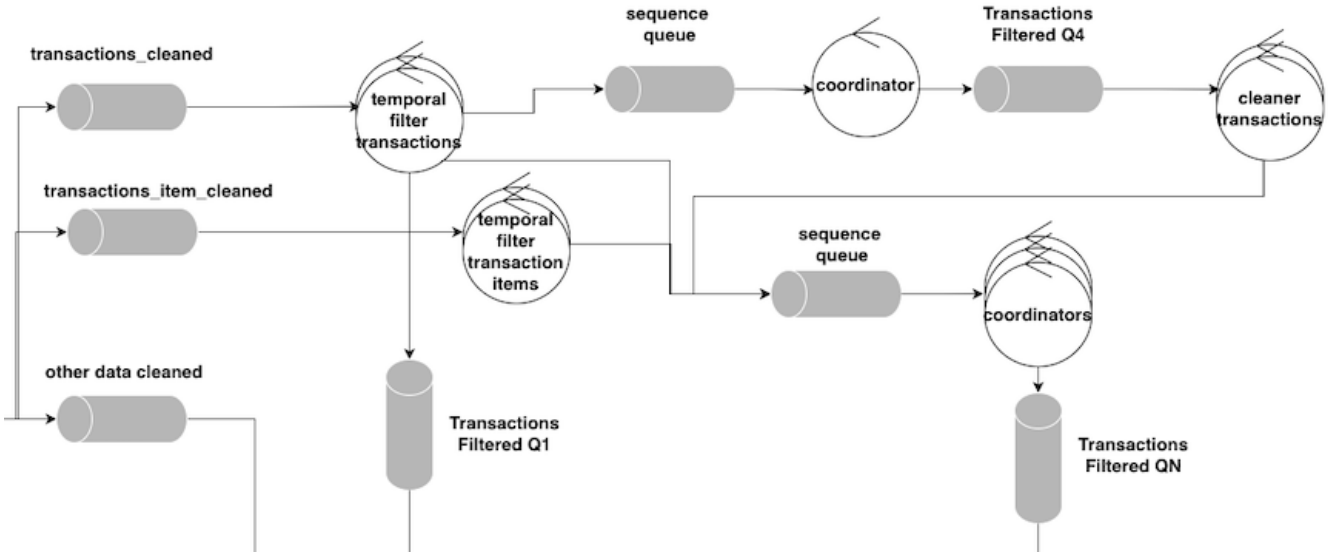
Este diagrama representa cómo diferentes actores y componentes interactúan para procesar las consultas, comunicándose a través de colas. A continuación se detalla su funcionamiento.

En la primera imagen se observa el inicio del flujo de datos: el cliente envía la información al *gateway*, que luego la distribuye (dependiendo del *data type*) al *cleaner* correspondiente. Por simplicidad, no se diagraman todas las colas; sin embargo, para cada tipo de dato existe una cola, un *cleaner* y un coordinador específico.

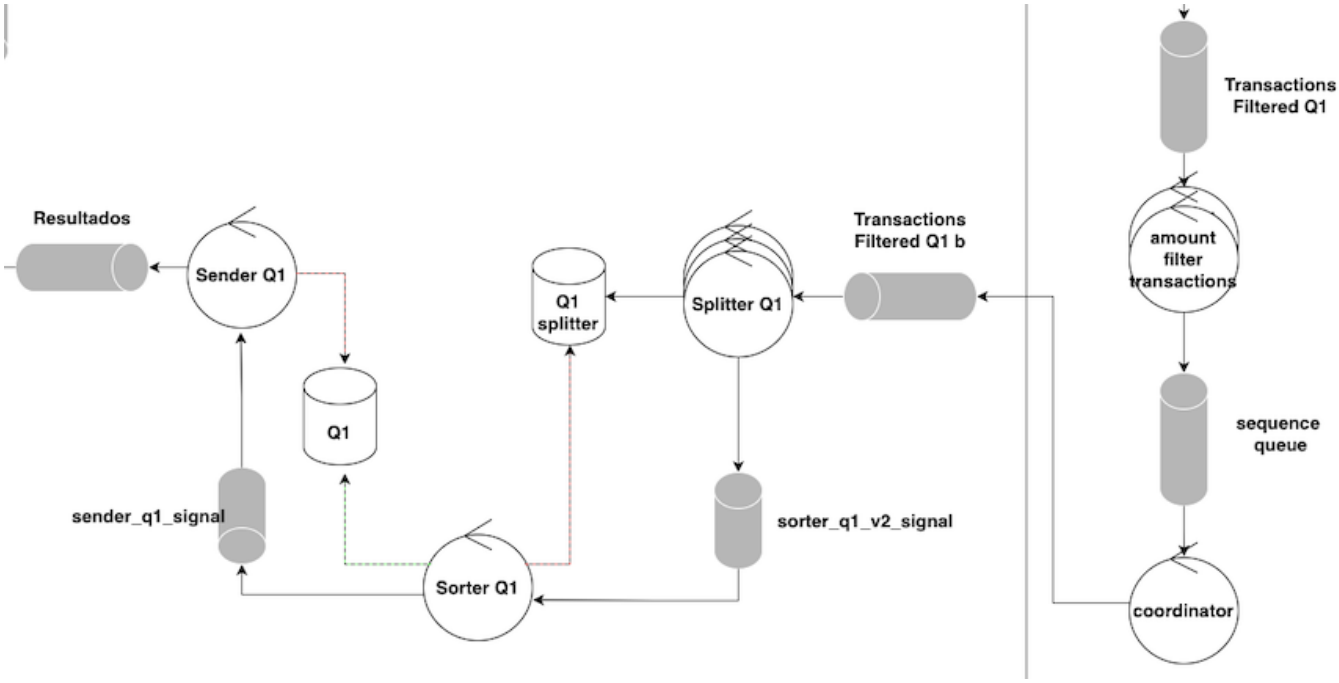
Una vez realizada la limpieza, la información se bifurca en *transactions*, *transaction items* y *other data*. Esta última representa las colas individuales de los demás tipos (*users*, *menu items*, *stores*), los cuales serán utilizados por el *joiner* en las operaciones finales de Q2, Q3 y Q4.



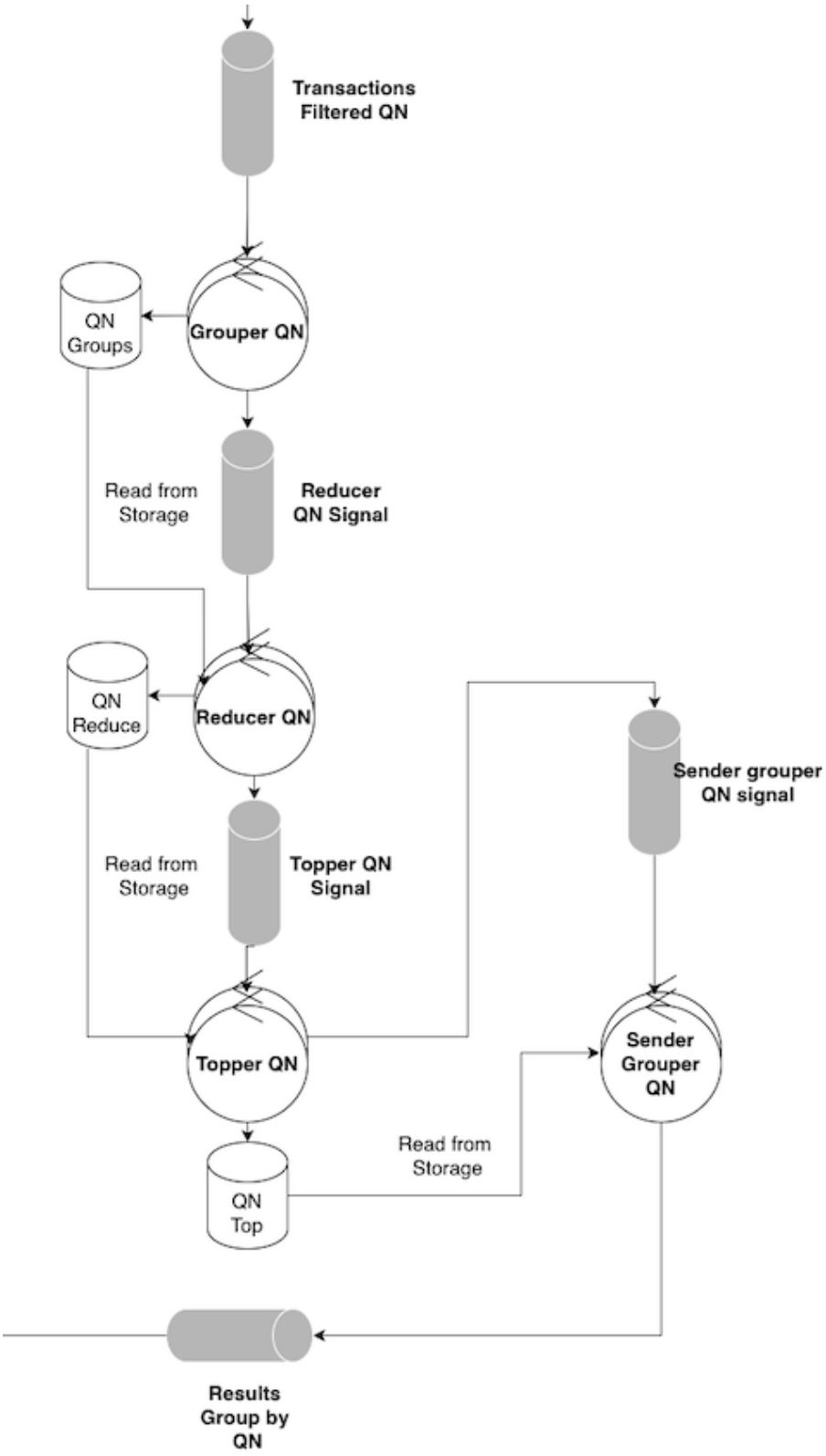
Luego, las distintas *rows* viajan hacia los filtros temporales, que se encargan de verificar las condiciones de cada *query* y enviar los datos hacia los agrupadores o hacia el filtro de monto, según corresponda. Se observa que, en cada caso, después de un conjunto de réplicas, los mensajes pasan por un coordinador dedicado, que vuelve a secuenciarlos asignándoles un número y cumple además la función de descartar duplicados en caso de errores.



Una vez que las transacciones de la Q1 son filtradas por tiempo, pasan por el filtro de *amount*. Luego llegan a los *splitters*, que se encargan de generar ordenamientos parciales, los cuales son completados posteriormente por el *sorter* utilizando una estrategia similar a *merge sort*. Finalmente, se obtiene el resultado final y se encola en la cola de *results*.

