Sistemas Distribuidos

Trabajo Práctico
Coffee Shop Analysis

Segundo cuatrimestre 2025

Siro Fatala 109.669	Martin Abramovich 108.762	Iara Jolodovsky 109.385

Introducción

El presente informe detalla la arquitectura, el diseño y las estrategias de implementación del Trabajo Práctico de Sistemas Distribuidos, cuyo objetivo es construir un sistema de análisis de datos para una cadena de cafeterías (Coffee Shop Analysis). Este sistema está diseñado para procesar grandes volúmenes de transacciones de múltiples clientes concurrentes de manera distribuida, eficiente y con alta tolerancia a fallos.

La solución se estructura como un pipeline de procesamiento de datos desacoplado, donde la comunicación central es gestionada por un Message Oriented Middleware (RabbitMQ). El diseño enfatiza la escalabilidad horizontal de sus componentes, permitiendo la ejecución paralela de múltiples instancias de workers de filtrado, agrupación y agregación. Se implementa un protocolo de finalización de flujo de datos (EOF) por sesión, asegurando que cada consulta de cliente sea procesada de forma completa y aislada.

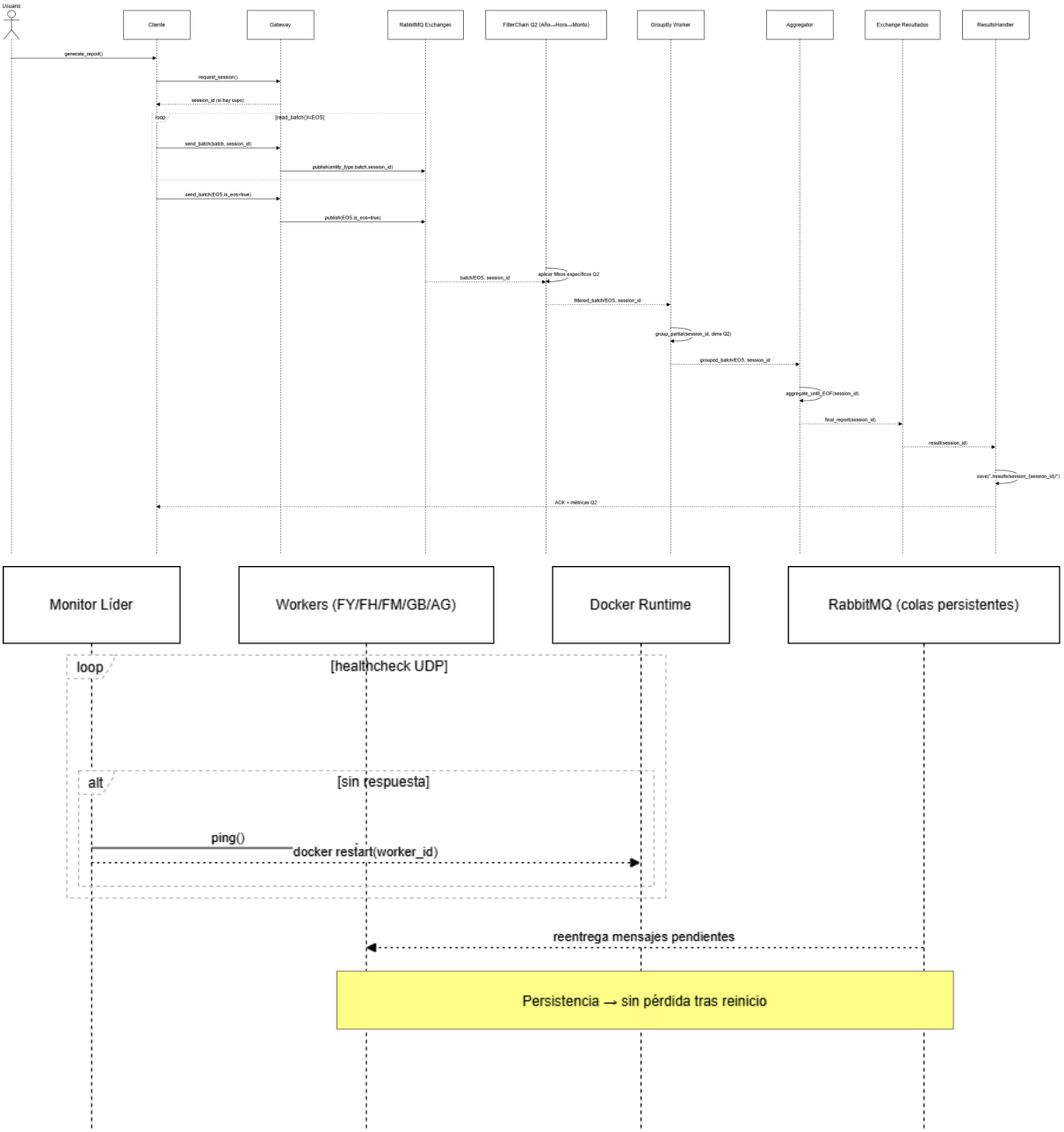
Un pilar fundamental del sistema es la robustez. La tolerancia a fallos se garantiza mediante la persistencia de mensajes, mecanismos de reconexión automática y un sistema de monitoreo redundante que utiliza el algoritmo Bully para la elección de líder. Este sistema de monitoreo es capaz de detectar la caída de nodos mediante healthchecks UDP y revivir automáticamente los contenedores fallidos utilizando Docker CLI. Adicionalmente, los nodos con estado (aggregators) implementan un sistema de persistencia basado en Write Ahead Logging para recuperarse sin pérdida de progreso.

A lo largo del documento, se describen las distintas vistas de la arquitectura (Lógica, de Proceso, de Desarrollo y Física) y se profundiza en las estrategias adoptadas para lograr alta disponibilidad, consistencia y rendimiento en el contexto de un sistema multicliente.

1. Vistas

1.1 Vista Lógica

Diagrama de Secuencia



El diagrama de secuencia describe el ciclo de vida completo de la generación de reportes del sistema multicliente y tolerante a fallos. El sistema admite hasta N clientes concurrentes (configurable, con un valor por defecto de 5), donde cada uno posee un session_id único (UUID) que identifica su sesión. Cuando se alcanza el límite máximo de conexiones simultáneas, el Gateway rechaza nuevas solicitudes para preservar la estabilidad del sistema.

El proceso inicia con la lectura y el envío de datos por parte de cada cliente. Estos leen los datos fuente en batches mediante la operación read_batch() y los envían al Gateway junto con su session_id. El Gateway cumple el rol de punto de entrada centralizado: valida los datos recibidos y los publica en los exchanges de RabbitMQ correspondientes según el tipo

de entidad, etiquetando cada mensaje con el session_id para mantener la independencia entre sesiones.

El procesamiento posterior se realiza de manera distribuida a través de una cadena de Filter Workers que intercambian mensajes mediante RabbitMQ. Cada worker aplica criterios de filtrado específicos según la query (por ejemplo, por año, por hora o por monto), y transmite los datos resultantes al siguiente worker en la cadena.

Una vez filtrados, los datos avanzan hacia el GroupBy Worker en los casos en que la query lo requiera. Este componente se encarga de realizar las agregaciones parciales necesarias según las dimensiones definidas (mes, producto, sucursal, cliente, entre otras). Los resultados intermedios se envían por RabbitMQ al Aggregator correspondiente, que es el responsable de la consolidación final.

Cada sesión mantiene un protocolo de finalización independiente basado en un mecanismo de EOF por sesión. Cuando un cliente deja de enviar datos de un tipo de entidad, transmite un batch vacío (EOS). El Gateway marca ese mensaje con is_eos=true y lo reenvía a los workers. Cada worker utiliza un componente llamado SessionTracker para registrar los batches recibidos por sesión y detectar con precisión cuándo una sesión ha completado su flujo de datos. No existe ninguna coordinación global entre sesiones; cada una se procesa de forma completamente aislada.

El Aggregator Worker combina todos los resultados parciales y construye la vista global correspondiente a la query solicitada. Para ello mantiene estado temporal por session_id hasta recibir la señal de finalización (EOF) proveniente de todas las fuentes involucradas en esa sesión. Una vez generado el resultado final, el aggregator lo publica en un exchange de resultados en RabbitMQ, utilizando nuevamente el session_id para permitir la recuperación específica del contenido por parte del cliente.