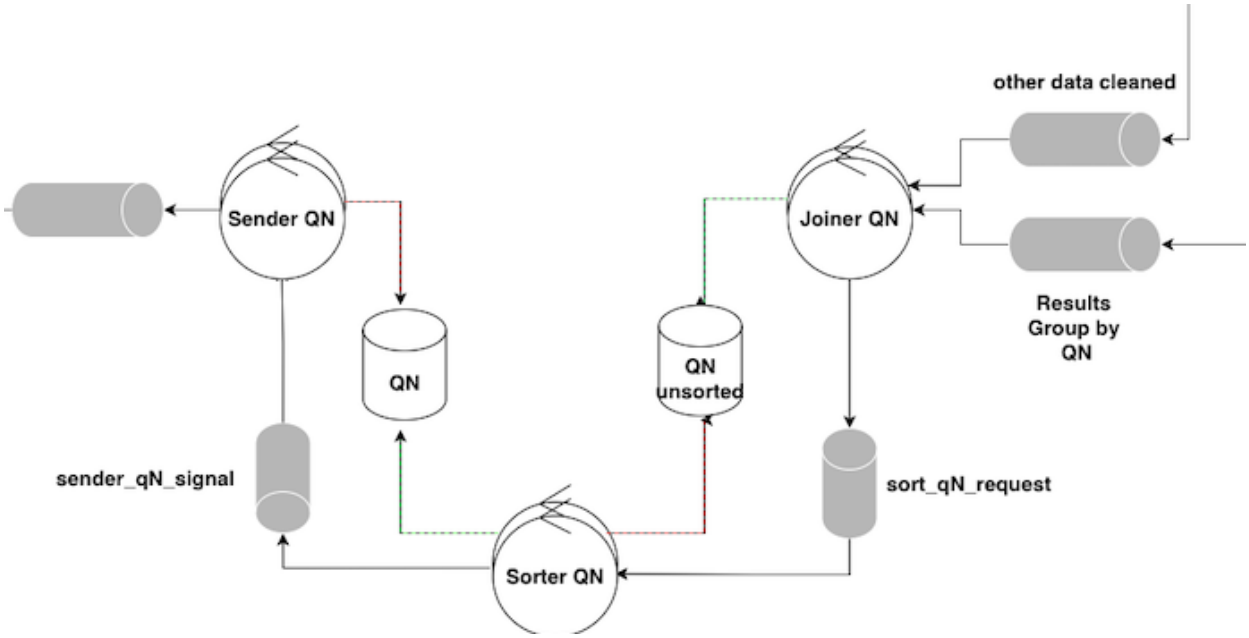


El flujo de datos continúa con el agrupamiento de los registros correspondientes a las *queries* 2, 3 y 4. Por simplicidad, y dado que su comportamiento es idéntico, se las representa en conjunto como QN. En este proceso se ejecuta una operación de estilo *map-reduce*: el *grouper* construye agrupamientos parciales, para luego obtener el resultado final en el *reducer*. Posteriormente, se calcula el *top n* según los criterios de cada *query*, y finalmente se notifica al *sender* para que envíe los datos agrupados al *joiner*.

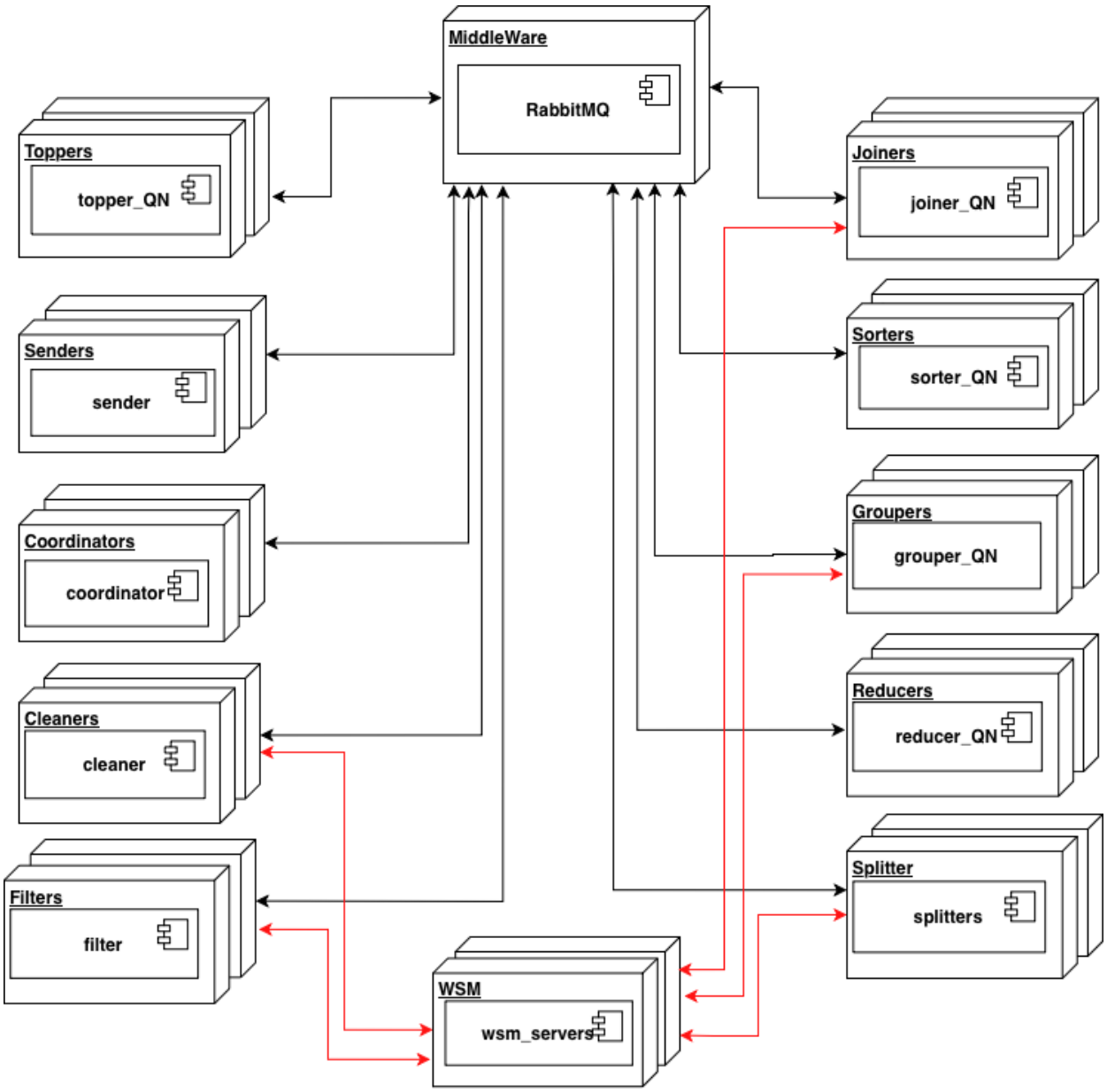


Finalmente, se mantiene la siguiente estructura para las Q1, Q2 y Q3: el *joiner* recibe, por un lado, los resultados agrupados de cada *query* en su cola correspondiente, y por otro, los datos

limpios de los tipos de datos que no requieren agrupamiento. Su función es unir ambos conjuntos y almacenar la unión completa en disco. Luego, notifica al *sorter*, que en cada caso ordena los registros según los lineamientos definidos y, una vez finalizado el ordenamiento, notifica al *sender* para enviar el resultado a la cola de *results*, desde donde el *gateway* consumirá y reenviará al cliente el resultado final.



4.2. Despliegue del Sistema



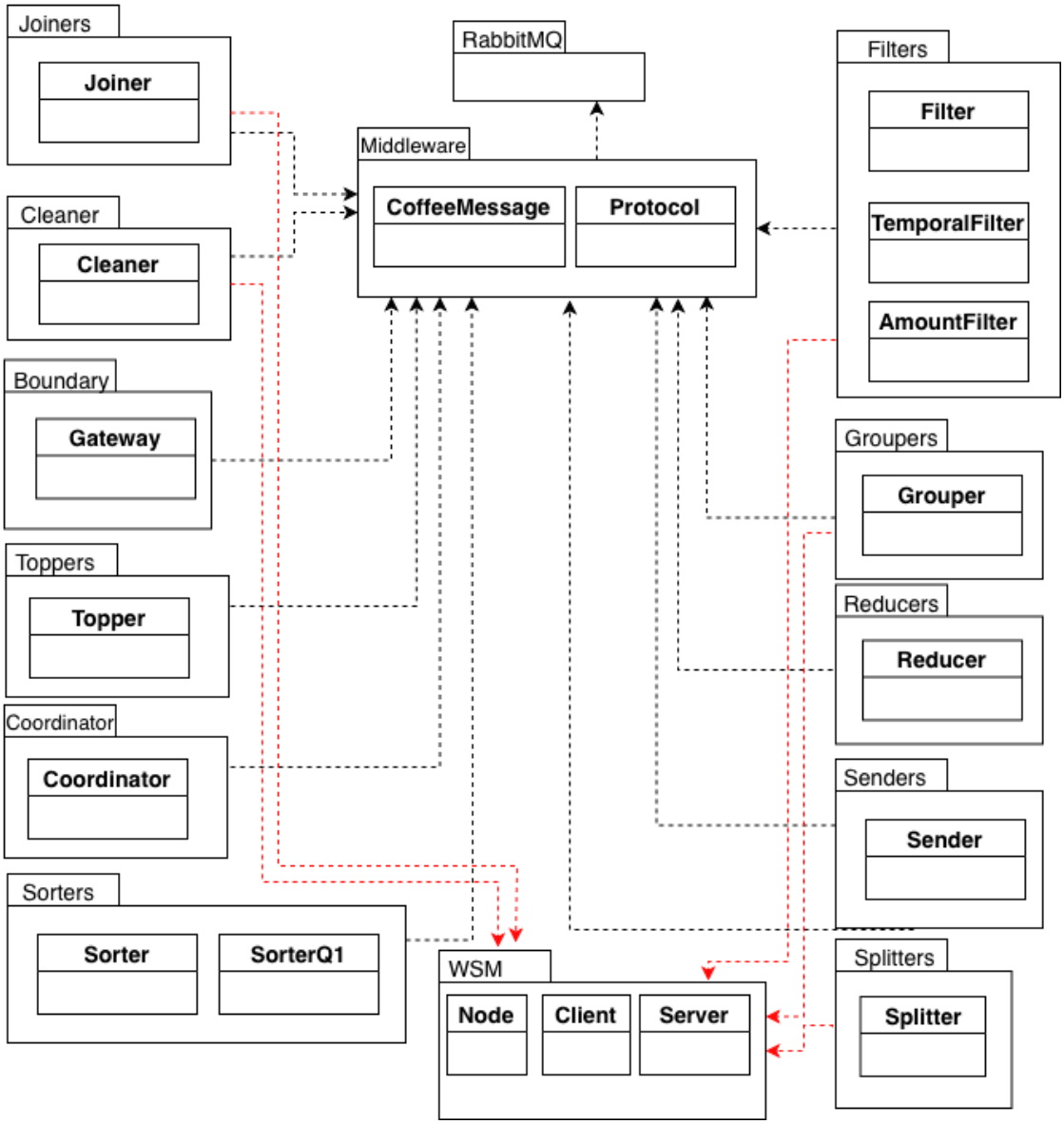
Estos diagramas muestran el sistema distribuido diseñado, en el cual el procesamiento de las consultas se reparte entre filtros, agrupadores, *joiners*, *senders*, y otros componentes especializados. Toda esta interacción es coordinada por un *Middleware* que cumple el rol de entidad orquestadora del flujo de datos.

Como se observa, RabbitMQ se integra dentro del *Middleware* para gestionar la comunicación y la distribución de tareas entre las distintas etapas del sistema, asegurando un enrutamiento eficiente y desacoplado de los mensajes que circulan a lo largo del *pipeline*.

Además se presenta la interacción entre los diferentes conjuntos con sus *WSM servers*

5. Vista de Desarrollo

5.1. Diagrama de Paquetes



El objetivo de este diagrama es presentar la organización lógica de los paquetes que componen el sistema, así como las dependencias entre ellos. Cada paquete agrupa responsabilidades específicas dentro del flujo de procesamiento: filtros, agrupadores, *joiners*, *senders*, coordinadores y módulos de frontera como el *Gateway*.

La comunicación entre estos componentes se articula a través del *Middleware*, donde reside el módulo *CoffeeMessage* encargado de la serialización y deserialización de los mensajes, junto con el módulo *Protocol*, responsable de estructurar y normalizar los intercambios. RabbitMQ participa como infraestructura de mensajería, permitiendo desacoplar los paquetes y facilitar la distribución de tareas en el sistema.

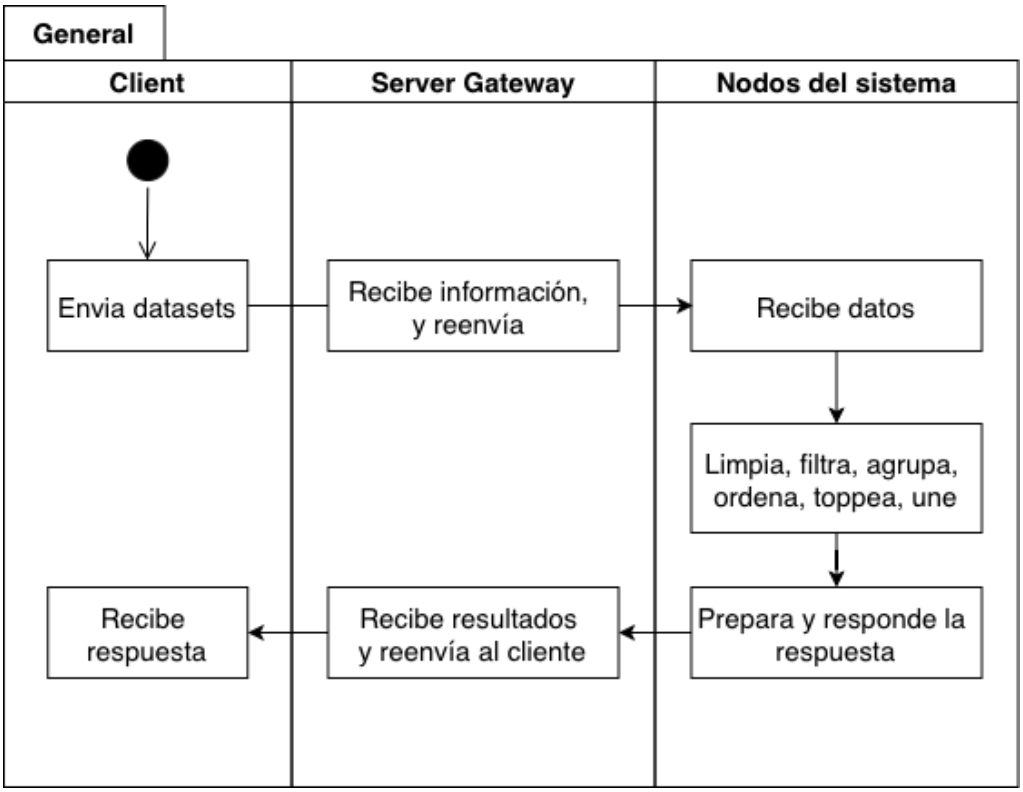
Este diagrama se centra en mostrar cómo se divide el sistema en unidades lógicas cohesivas, más allá de su implementación física o del entorno de ejecución.

5.2. Diagramas de Actividades

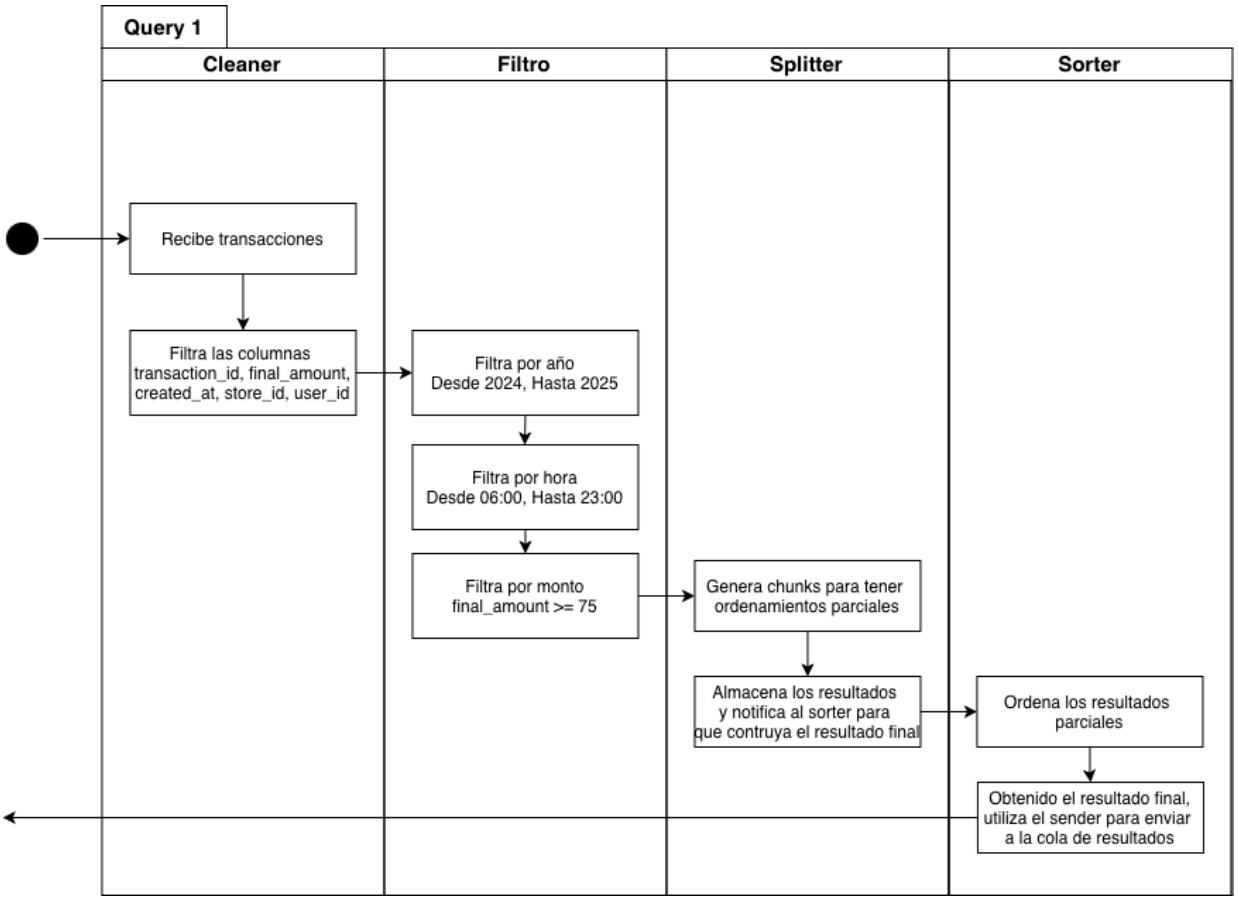
A continuación se presentan una serie de diagramas que nos permiten observar el flujo del sistema en los casos más relevantes, para la correcta comprensión de las intenciones de diseño del equipo.

5.2.1. Entrada al sistema

El siguiente diagrama se utiliza para ejemplificar el origen y destino de los datos, junto con la interacción del cliente con el gateway. En los diagramas subsiguientes, estas entidades no serán detalladas nuevamente, ya que se repiten en todos ellos.