Luego, el ResultsHandler, que se ejecuta en el Gateway, consume los resultados publicados en RabbitMQ por los aggregators y los acumula en memoria. Cuando todas las queries de una sesión terminan, el ResultDispatcher coordina y empaqueta los resultados en un JSON que el Gateway envía al cliente por TCP. Finalmente, el cliente recibe el JSON y genera los archivos CSV localmente en la carpeta ./results/session_{session_id}/

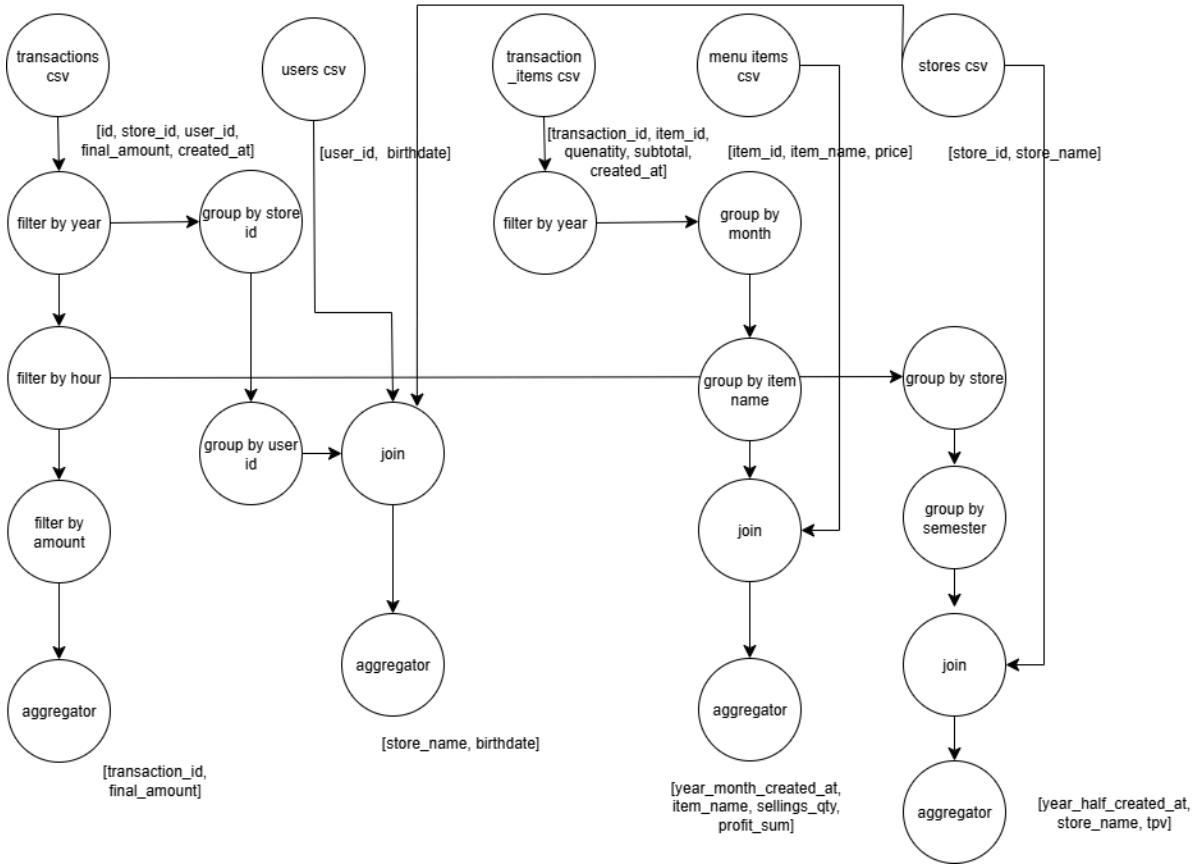
En cuanto a la tolerancia a fallos, el sistema cuenta con monitores redundantes que verifican periódicamente la salud de los nodos mediante healthchecks por UDP. En caso de que un worker falle, el Monitor Líder detecta el problema y relanza el contenedor utilizando Docker CLI. Gracias a la persistencia configurada en las colas de RabbitMQ, los mensajes no se pierden durante el fallo, y el worker recuperado puede continuar procesando desde donde dejó.

Los bucles visibles en el diagrama reflejan el procesamiento incremental mediante batches, lo que permite manejar grandes volúmenes de datos sin saturar la memoria. Asimismo, el diseño distribuido desacopla las etapas del procesamiento y las conecta mediante un middleware de mensajería, lo que garantiza escalabilidad —mediante la ejecución paralela

de múltiples instancias de cada worker— y flexibilidad, ya que cada query puede resolverse componiendo diferentes pipelines de procesamiento.

1.2 Vista de Proceso

DAG



El diagrama representa el flujo de datos a través de las distintas etapas de procesamiento que conforman las queries del sistema distribuido. Cada nodo corresponde a una operación realizada por un worker, y las aristas reflejan cómo la salida de un proceso alimenta al siguiente.

Los datasets se leen en batches y se envían a través del middleware hacia los workers correspondientes.

- Para la Query 1 (Transacciones filtradas por año, hora y monto ≥ 75), se utilizan los nodos filter by year, filter by hour y filter by amount. Los resultados se consolidan en el aggregator.

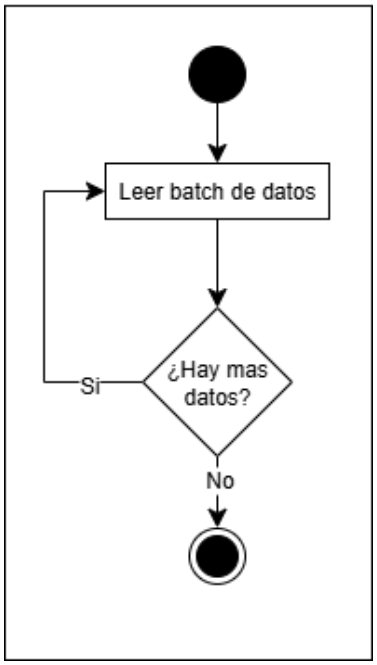
- Para la Query 2 (Productos más vendidos y más rentables), los datos de transaction_items se filtran por año, se realizan dos caminos de agrupamiento paralelos: group by month + group by item_name, y luego se unen con menu_items para obtener los nombres → aggregator (para cantidad y ganancias).

- Para la Query 3 (TPV por semestre y sucursal), los datos de transactions se filtran por año y hora, luego se aplican group by store y group by semester y se hace un join con stores, antes de enviar a su aggregator.

- Para la Query 4 (Top 3 clientes por sucursal), los transactions se filtran por año, se agrupan por store_id, luego por user_id, y finalmente se hace un join con la tabla de users para obtener la fecha de nacimiento. El aggregator selecciona los tres clientes con mayor cantidad de compras en cada sucursal.

Diagramas de Actividad

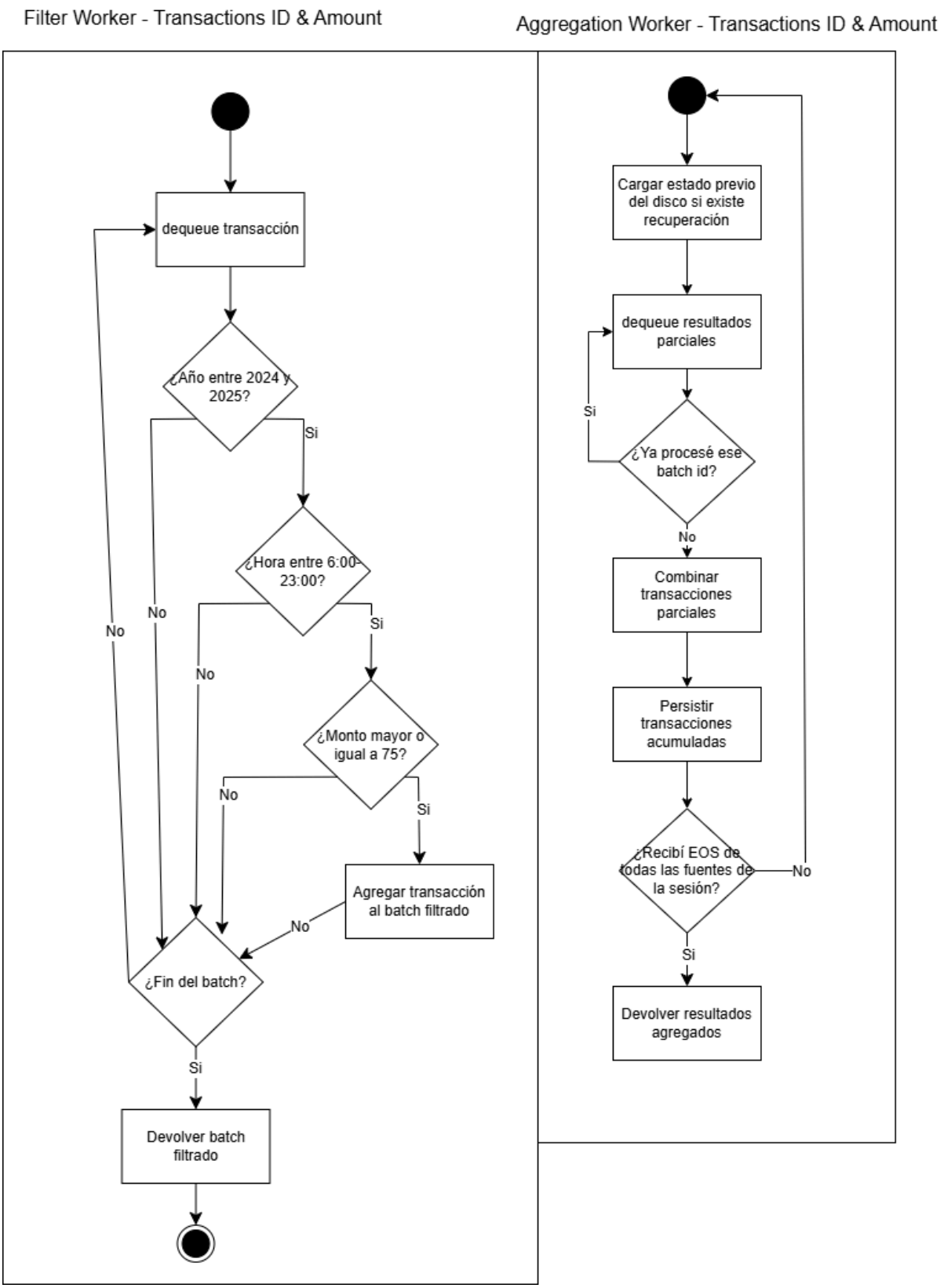
Lectura de datos



Este diagrama describe el proceso general de lectura de los archivos de entrada en el sistema.

* Se leen los datos en batches que son enviados al Gateway para su posterior procesamiento por los workers.

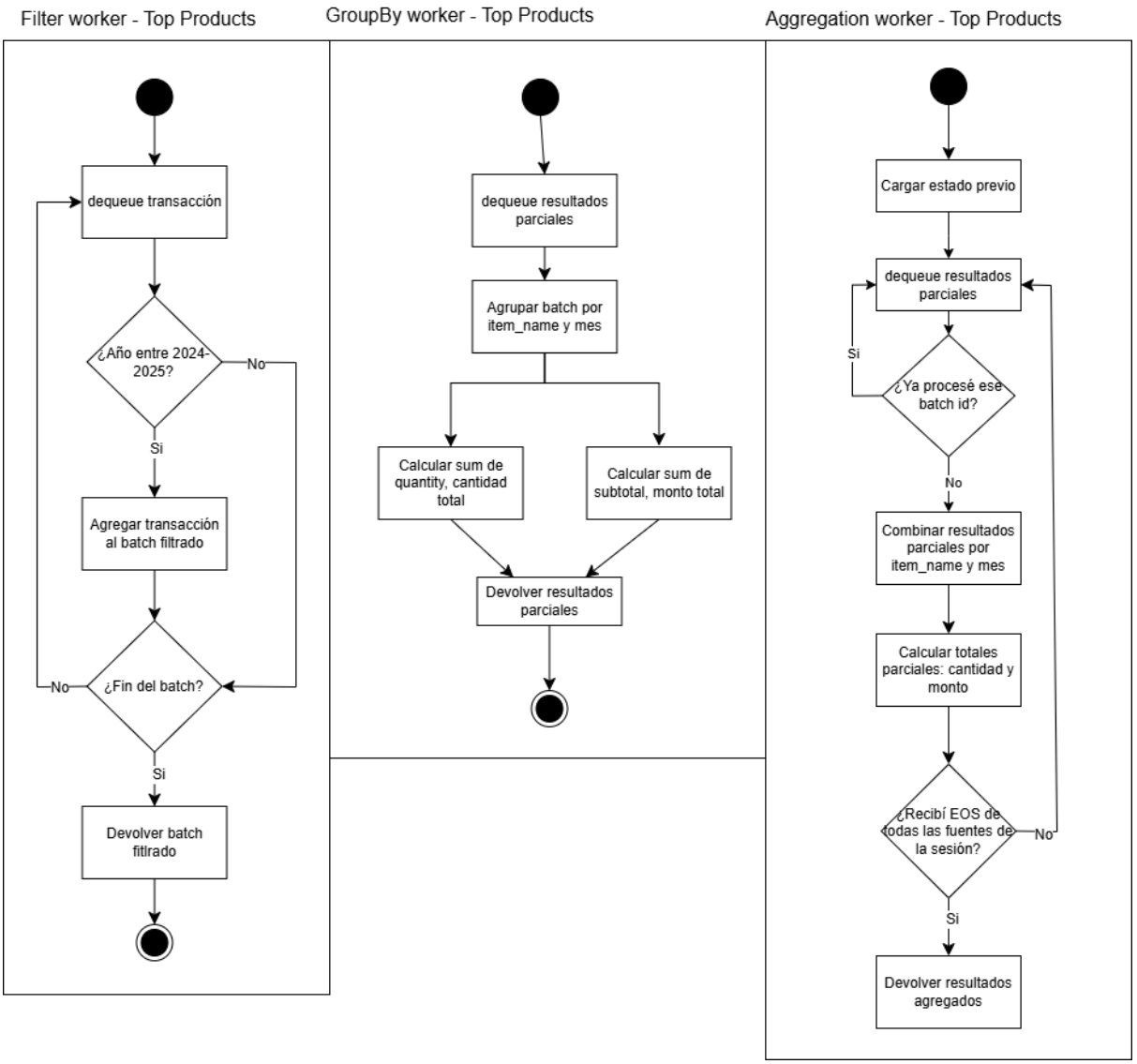
* **Cada cliente mantiene su propio contexto de sesión** con un `session_id` único.



Este diagrama muestra el pipeline para obtener los pares (transaction_id, amount) de todas las transacciones válidas dentro de una sesión. El Filter Worker recibe batches desde RabbitMQ, identifica la sesión correspondiente y filtra las transacciones que cumplen: año

2024–2025, hora entre 06:00 y 23:00, y monto ≥ 75. El resultado es un batch con solo las transacciones válidas para esa sesión.