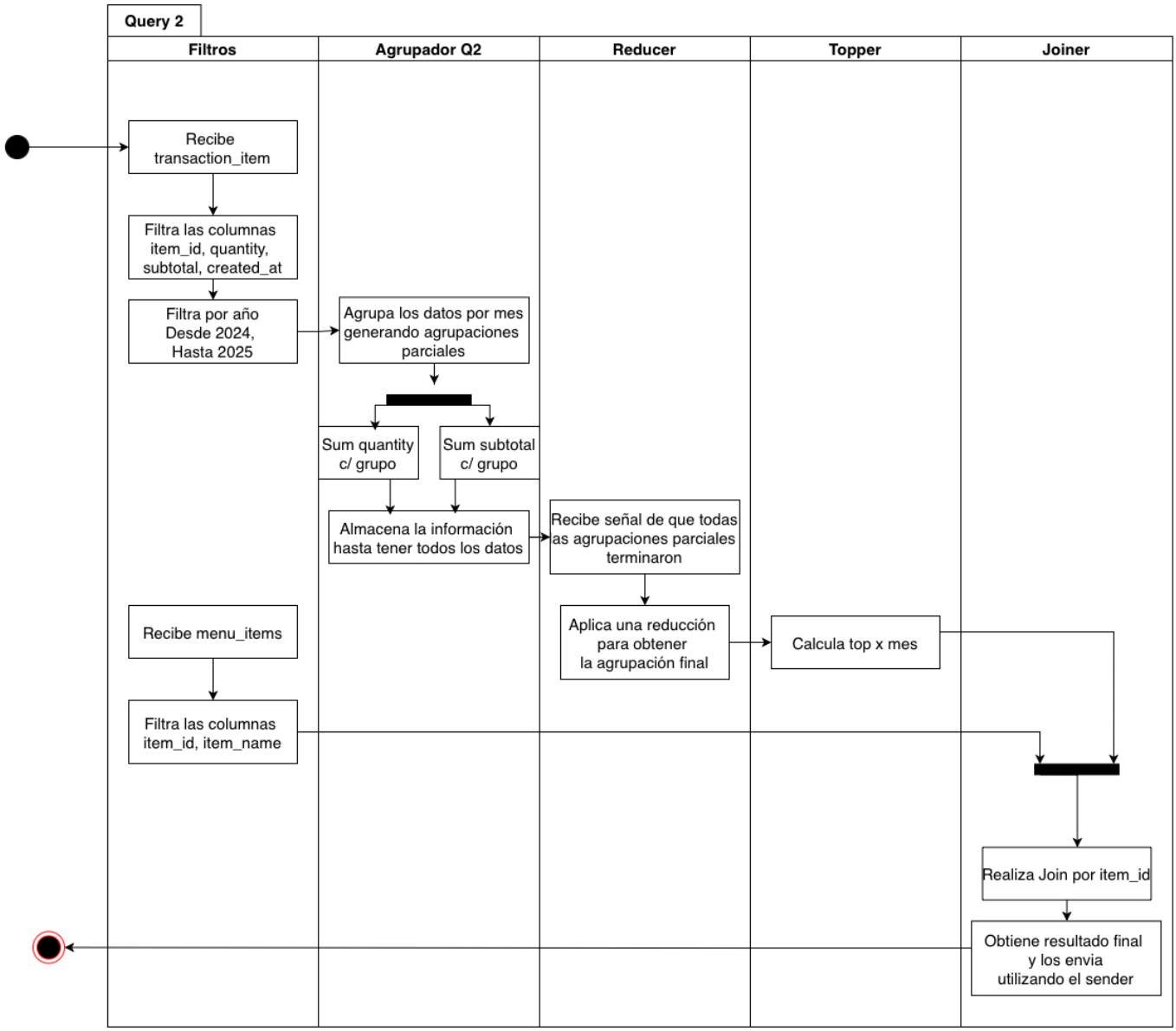


5.2.2. Query 1



En la *query* número uno podemos observar claramente el flujo del *pipeline* en nuestro modelo de *workers por filter*. La comunicación se inicia con el primer filtro de columnas (*cleaner*). Luego se encadenan las distintas operaciones de filtrado entre los nodos dedicados. Posteriormente, los *splitters* se encargan de generar ordenamientos parciales de un tamaño definido, para ser finalmente unificados y ordenados por completo por el *sorter* mediante una estrategia similar a *merge sort*. Una vez finalizado el ordenamiento, se utiliza nuestra estructura de *sender* para enviar los resultados a la cola correspondiente.

5.2.3. Query 2

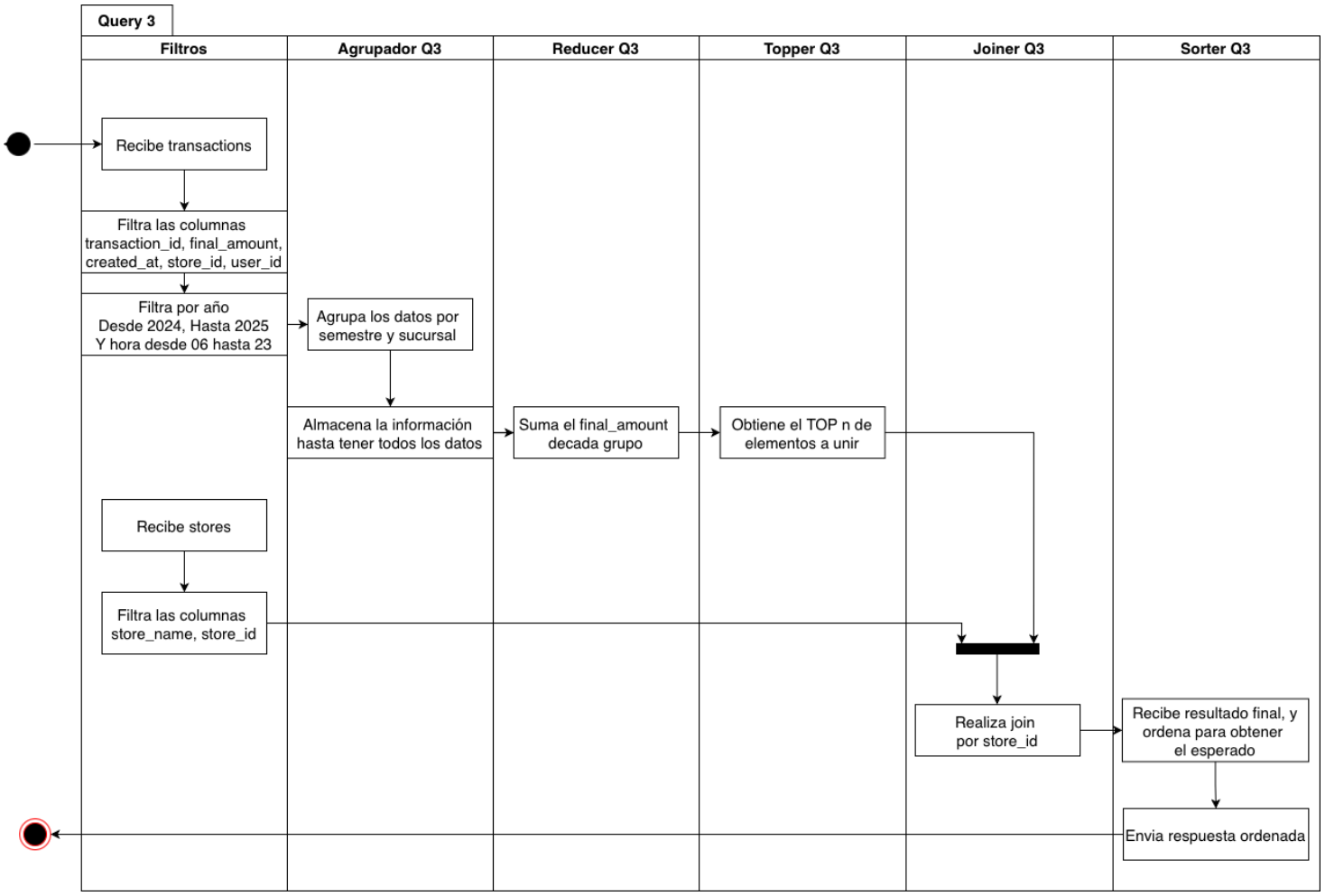


En el diagrama de actividades de la *query* dos se puede notar un caso clave, que es la situación de paralelismo al tener que realizar un *sum* sobre el mismo set de datos pero para diferentes columnas, tanto por cantidad como por monto. De esta forma observamos como el camino se bifurca para dar lugar a dos procesos que trabajan de forma paralela en resolver la *query* 2, ya que la misma implica dos selecciones de datos.

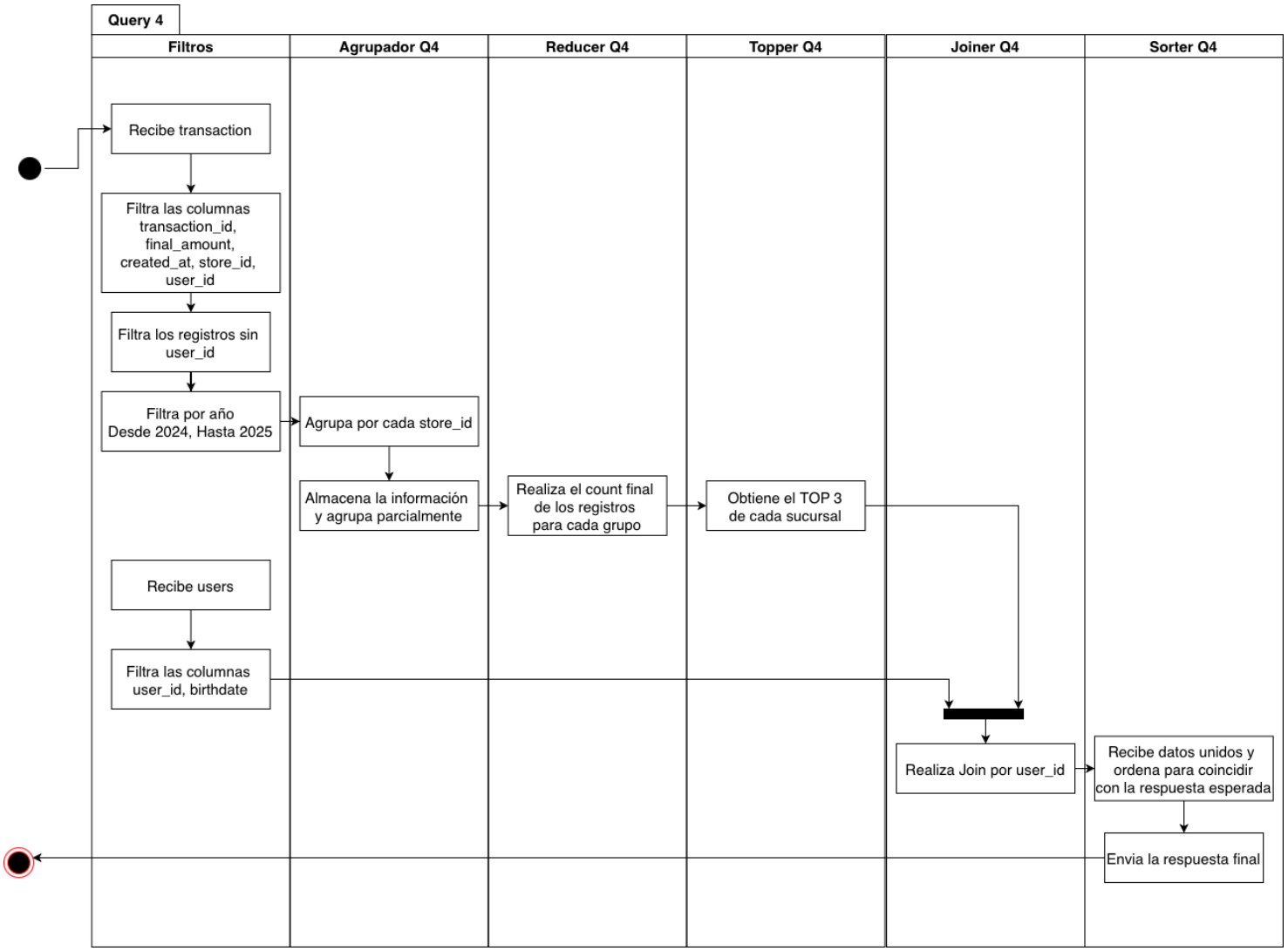
Además se contempla el caso del envío de dos *datasets* que deben ser procesados también de forma paralela, a medida que van llegando y es posible. Para finalmente terminar en el *Joiner* Q2, quien se encarga de juntar los datos de ambos *datasets*, y obtener el resultado final que posteriormente recibe el cliente.