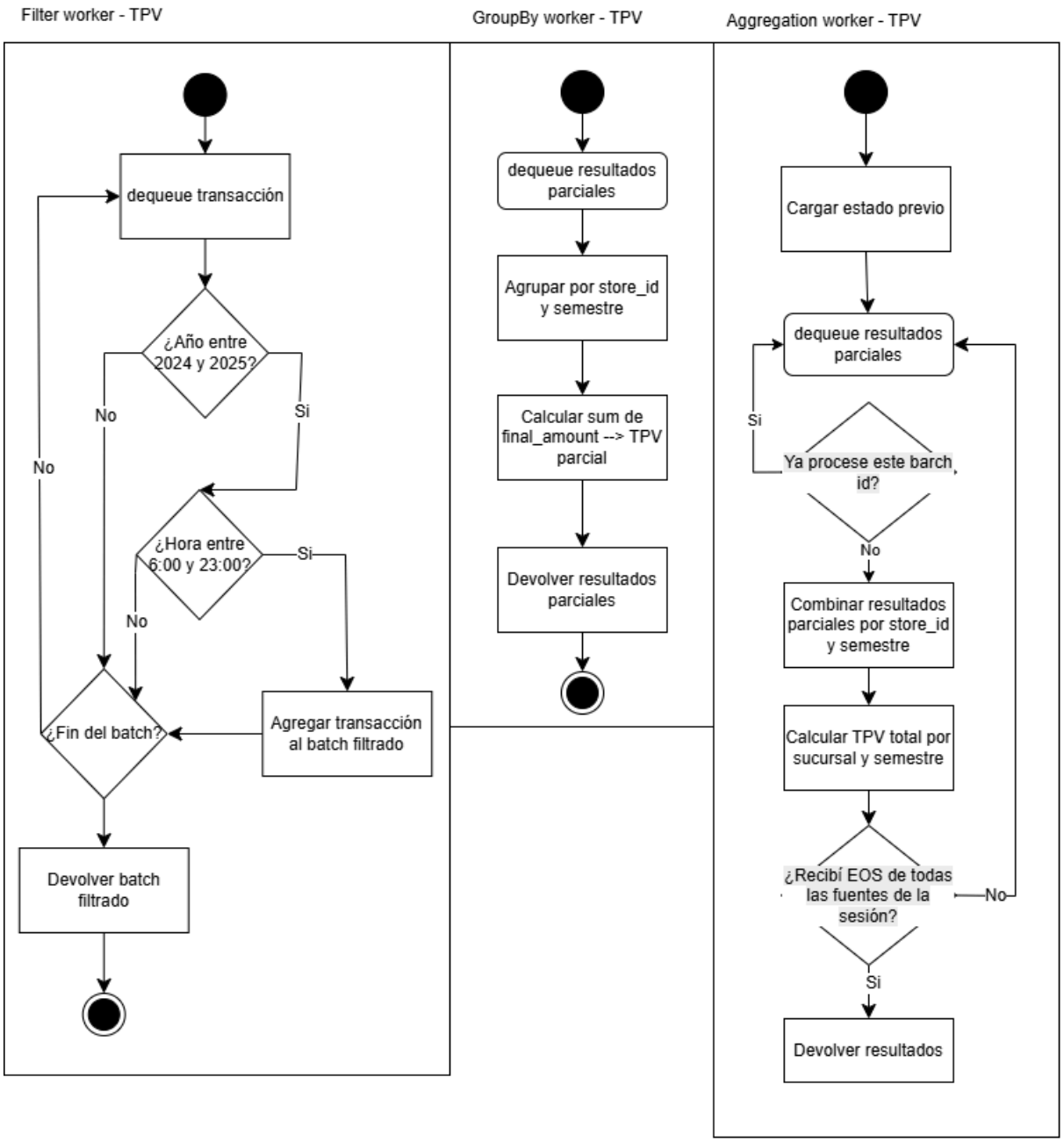
El Aggregation Worker recibe estos batches filtrados, cargando el estado previo si está en recuperación. Descarta batches ya procesados y combina los nuevos datos con el estado acumulado por session_id. Cuando llegan todos los EOF de esa sesión, el worker utiliza el estado final para producir el conjunto de pares (transaction_id, amount). Esto permite procesar múltiples sesiones en paralelo y recuperarse sin perder progreso.



Este diagrama describe la generación del ranking de productos más vendidos y con mayor facturación por mes para los años 2024–2025.

- El Filter Worker toma cada transacción del batch y conserva solo las que pertenecen a 2024 o 2025. Al finalizar el batch, devuelve el conjunto filtrado.
- El GroupBy Worker recibe esos datos, los agrupa por item_name y mes, y calcula totales parciales de cantidad y monto para cada grupo.
- El Aggregation Worker combina los resultados parciales, evitando reprocesar batches ya vistos. A medida que recibe todos los EOF de la sesión, acumula

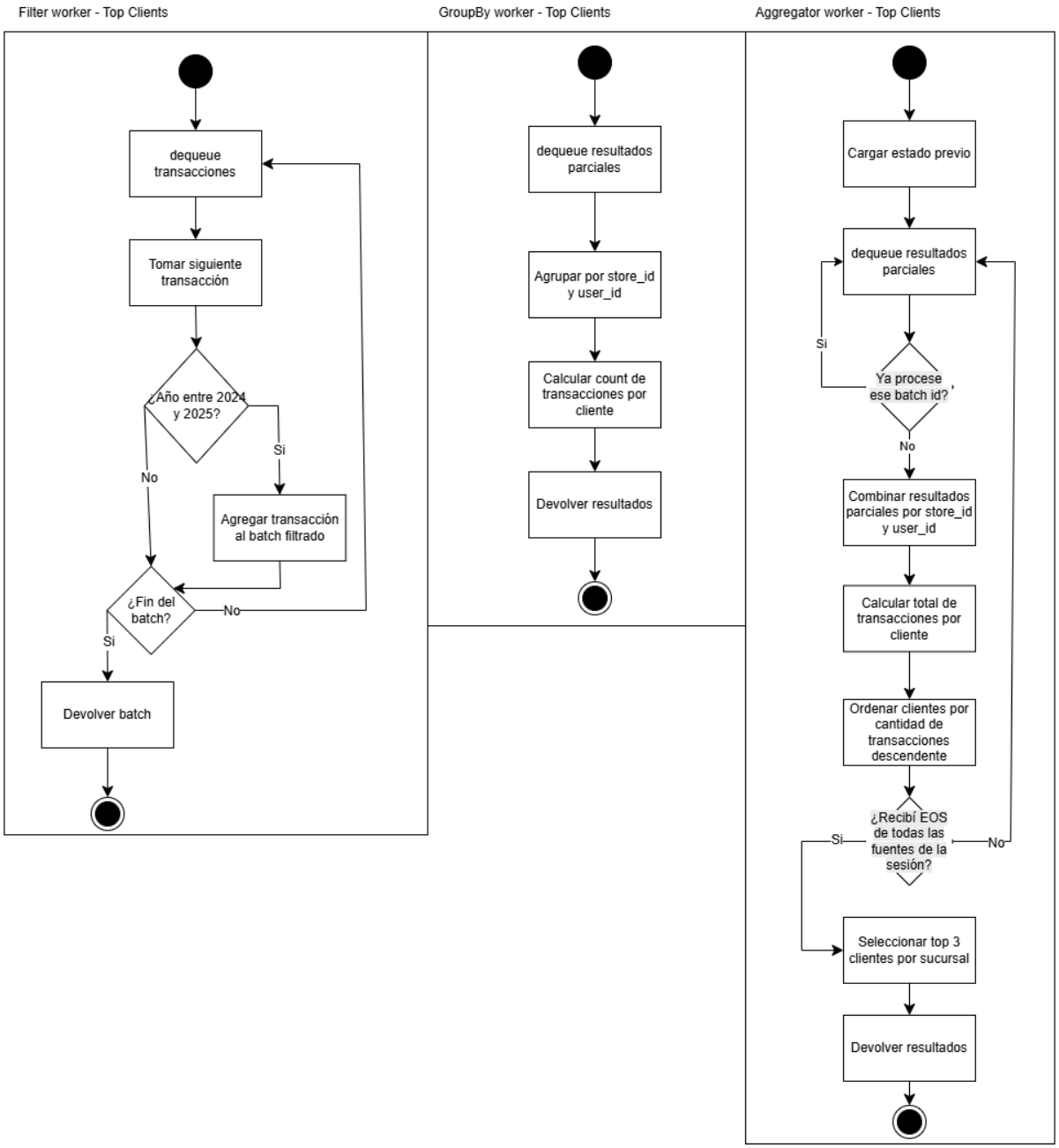
cantidades y montos por producto y mes. Finalmente devuelve los totales agregados.



Este diagrama representa la consulta que calcula el total de ventas por sucursal y semestre para los años 2024–2025, considerando solo transacciones realizadas entre las 6:00 y las 23:00.

- El Filter Worker selecciona únicamente las transacciones dentro del rango de años y horario permitido.
- El GroupBy Worker agrupa los datos válidos por store_id y semestre, calculando el TPV parcial a partir de la suma de final_amount.

- El Aggregator Worker combina los parciales recibidos, acumula el TPV total por sucursal y semestre y devuelve los resultados finales al completar la sesión.

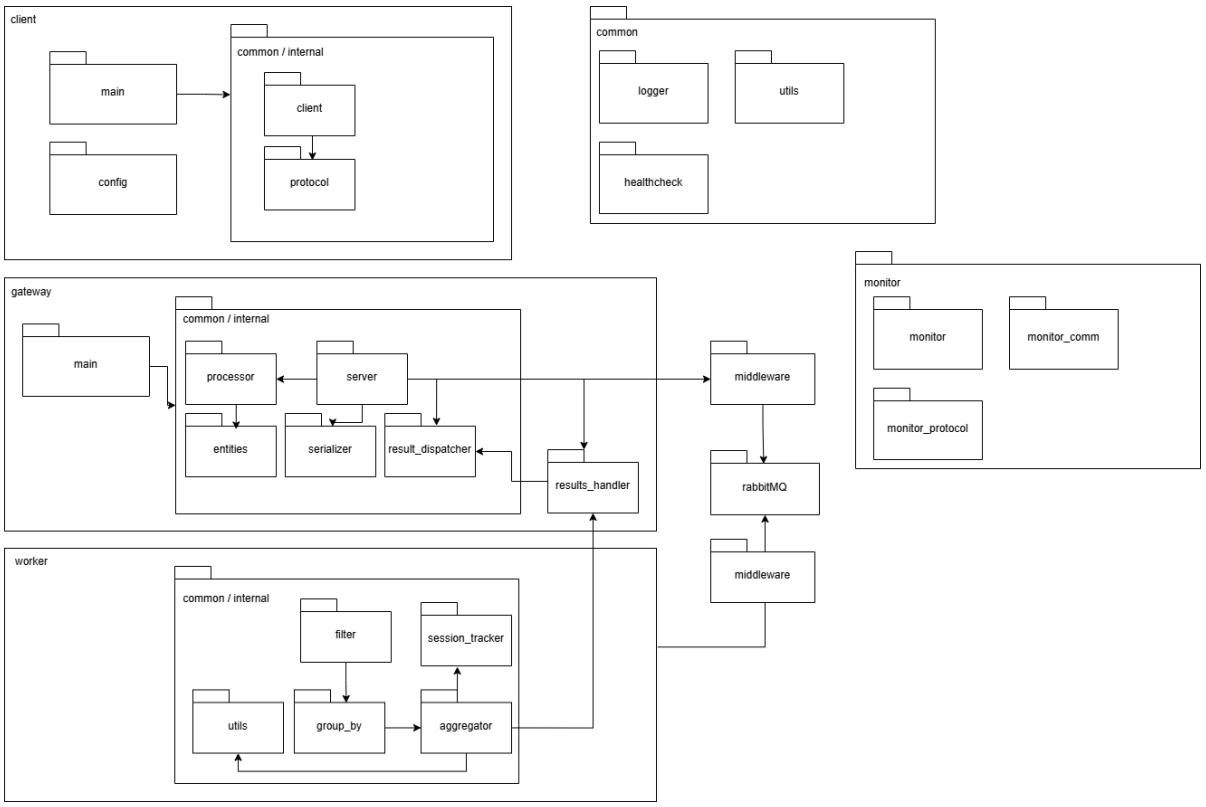


Este diagrama representa la consulta que obtiene los tres clientes con mayor cantidad de transacciones por sucursal para los años 2024–2025.

- El Filter Worker procesa cada lote y conserva solo las transacciones pertenecientes a los años requeridos.
- El GroupBy Worker agrupa los datos válidos por store_id y user_id, calculando el total parcial de transacciones por cliente.
- El Aggregator Worker combina los parciales, acumula el total por cliente y sucursal, ordena los resultados y selecciona el Top 3 por sucursal al finalizar la sesión.

1.3 Vista de Desarrollo

Diagrama de Paquetes



La capa Cliente reúne los módulos principales encargados de la configuración y la ejecución del sistema. El archivo main.py coordina la lectura y el envío de datos empleando múltiples threads —uno dedicado a la lectura de CSV y otro al envío— con el fin de maximizar el throughput. Dentro de la subcapa common se encuentran dos componentes esenciales: client.py, que implementa un cliente TCP con manejo de conexión, protocolo binario y detección de rechazos de conexión; y protocol.py, que define la codificación y decodificación de los mensajes binarios según la especificación del sistema. Una vez que el cliente recibe los resultados finales provenientes del gateway, los persiste localmente como archivos CSV en la ruta ./results/session_{session_id}/.

La capa Gateway centraliza la recepción de solicitudes provenientes de múltiples clientes concurrentes. Entre sus componentes principales se encuentra main.py, un servidor TCP multithreaded capaz de atender hasta N clientes simultáneos (configurable, con un valor por defecto de 5). El módulo server.py administra las sesiones de cada cliente utilizando un session_id único e implementa el protocolo EOF por sesión, permitiendo que cada flujo de datos finalice de manera independiente. Por su parte, processor.py se encarga de validar y procesar los batches recibidos; serializer.py gestiona la serialización de mensajes hacia RabbitMQ incorporando el correspondiente session_id; y result_dispatcher.py organiza la recepción y entrega de resultados por sesión, coordinando múltiples queries si es necesario. Finalmente, results_handler.py consume los resultados publicados en RabbitMQ, los

acumula por sesión mediante el componente SessionStateManager —que persiste el estado parcial en ./state/— y los envía al cliente una vez completo el procesamiento.

En la capa Workers residen los módulos encargados del procesamiento distribuido. Esta capa incluye los workers de filtrado (filter_year, filter_hour, filter_amount), que aplican criterios específicos; los workers de agrupación (group_by_query2, group_by_query3, group_by_query4), responsables de realizar las agregaciones parciales; y los aggregators para las distintas queries (aggregator_query1-4), que mantienen un estado interno por session_id hasta recibir la señal de EOF para todas las fuentes correspondientes a esa sesión. Además, el módulo session_tracker.py registra los batches recibidos por sesión y por tipo de entidad con el objetivo de detectar la completitud del flujo de datos, mientras que utils.py aporta funciones auxiliares compartidas entre los distintos procesos. Todos los workers incorporan lógica de reconexión automática para restablecer la comunicación con RabbitMQ en caso de fallos.

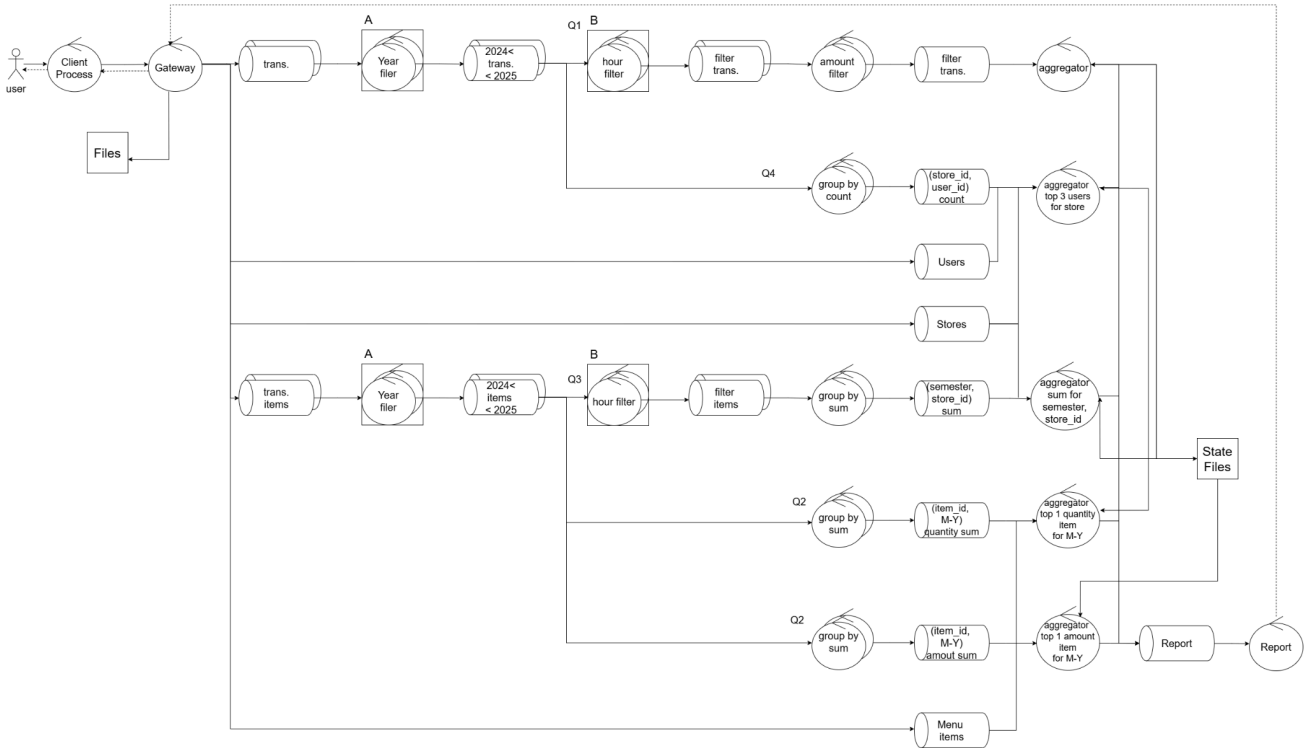
La capa Monitor implementa un sistema de monitoreo redundante orientado a proporcionar tolerancia a fallos. El módulo monitor.py aplica el algoritmo Bully para la elección de líder y se encarga de verificar el estado de salud de los nodos mediante healthchecks UDP, además de revivir contenedores caídos utilizando Docker CLI. Los módulos monitor_comm.py y monitor_protocol.py complementan este funcionamiento mediante la comunicación TCP entre monitores y la definición del protocolo de mensajes para intercambio de heartbeats, elección de líder y sincronización del estado.