En las siguiente subsecciones se puede observar el flujo de las *queries* 3 y 4, que comparten similitud en el funcionamiento, pero funcionan como ejemplo práctico de lo que sucede.

5.2.4. Query 3



5.2.5. Query 4


Query 4
Filtros
Agrupador Q4
Reducer Q4
Topper Q4
Joiner Q4
Sorter Q4
Recibe transaction
Filtra las columnas transaction_id, final_amount, created_at, store_id, user_id
Filtra los registros sin user_id
Filtra por año Desde 2024, Hasta 2025
Agrupa por cada store_id
Almacena la información y agrupa parcialmente
Realiza el count final de los registros para cada grupo
Obtiene el TOP 3 de cada sucursal
Recibe users
Filtra las columnas user_id, birthdate
Realiza Join por user_id
Recibe datos unidos y ordena para coincidir con la respuesta esperada
Envia la respuesta final

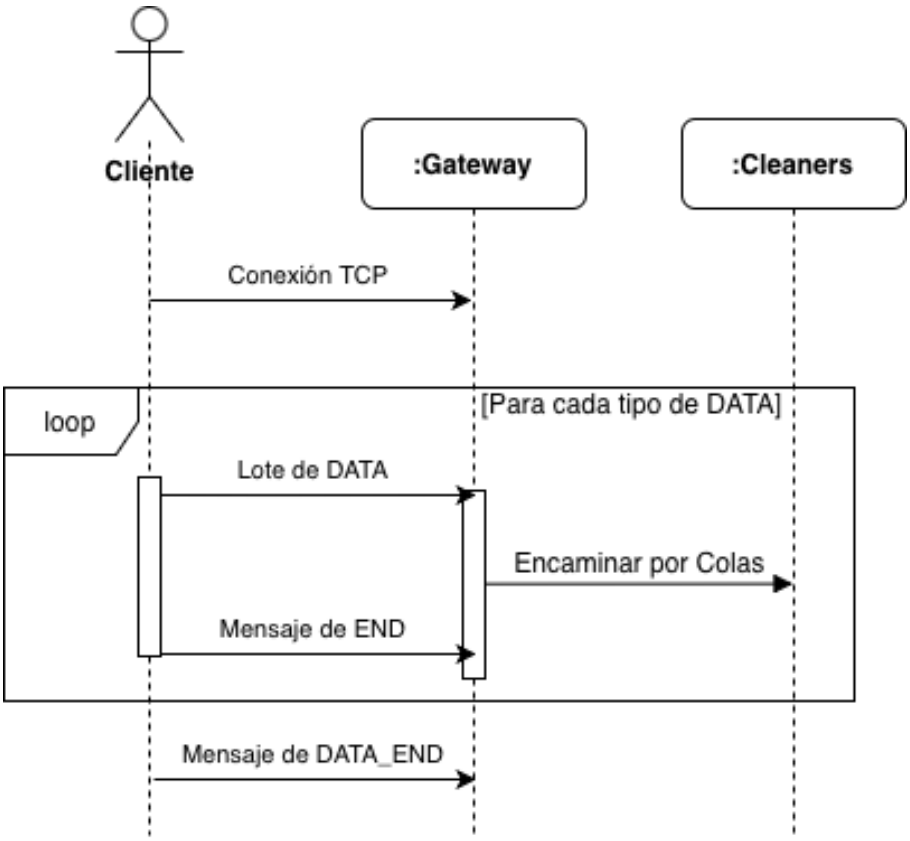
6. Vista De Procesos

A continuación se observan casos relevantes haciendo uso de diagramas de secuencia para mayor esclarecimiento de la arquitectura, enfocada en los procesos e intercambios de mensajes.

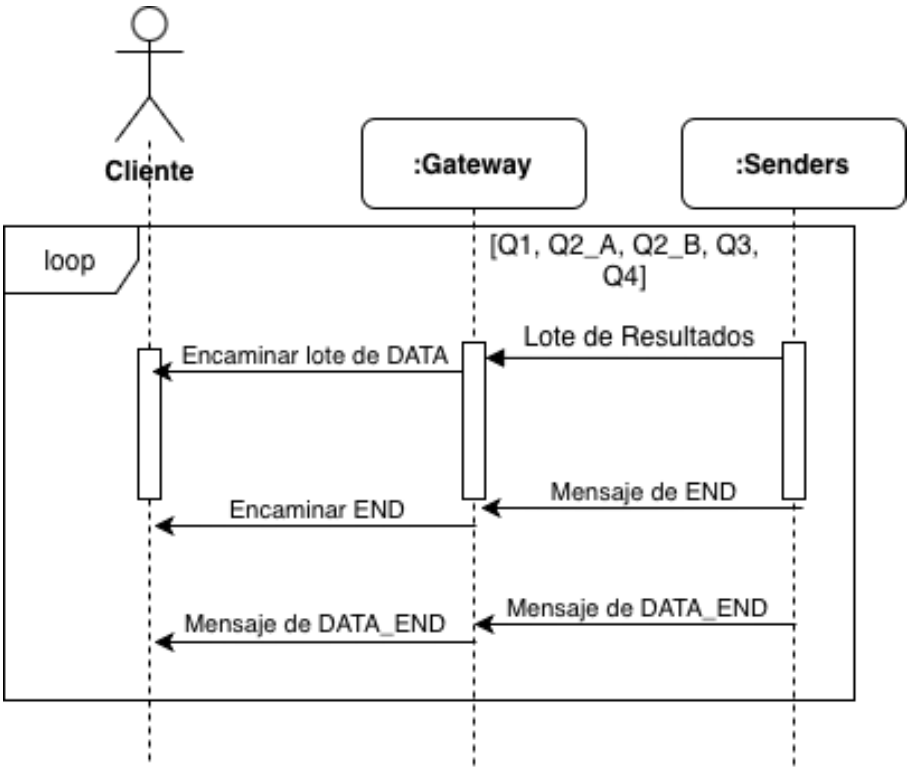
6.1. Envío y recepción de información

En el primer diagrama se ejemplifica cómo el cliente implementa una conexión TCP con el *gateway* y comienza a enviar en *loop* toda la información de cada tipo de dato en *batches*. Cuando el *gateway* la recibe, dependiendo del tipo, la redirige a la cola correspondiente para que inicie el proceso de *pipes and filters*.

Cada vez que un tipo de dato termina de enviar todos los *batches*, se envía un mensaje de *end* para ese tipo. Además, cuando toda la información fue enviada, también se encola un mensaje *DATA_END* final que identifica la completitud del envío de datos.



Cuando la información completa todo el recorrido, regresa al *gateway* a través de los *senders* finales, que encolan los datos en la cola de resultados. A medida que estos van llegando, el *gateway* reenvía al cliente los resultados de las consultas. De forma análoga a la estructura de mensajes de finalización utilizada en el envío, el cliente puede identificar cuándo ha recibido la totalidad de los datos.



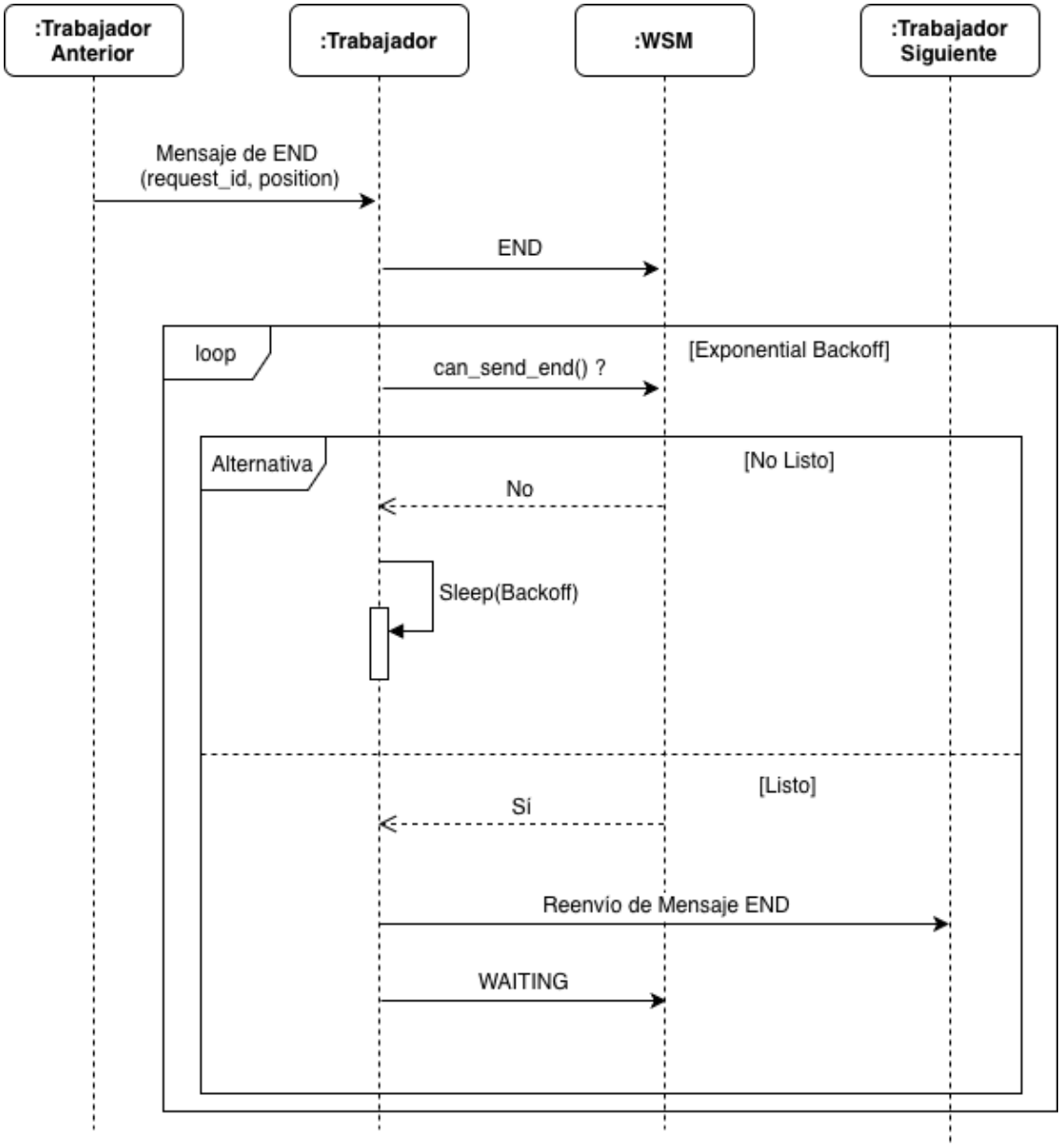
6.2. Sincronización de réplicas

A continuación se estudia el funcionamiento de la sincronización de las réplicas. Al tener varios trabajadores procesando *streaming* de datos en paralelo, todos deben notificar su estado y el identificador del mensaje procesado al WSM, para que este lleve la cuenta de los mensajes restantes.