El middleware encapsula la lógica común de interacción con RabbitMQ y proporciona una abstracción unificada a través de middleware.py, que define MessageMiddlewareQueue para colas y MessageMiddlewareExchange para exchanges. Este componente incluye mecanismos de reconexión automática ante fallos y soporta persistencia de mensajes mediante durable queues, garantizando así la no pérdida de información durante fallas temporales en los nodos.

Finalmente, la capa Common agrupa componentes compartidos por todo el sistema, entre ellos healthcheck.py, que implementa un servidor UDP utilizado por cada nodo (gateway y workers) para reportar su estado de salud, y logger.py, que centraliza el sistema de logging. En conjunto, el diagrama de paquetes refleja un diseño donde el cliente produce los datos, el gateway centraliza y organiza la comunicación multicliente, los workers ejecutan el procesamiento paralelo sobre los mensajes provenientes de RabbitMQ, el middleware estandariza la interacción con la infraestructura de mensajería y el sistema de monitores asegura tolerancia a fallos mediante detección temprana y recuperación automática de nodos.

1.4 Vista de Física

Diagrama de Robustez



Este diagrama ilustra el flujo completo de procesamiento de datos transaccionales destinado a la generación de reportes en un sistema multicliente y tolerante a fallos. El proceso comienza con la entrada de datos por parte de múltiples clientes (Client1, Client2, ..., ClientN), cada uno de los cuales carga archivos CSV en batches y mantiene su propio contexto de sesión identificado mediante un session_id único. Los datos son enviados al Gateway a través de conexiones TCP en el puerto 9000. Este componente controla la cantidad máxima de clientes concurrentes —configurable, con un valor por defecto de cinco— y rechaza nuevas conexiones cuando se alcanza dicho límite.

El Gateway funciona como punto de entrada centralizado del sistema. Gestiona conexiones concurrentes mediante multithreading, valida y procesa los batches recibidos y publica los mensajes resultantes en RabbitMQ según el tipo de entidad, siempre etiquetándolos con el session_id correspondiente para mantener la independencia entre sesiones. También implementa un protocolo de finalización EOF por sesión, permitiendo que cada cliente marque de forma independiente el fin de su flujo de datos. Además, consume los resultados generados por la capa de procesamiento distribuido a través del exchange de RabbitMQ y los almacena localmente en formato CSV.

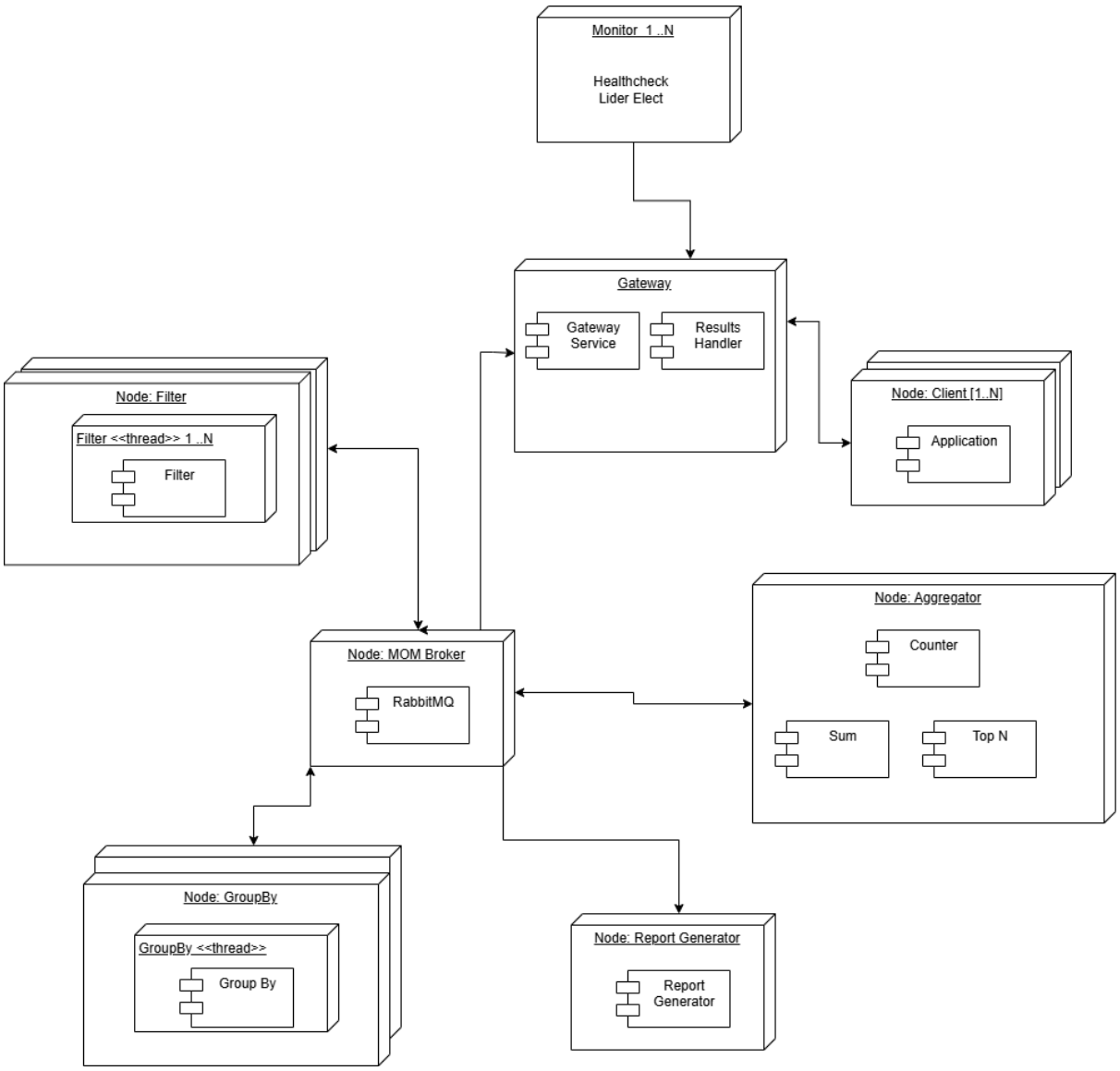
El Middleware de Mensajería, basado en RabbitMQ, constituye la columna vertebral del sistema distribuido. Este componente utiliza exchanges y colas persistentes (durable) para garantizar que los mensajes no se pierdan en caso de fallos. Todos los mensajes están etiquetados con el session_id, lo que habilita el procesamiento concurrente y aislado por sesión. Asimismo, múltiples instancias de workers pueden consumir de las mismas colas, permitiendo escalabilidad horizontal cuando aumenta la carga de trabajo.

La etapa de filtrado se lleva a cabo mediante múltiples nodos Filter Workers, que aplican las condiciones definidas por la consulta correspondiente. Cada worker puede desplegar múltiples instancias para operar en paralelo, y todos cuentan con lógica de reconexión automática ante fallos de conexión con RabbitMQ. Posteriormente, la etapa de agrupación es ejecutada por los GroupBy Workers, responsables de realizar operaciones de agregación —como sumas o conteos— en campos específicos, lo que contribuye significativamente a reducir el volumen de datos. Una vez completadas estas operaciones, los Aggregator Workers combinan los resultados parciales; para ello mantienen estado interno por session_id hasta recibir la señal de EOF de todas las fuentes asociadas a esa sesión. Al finalizar, generan los resultados definitivos y los publican en el exchange destinado a la salida.

En lo referente al monitoreo y la tolerancia a fallos, el sistema incorpora tres monitores redundantes que verifican continuamente el estado de todos los nodos mediante healthchecks UDP (puerto 8888). Estos monitores implementan el algoritmo Bully para la elección del líder, de modo que, si el monitor principal falla, otro asume automáticamente el rol. El Monitor Líder es responsable de detectar la caída de nodos y revivir los contenedores afectados mediante Docker-in-Docker.

El procedimiento de recovery automático es fundamental para garantizar la continuidad del servicio. Cuando un worker falla, el monitor detecta la ausencia de respuesta a los healthchecks después de varios intentos fallidos. El Monitor Líder identifica el contenedor Docker correspondiente y ejecuta el comando docker start <container_id> para reiniciarlo. Una vez restaurado, el worker se reconecta automáticamente a RabbitMQ y continúa su procesamiento. Gracias a la persistencia de las colas, los mensajes pendientes permanecen disponibles y pueden ser procesados sin pérdida alguna tras la recuperación del nodo.

Diagrama de Despliegue

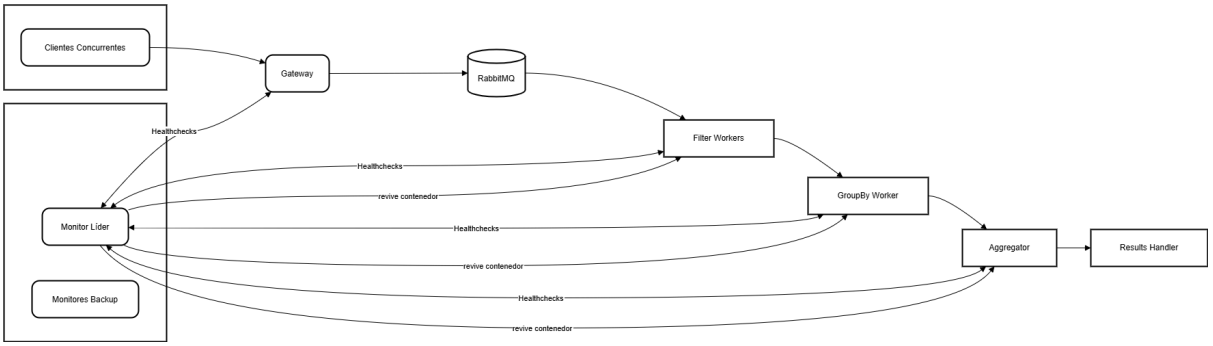


El sistema se compone de varios nodos distribuidos que cooperan mediante un middleware de mensajería para procesar grandes volúmenes de información transaccional, optimizando el rendimiento en entornos multicomputadoras.
Los principales nodos son:

- Cliente: ejecuta la aplicación que inicia el proceso de generación de reportes, enviando lotes de transacciones al sistema.
- Gateway: punto de entrada al sistema distribuido. Recibe los batches de datos del cliente y los publica en el middleware para su posterior procesamiento.
- Middleware: actúa como Message Oriented Middleware (MOM), implementando la comunicación entre los distintos nodos. Permite desacoplar y escalar el procesamiento.
- Filter: conjunto de instancias paralelas que eliminan las transacciones que no cumplen con los criterios de filtrado.
- GroupBy: agrupa las transacciones filtradas por criterios como sucursal o producto, preparando los datos para su posterior agregación.

- Aggregator: calcula métricas consolidadas. Está compuesto por submódulos especializados: Counter, TopN y Sum.
- Report Generator: genera los reportes finales a partir de los resultados agregados, que luego son devueltos al cliente.
- Monitor: Nodo independiente para healthcheck UDP de todos los componentes, detección de fallos y elección de líder (algoritmo Bully).

2. Tolerancia a Fallos



2.1 Estrategias Implementadas

El sistema incorpora diversas estrategias orientadas a garantizar tolerancia a fallos y continuidad operativa incluso bajo condiciones adversas. En primer lugar, se asegura la persistencia de mensajes en RabbitMQ, configurando todas las colas y exchanges como durable=True y marcando los mensajes con delivery_mode=2, lo que los vuelve persistentes. Gracias a esto, los mensajes no se pierden en caso de que un worker falle antes de procesarlos; una vez recuperado, el worker puede retomar la ejecución y continuar procesando los mensajes pendientes sin inconsistencias.

Otra capa fundamental de resiliencia es la reconexión automática. Tanto el Gateway como los workers implementan mecanismos de reconexión a RabbitMQ con backoff exponencial ante la pérdida de conectividad. Este comportamiento permite que los componentes se recuperen de fallos temporales de red o reinicios del broker sin intervención manual y sin pérdida de datos.

El sistema también incluye monitores redundantes encargados de supervisar la salud de todos los nodos. Se despliegan tres monitores que implementan el algoritmo Bully para la elección automática de un líder. Solo el monitor líder ejecuta acciones de recuperación, pero, si este falla, otro monitor asume el rol de forma inmediata. La supervisión se realiza mediante healthchecks UDP enviados cada cinco segundos a todos los nodos.

Los healthchecks UDP forman parte esencial del mecanismo de detección de fallos. Cada nodo ejecuta un servidor UDP en el puerto 8888 que responde “OK” ante mensajes de tipo “PING”. Un fallo se detecta cuando un monitor acumula tres intentos consecutivos sin respuesta por parte del nodo supervisado.

Una vez confirmada la caída de un nodo, el sistema ejecuta un proceso de revival automático de contenedores. El monitor líder, que posee acceso al socket Docker del host,

identifica el contenedor afectado y ejecuta docker start <container_id> para reiniciarlo. Para evitar bucles de fallos y reinicios incontrolados, este proceso cuenta con límites en la cantidad de intentos permitidos. Además, los contenedores están etiquetados con role=node, lo que simplifica su identificación.

La tolerancia a fallos también se aborda mediante una correcta gestión de sesiones. Cada cliente opera con un session_id único (UUID), y el Gateway mantiene un seguimiento detallado de las sesiones activas, liberando recursos al finalizar cada una. Del mismo modo, los workers almacenan estado temporal por sesión y eliminan automáticamente aquellas que permanezcan inactivas durante un tiempo prolongado, evitando acumulaciones innecesarias de estado y contribuyendo a la estabilidad del sistema.

Persistencia de estado en aggregators

Los aggregators son nodos stateful que acumulan datos parciales de múltiples batches antes de generar resultados finales. Para garantizar tolerancia a fallos implementan un sistema de persistencia inspirado en Write Ahead Logging.

El sistema usa dos estrategias según las características de los datos:

1. Persistencia Atómica (Replace): Para datos pequeños que se actualizan completos (ej: catálogos de stores, menu_items, users). Se usa el patrón temp + fsync + rename atómico. Esto garantiza que el archivo final siempre esté en estado consistente, nunca a medio escribir
2. Persistencia Incremental: Para grandes volúmenes de datos transaccionales se usa un patrón WAL con dos archivos:
    a. data.pkl: archivo append only donde se apilan chunks de datos con pickle
    b. info.pkl: metadata que registra hasta que byte de data.pkl es válido, actualizado automáticamente

info.pkl actúa como commit log: solo cuando se actualiza exitosamente, los datos en data.pkl se consideran válidos. Si el proceso se interrumpe entre ambas escrituras, la recuperación automática descarta los bytes huérfanos en data.pkl que excedan el tamaño registrado en info.pkl

Al iniciar, cada aggregator escanea el directorio de estado, valida la integridad de los archivos, reconstruye el estado en memoria y carga el tracker de batches procesados. Esto permite que un aggregator que falle (crash, reinicio, kill) se recupere completamente sin intervención manual.