Cuando uno de los *workers* recibe el mensaje *END* desde la cola de entrada, esto indica que todos los demás mensajes de datos ya fueron desencolados, aunque no necesariamente procesados.

En ese momento, el trabajador que recibe el mensaje *END* consulta al WSM mediante la operación *"can send end?"*. El *Worker State Manager* se encarga de cerciorarse de que todos los mensajes hayan sido completamente procesados.

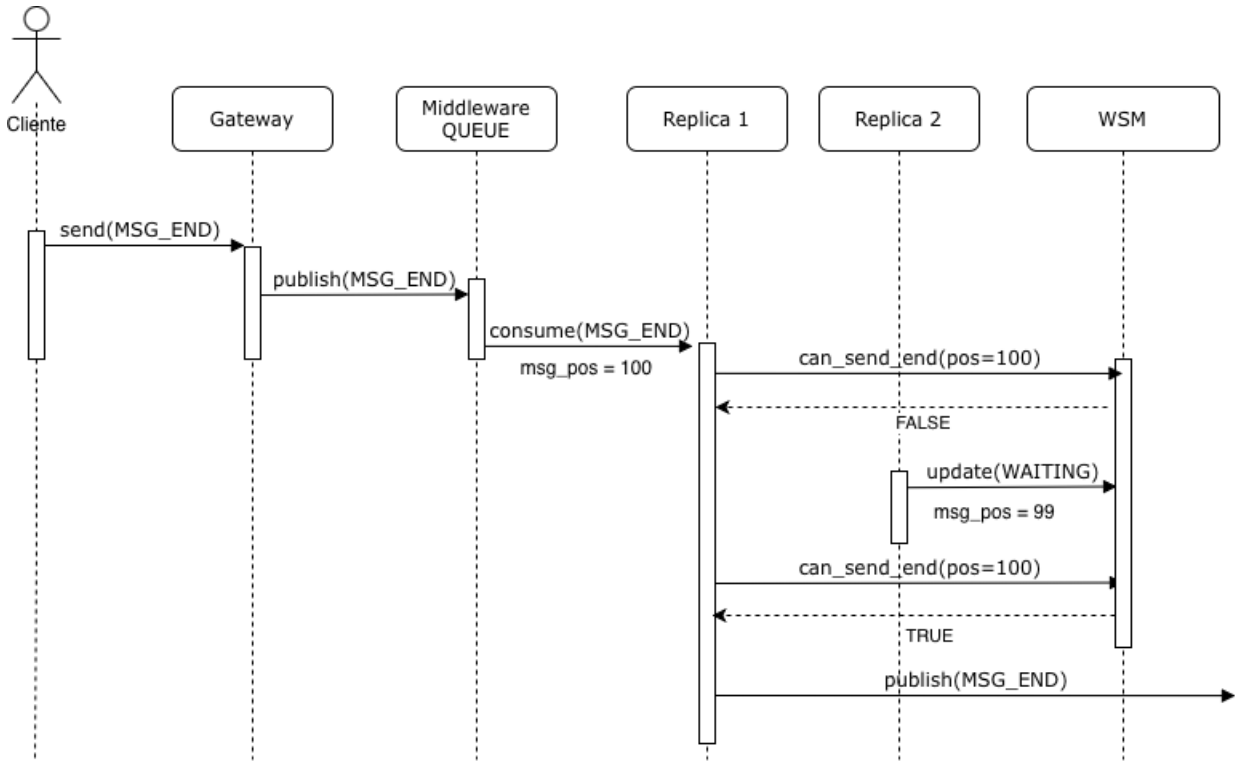
En caso afirmativo, responde que el mensaje *END* puede ser encolado en la cola de salida. En caso negativo, el *worker* debe permanecer en espera, realizando *polling* con *exponential backoff*, hasta que el envío del *END* sea habilitado.



6.3. Protocolo de ENDs

El protocolo de terminación es un mecanismo jerárquico y coordinado que asegura el cierre ordenado de los flujos de datos distribuidos. Se inicia en el **Client**, que envía un mensaje `MSG_TYPE_END` específico por cada tipo de archivo (Transactions, Users, etc.) una vez completada su transmisión, y culmina con un `DATA_END` global. El **Gateway** actúa como enrutador, reenviando estos mensajes a las colas RabbitMQ correspondientes.

La complejidad reside en los **Workers** y su interacción con el **WSM**. Dado que los mensajes pueden procesarse desordenadamente debido a la concurrencia de réplicas, un worker no puede propagar un `END` inmediatamente al recibirlo. Como explicamos en la sección anterior, se consulta al WSM con (`can_send_end`, y es el WSM quien autoriza al worker a propagar la señal a la siguiente etapa. En nodos de agregación como el **Joiner**, existe una barrera adicional (`can_send_last_end`) que sincroniza la finalización de múltiples flujos de entrada antes de cerrar el procesamiento.



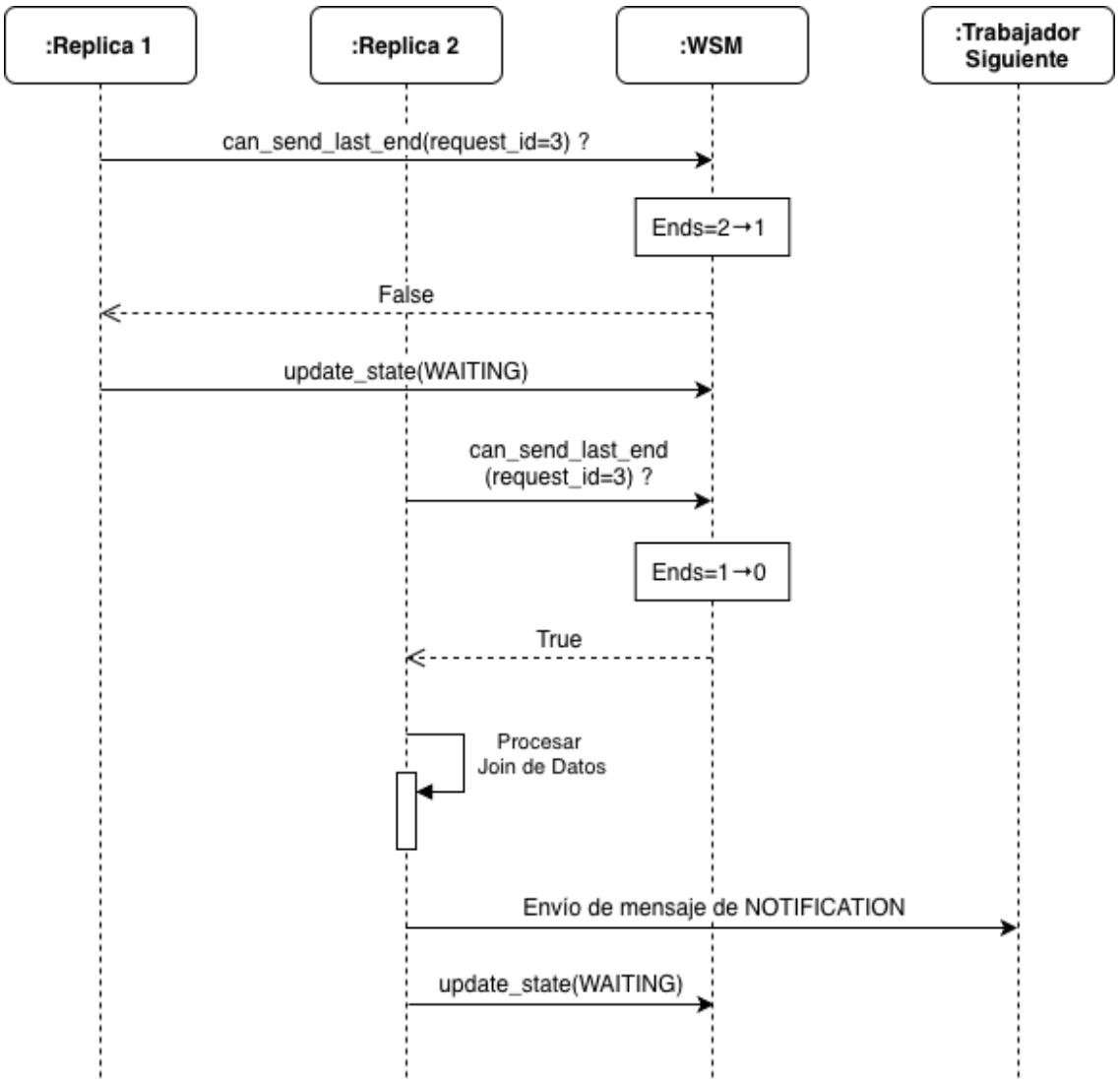
En la figura podemos observar como funciona el protocolo de END en el sistema. El Cliente envía un MSG_END que es consumido prematuramente por la Réplica 1 (posición 100). Al consultar al WSM (can_send_end), recibe FALSE porque aún falta procesar el mensaje previo (99). La Réplica 1 queda bloqueada en espera hasta que la Réplica 2 finaliza su tarea pendiente y notifica al WSM con update(WAITING) para la posición 99 (recordar que el WSM solo registra una posición como çompletadaçuando detecta una transición PROCESSING ->WAITING). Al reintentar la consulta, el WSM valida que la secuencia está completa, devuelve TRUE y permite a la Réplica 1 propagar finalmente la señal de fin.

6.4. Sincronización de uniones

Cuando un *joiner* debe realizar la unión de dos tipos de datos, tendrá n colas de entrada, dependiendo de la cantidad de flujos a unir (por ejemplo, dos). En caso de existir múltiples réplicas del *joiner*, estas podrán ejecutar uniones parciales en paralelo. Las réplicas irán desencolando información desde sus colas de entrada, almacenando resultados parciales en disco.

Cuando alguna de ellas recibe el primer mensaje *END*, notificará al WSM mediante la señal *"can send last end?"*. El WSM conoce cuántos mensajes *END* debe recibir en total, a partir de la cantidad de colas de entrada configuradas. Por lo tanto, al recibir la primera señal, descontará en su contador interno y devolverá *false* hasta que dicho contador llegue a cero.

Una vez que se hayan recibido todos los *END*, el WSM responderá a la réplica correspondiente que puede enviar el *END* final. De esta forma se dispara el proceso encargado de unir los datos finales, para posteriormente enviarlos al siguiente paso del pipeline.