El SessionTracker registra los batches procesados. Antes de procesar cualquier batch, se verifica si ya fue procesado, descartando duplicados y garantizando exactly-once semantics.

2.2 Protocolo EOF por Sesión

El sistema implementa un protocolo de finalización independiente por sesión.

1. Cliente envía EOS:

    - Cuando termina de leer datos de un tipo de entidad, el cliente envía un batch vacío
    - El batch viaja al Gateway igual que los demás
2. Gateway procesa EOS:
    - Detecta EOS cuando el batch tiene items vacíos
    - Serializa el mensaje marcando is_eos=true
    - Lo envía inmediatamente a los workers
    - Si el cliente se desconecta antes de enviar EOS, el Gateway genera EOS automáticamente para cada entidad procesada
3. Workers procesan EOS por sesión:
    - Reciben mensajes marcados con is_eos=true
    - Usan SessionTracker para registrar el avance por session_id
    - Cuando todos los tipos de entidad requeridos de una sesión están completos, el aggregator genera los resultados finales

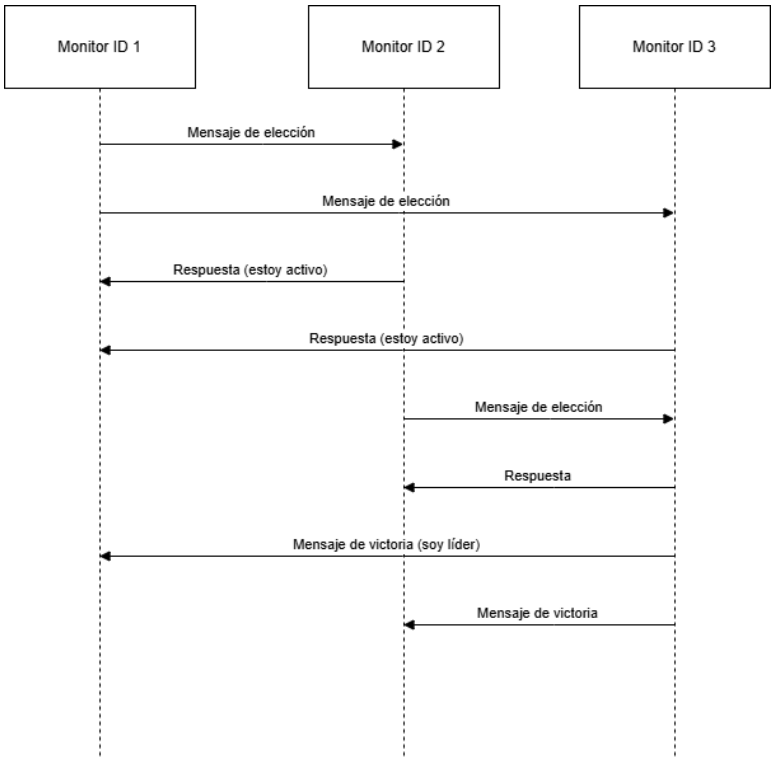
Hay independencia total entre sesiones, no existe coordinación entre clientes y cada aggregator mantiene un SessionTracker individual

2.3 Límite de Clientes Simultáneos

El Gateway controla la concurrencia para evitar sobrecargas.

Se usa un BoundedSemaphore configurable, por defecto 5. Cuando se alcanza el límite, nuevas conexiones se rechazan. El cliente recibe un error y se cierra la conexión adecuadamente. Esto previene la sobrecarga y garantiza calidad en el servicio.

3. Arquitectura de Monitoreo



3.1 Algoritmo Bully para Elección de Líder

El sistema de monitores implementa el algoritmo Bully para garantizar que siempre exista un único líder.

Funcionamiento:

1. Cada monitor tiene un ID único (configurable con MONITOR_ID).
2. Al iniciar, cada monitor envía un mensaje de elección a todos los monitores con ID mayor.
3. Si no recibe respuesta, se declara líder.
4. Si recibe respuesta, espera un mensaje de victoria del monitor con mayor ID.
5. El monitor con mayor ID se convierte en líder.

Los monitores se comunican mediante TCP en el puerto 9999. El líder envía heartbeats periódicos a los followers. Si los followers no reciben heartbeats dentro del tiempo configurado (LEADER_TIMEOUT), inician una nueva elección.

3.2 Healthchecks y Detección de Fallos

Cada nodo expone un servidor UDP en el puerto 8888. Responde "OK" a los mensajes "PING". El timeout es configurable, con un valor por defecto de dos segundos.

Detección de fallos: Los monitores envían healthchecks cada CHECK_INTERVAL segundos (por defecto cinco). Si un nodo no responde después de MAX_FAILED_ATTEMPTS intentos (por defecto tres), se marca como caído. El monitor líder intenta revivir el contenedor inmediatamente.

3.3 Revival de Contenedores

Proceso de revival:

1. El monitor detecta que un nodo no responde.
2. El monitor identifica el container ID del nodo mediante Docker CLI.
3. El monitor ejecuta docker start <container_id>.
4. Si el revival es exitoso, el nodo se marca como recuperado.
5. Si falla, se incrementa el contador de intentos hasta el máximo permitido.

Existe un máximo de intentos de revival por nodo para evitar loops infinitos. Tambien backoff entre intentos para evitar sobrecargar el sistema y logging detallado de cada intento de revival.

4. Consideraciones de Diseño

4.1 Escalabilidad

El sistema está preparado para escalar horizontalmente de manera eficiente. Cada tipo de worker puede ejecutarse en múltiples instancias simultáneas, lo que permite distribuir la carga de trabajo sin generar cuellos de botella. RabbitMQ se encarga de distribuir los

mensajes entre las instancias mediante un esquema round-robin, equilibrando automáticamente el procesamiento. El Gateway también admite múltiples clientes concurrentes, hasta un límite configurable según la capacidad del entorno. Del mismo modo, los monitores pueden incrementarse en número si fuera necesario, aunque un conjunto de tres instancias proporciona suficiente redundancia para un funcionamiento estable.

4.2 Consistencia

La consistencia del sistema se garantiza a través de varios mecanismos complementarios. El protocolo EOF por sesión asegura que cada consulta se procese como una unidad completa y aislada, evitando mezclas entre sesiones. Los aggregators mantienen estado por sesión hasta asegurar su completitud, lo que permite consolidar resultados sin pérdida de información. RabbitMQ utiliza persistencia de mensajes para evitar la pérdida de datos ante fallos temporales, mientras que el seguimiento de batches por sesión permite detectar con precisión cuándo el procesamiento ha finalizado completamente.

4.3 Disponibilidad

El diseño incorpora múltiples medidas para asegurar alta disponibilidad. Los monitores operan en un esquema redundante con tres instancias que seleccionan automáticamente a un líder, garantizando continuidad ante fallos. Los nodos cuentan con mecanismos de recuperación automática que permiten restablecer rápidamente componentes caídos. Además, todas las conexiones a RabbitMQ implementan reconexión automática para manejar interrupciones transitorias. El sistema ejecuta healthchecks periódicos que permiten identificar problemas antes de que afecten al funcionamiento general.

4.4 Performance

Se han aplicado diversas optimizaciones para mejorar el rendimiento del sistema. El Gateway implementa buffering de mensajes antes de enviarlos a RabbitMQ, disminuyendo el overhead de comunicación. El procesamiento se realiza en batches, reduciendo significativamente la cantidad de mensajes individuales que deben atravesar la infraestructura. En el cliente, el uso de threads permite leer y enviar datos en paralelo, aprovechando mejor los recursos disponibles. Por último, cada thread del Gateway utiliza conexiones exclusivas, lo que evita la contención y mejora la eficiencia del procesamiento concurrente.