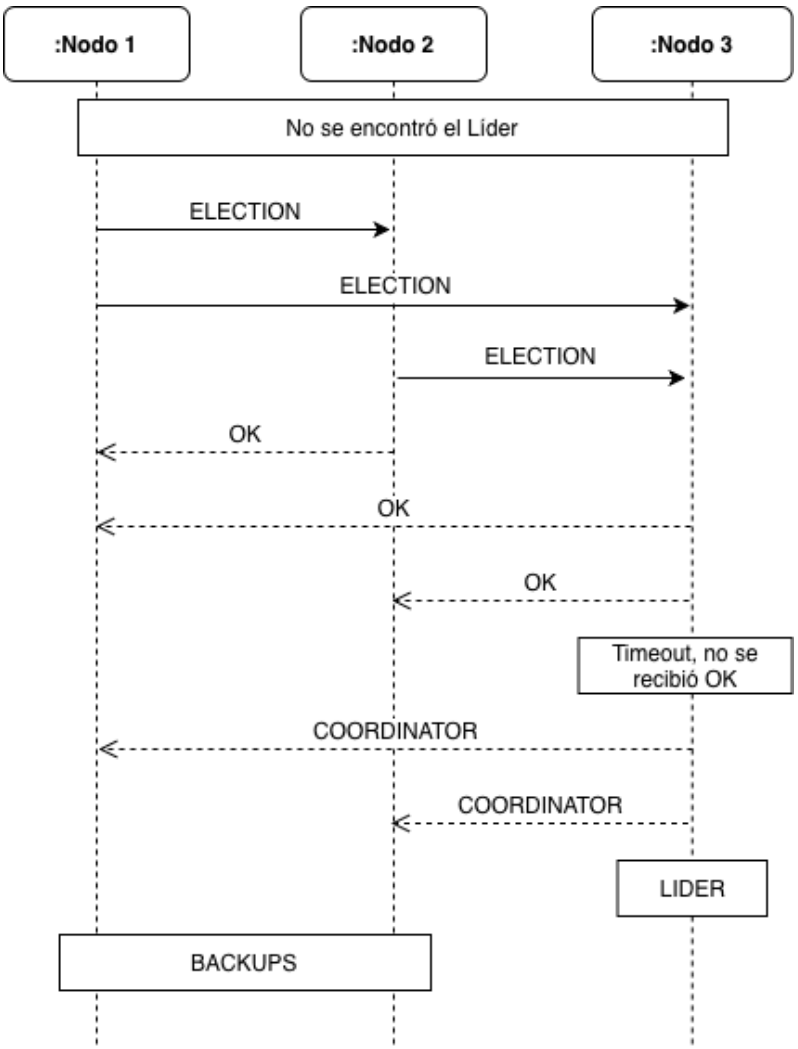


6.5. Técnicas de tolerancia a fallos

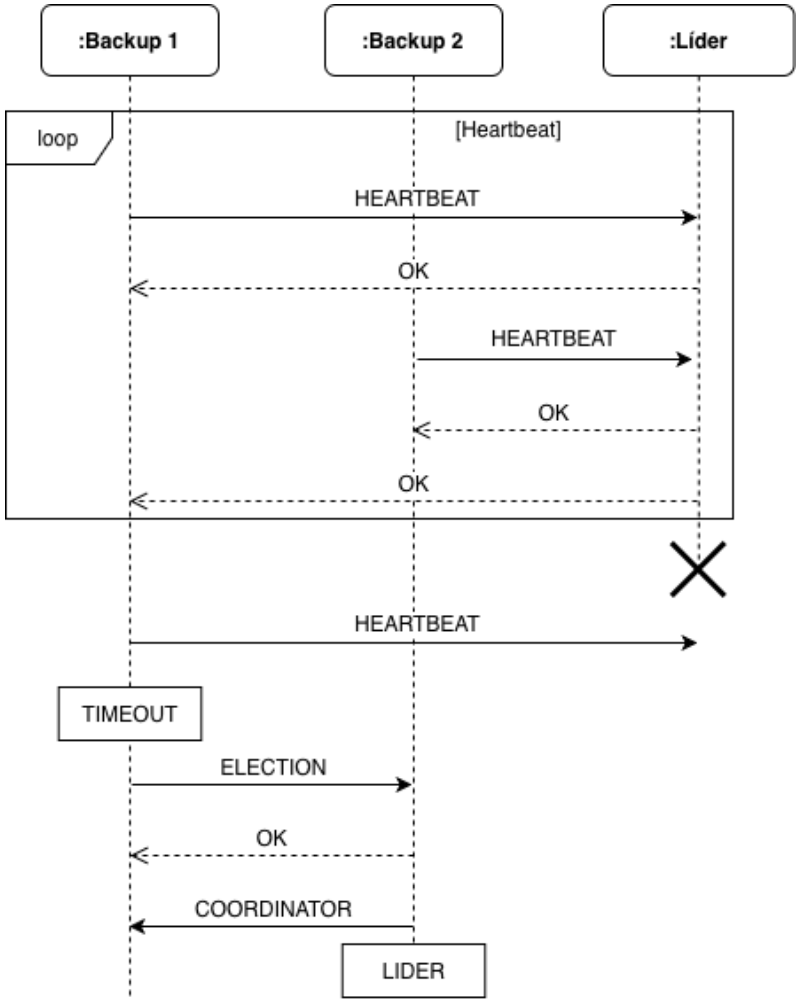
6.5.1. Backups

Al ser el *WSM* un servidor que centraliza información para sincronizar los mensajes, se requiere un sistema que garantice disponibilidad y tolerancia a caídas. Para ello se adopta un esquema de *Leader* con *Backups*.

Cuando el *WSM* se inicia, se levantan, por ejemplo, tres instancias en contenedores distintos que comparten un mismo volumen. Al conectarse, cada instancia envía por *UDP* un mensaje *READY*. Una vez que todos los *peers* se encuentran *READY*, se habilita el plano de control y se inicia una elección inicial, donde el nodo con mayor identificador se proclama *Leader* como se observa en la figura.



El *Leader* es el encargado de interactuar con las réplicas y de escribir los estados en disco. Las demás instancias funcionan como *Backups*, cuya labor es monitorear la salud del *Leader* mediante *heartbeats* enviados por *UDP*. Cada un segundo se envía un *heartbeat*, con un tiempo de espera de tres segundos para detectar fallas. Si algún *Backup* determina que el *Leader* dejó de responder, se inicia una elección utilizando el algoritmo *Bully*. Para evitar condiciones de carrera durante este proceso, se emplean bloqueos internos que garantizan que solo un hilo pueda autoproclamarse líder.

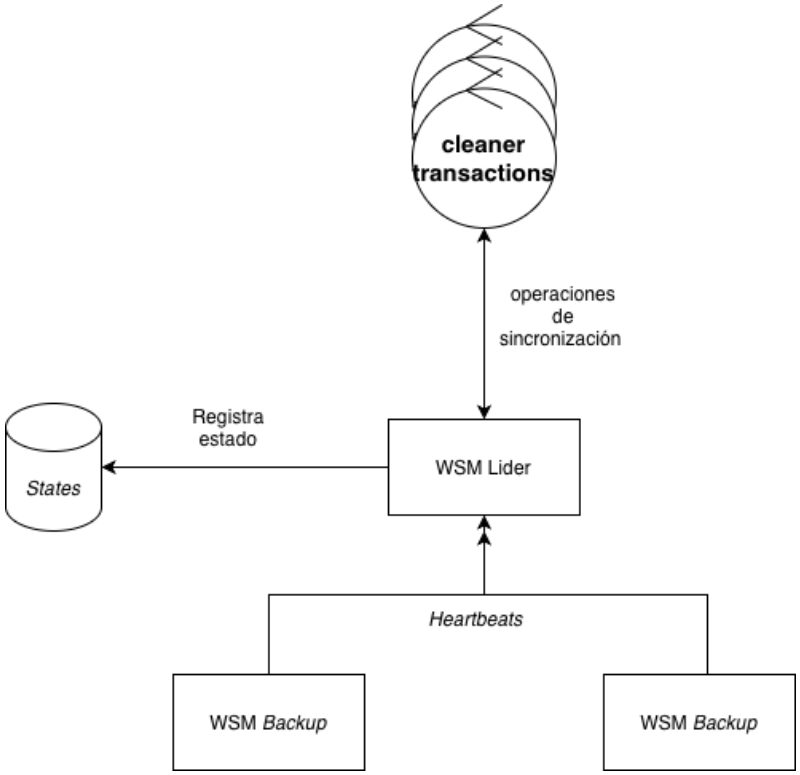


Antes de iniciar una elección, cada nodo intenta descubrir si ya existe un líder activo a través de los mensajes *WHO_IS_LEADER* y *LEADER_INFO*. Este mecanismo reduce elecciones innecesarias y contribuye a la estabilidad del sistema. En esta arquitectura, el plano de control —responsable de heartbeats, elecciones y descubrimiento de líder— funciona enteramente sobre *UDP*, mientras que el plano de datos, donde opera el servidor *WSMServer*, utiliza *TCP*. Esta separación explícita evita interferencias entre ambos flujos de comunicación y mejora la robustez general.

Una vez resuelta la elección, el nuevo *Leader* carga desde el volumen compartido el estado persistido y continúa recibiendo mensajes de las réplicas sin pérdida de consistencia. La persistencia en disco permite que la recuperación sea inmediata y que no se reprocesen *batchs* ya confirmados.

Desde el lado del cliente, cuando se detecta la pérdida de la conexión, se inicia un proceso de descubrimiento para identificar cuál es el nuevo *Leader* entre los nodos disponibles. Una vez encontrado, el cliente reenvía el último mensaje transmitido. El nuevo *Leader* verifica si dicho mensaje ya había sido registrado y, de no ser así, procede a almacenarlo. De esta forma se previene la pérdida de paquetes y se mantiene la entrega confiable incluso ante fallos del líder.

El siguiente diagrama representa una estructura general de la arquitectura, la cual se replica en cada conjunto de trabajadores que procesan datos en paralelo.



6.5.2. Réplicas

El sistema implementa tolerancia a fallos mediante la instanciación de múltiples **réplicas** para workers críticos, garantizando alta disponibilidad y procesamiento paralelo. Los workers que operan con réplicas son los **Cleaners** (Transactions, Users, Stores, etc.), **Splitter Q1**, **Temporal Filters** y **Groupers V2**.

El **Middleware** gestiona la comunicación con RabbitMQ utilizando *reconocimiento manual* (manual acknowledgment). Al iniciar, cada réplica configura el canal con **prefetch count = 1**, asegurando que solo procese un mensaje a la vez. El ciclo de vida implica que el Middleware recibe un mensaje y ejecuta el callback del worker. Si el procesamiento es exitoso, envía un **basic ack**, eliminando el mensaje de la cola. Si ocurre una excepción no controlada, captura el error y envía un **basic reject** con **requeue=True**, devolviendo el mensaje a la cola para ser procesado por otra réplica.

Si una réplica toma un mensaje y **se cae (crash o desconexión)** antes de enviar el ACK, RabbitMQ detecta la desconexión y re-encola el mensaje automáticamente. Antes de procesar, la nueva réplica consulta al WSM si la dupla *request id, position* ya fue procesada. Si la réplica

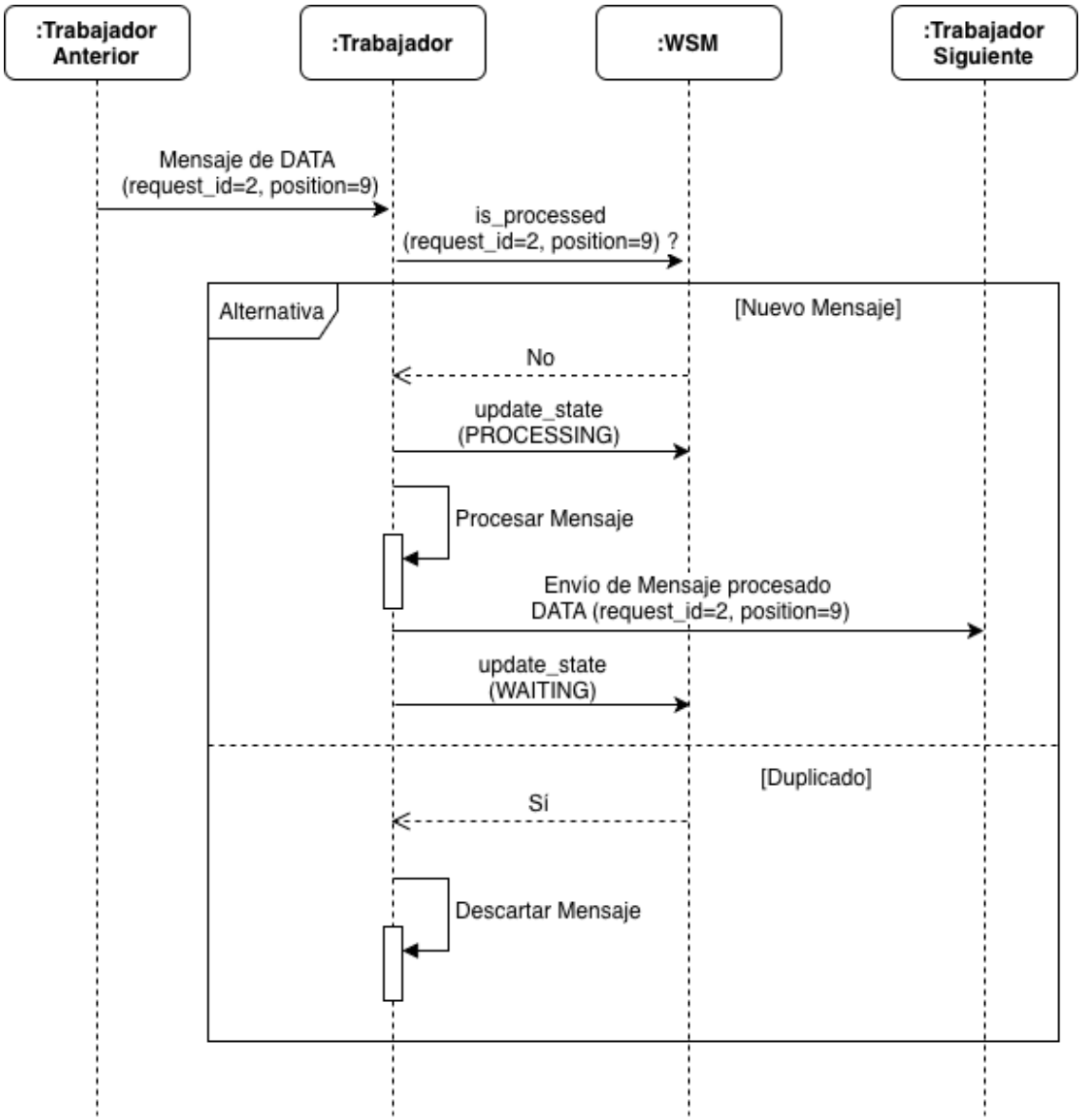
anterior cayó mientras procesaba, el WSM informa que no ha finalizado, permitiendo que la nueva réplica realice el trabajo. Si la réplica anterior terminó pero cayó justo antes del ACK, el WSM detecta el duplicado y la nueva réplica descarta el mensaje. Además, el WSM actúa como una barrera para las señales de **END**, esperando a que todas las réplicas terminen antes de autorizar a una sola a propagar la señal.

6.5.3. Deduplication

La *Deduplicación de Mensajes* es un mecanismo de tolerancia a fallos que previene el procesamiento duplicado de datos en un sistema distribuido donde los nodos pueden fallar y reintentar. El proceso funciona mediante el rastreo persistente de posiciones de entrada: cada worker mantiene en disco el conjunto de posiciones de mensaje ya procesadas para cada request. Cuando un nuevo mensaje llega, el sistema verifica atomicamente si su posición ya existe en el archivo de posiciones persistente. Si es duplicado, se descarta; si es nuevo, se añade al archivo usando `Worker.atomic_write()`.

La deduplicación se implementa, por ejemplo, en el **Coordinator**, que coordina outputs de múltiples workers distribuidos reordenando mensajes por posiciones secuenciales. El Coordinator mantiene dos estados persistentes: las posiciones de entrada ya recibidas (para detectar duplicados) y las posiciones asignadas (para recuperación tras crashes). Al procesar cada mensaje, verifica atomicamente si su posición es nueva, la persiste, asigna una nueva posición secuencial, y la guarda también atómicamente. Esto garantiza que incluso si el Coordinator falla durante el procesamiento, el estado en disco es siempre consistente. Al reiniciarse, el Coordinator carga las posiciones asignadas del último request procesado y continúa desde donde se quedó, evitando duplicación y procesamiento perdido.

El **WSM** actúa como la autoridad central de deduplicación para todos los workers replicados (Cleaners, Splitters, Groupers). A diferencia del Coordinator que gestiona su propio estado local, el WSM mantiene un *registro global persistente* de las posiciones procesadas por cada tipo de worker. Utiliza archivos **append** para registrar cada tupla $(request_id, position)$ confirmada. Antes de procesar cualquier mensaje, los workers consultan al WSM: si la posición ya existe en el registro, se descarta inmediatamente. Este diseño centralizado permite que múltiples réplicas compartan el estado de "procesado", evitando que un mensaje re-encolado por RabbitMQ sea ejecutado dos veces por distintas réplicas. En la figura de a continuación se ejemplifica el caso.



El **Joiner** implementa una estrategia de deduplicación híbrida. Para el flujo de entrada, delega la detección de duplicados al WSM, asegurando que cada fragmento de datos (Stores, Users, etc.) sea ingerido una única vez. Sin embargo, añade una capa adicional de seguridad mediante **escrituras atómicas** en sus archivos temporales y de salida. A continuación se va a explicar esta logica adicional de tolerancia a fallos.

6.5.4. Atomic Write

Atomic Write es otro de nuestros mecanismos de tolerancia a fallos, que garantiza la integridad de archivos en el sistema mediante una estrategia de escritura "todo-o-nada". El mecanismo funciona en tres fases: primero, crea un archivo temporal en el mismo directorio del destino usando `tempfile.mkstemp()`. Segundo, escribe completamente los datos al archivo temporal. Tercero, realiza un `rename()` atómico a nivel kernel del temporal al destino final. Si ocurre cualquier fallo en las fases uno o dos, el archivo temporal se descarta y el archivo destino nunca se modifica, garantizando que no existan archivos parciales o corruptos visibles a otros workers.

Este mecanismo nos permite volver a un estado anterior ante la caída inesperada de un nodo. La persistencia en disco del estado actual de cada nodo se utiliza extensamente en la arquitectura: en **Grouper_v2** para la escritura de resultados de agregación; en **Joiner_v2** para escribir archivos temporales de deduplicación y outputs joineados; en **Gateway** para guardar requests activas; en **Coordinator** para detectar deduplicación. Otros usos ocurren en **Splitter**, **Sort** y **Sorter_v2**.

También está integrado con el Worker State Manager (WSM), lo que permite recuperación por existencia de archivo. Si un worker crashea después de un atomic_write exitoso, al reiniciar se detecta el archivo y se evita reprocesamiento. Si crashea antes del rename, el archivo no existe y el WSM marca el estado como pendiente para reintentar.

6.6. Soporte multi-client y multi-request

6.6.1. Gestión de múltiples *requests*

El sistema posee la capacidad de gestionar múltiples *requests* secuenciales provenientes de un mismo cliente sin requerir reconexiones. El Gateway implementa un mecanismo de detección implícita de nuevos *requests* mediante el monitoreo continuo del flujo de datos asociado a cada conexión.

Cuando el Gateway recibe un mensaje de datos correspondiente a un tipo de archivo que ya había sido marcado como finalizado en el contexto actual, infiere automáticamente que ha comenzado un nuevo lote de procesamiento. En ese momento asigna un nuevo identificador de *request* de manera atómica y reinicia el estado interno asociado a la conexión. Este procedimiento permite mantener un flujo continuo y ordenado de lotes de procesamiento sin necesidad de delimitadores explícitos adicionales.

6.6.2. Enrutamiento de resultados

El retorno de los resultados procesados se organiza mediante una cola global de resultados, en la que los workers depositan sus respuestas finales. El Gateway ejecuta un componente dedicado que actúa como consumidor exclusivo de dicha cola y es responsable de reenviar cada resultado al cliente correspondiente.

Para ello, este componente inspecciona el encabezado del mensaje recibido, identifica el *request id* asociado y consulta la tabla interna de conexiones activas. Con esta información, reenvía el resultado al canal de comunicación del cliente que originó la solicitud. Esta arquitectura desacopla completamente el procesamiento distribuido del sistema de la gestión de las conexiones de red, dado que los workers solo deben propagar el *request id* correspondiente.

6.6.3. Protocolo y sincronización del cliente

El cliente opera mediante un modelo de procesamiento asíncrono que separa el envío de datos de la recepción de resultados. Mientras un hilo se encarga de transmitir los archivos de entrada al Gateway, otro permanece a la espera de recibir las respuestas asociadas.

Dado que los identificadores de *request* son asignados por el servidor en tiempo de ejecución, el cliente mantiene una estructura interna que asocia cada *request id* con la iteración de procesamiento iniciada localmente. Esto le permite organizar los resultados recibidos en directorios independientes y mantener la coherencia de los archivos generados, aun cuando el servidor gestione múltiples *requests* de manera intercalada o en paralelo.