.UBAfiuba
FACULTAD DE INGENIERÍA

Sistemas Distribuidos 1
(TA050) Roca

Diseño
Coffee Shop Analysis

Fecha: 4 de diciembre de 2025

Ascencio Felipe Santino - 110675

Gamberale Luciano Martín - 105892

Zielonka Axel - 110310

Índice

1. Introducción

El presente documento describe el diseño de un sistema distribuido para el análisis de datos de una cadena de cafeterías en Malasia. El objetivo principal es procesar grandes volúmenes de información transaccional, de clientes, de sucursales y de productos, para obtener métricas clave que apoyen la toma de decisiones de negocio.

De acuerdo con los requisitos existentes, el sistema debe permitir responder las siguientes consultas:

1. Listado de transacciones (ID y monto) realizadas entre los años 2024 y 2025, en el rango horario de 06:00 a 23:00, con un monto total mayor o igual a 75.
2. Identificación de los productos más vendidos (nombre y cantidad) y los que más ganancias han generado (nombre y monto) para cada mes de 2024 y 2025.
3. Cálculo del *Total Payment Value* (TPV) por cada semestre en 2024 y 2025, discriminado por sucursal, considerando sólo transacciones entre 06:00 y 23:00.
4. Obtención de la fecha de cumpleaños de los tres clientes con mayor cantidad de compras en 2024 y 2025, para cada sucursal.

Además de los requerimientos funcionales, el sistema deberá cumplir con los siguientes requisitos no funcionales:

- Optimización para entornos multicomputadoras, asegurando la escalabilidad ante el crecimiento de datos.
- Inclusión de un *middleware* que abstraiga la comunicación entre nodos mediante grupos.
- Ejecución única del procesamiento con capacidad de *graceful quit* frente a señales de terminación (SIGTERM).

El diseño se realizará bajo un enfoque de arquitectura distribuida, buscando flexibilidad, robustez y capacidad de escalar en entornos reales de procesamiento de datos.

2. Definición Técnica del Sistema Distribuido

El sistema distribuido propuesto permite al usuario interactuar mediante una consola, desde la cual los usuarios envían las instrucciones y datasets necesarios para el procesamiento, y a cambio reciben como salida de la misma el resultado de las consultas realizadas. La infraestructura se implementará sobre contenedores Docker, los cuales emulan múltiples nodos computacionales. Cada nodo se especializa en la ejecución de una operación determinada, tales como filtrado, mapeo, reducción, ordenamiento o combinación de datos.

El sistema recibe como entrada un conjunto de archivos con la información a procesar:

- `menu_items.csv`
- `payment_methods.csv`
- `stores.csv`
- Múltiples archivos `transaction_items.csv`
- Múltiples archivos `transactions.csv`
- Múltiples archivos `users.csv`
- `vouchers.csv`

Estos archivos son distribuidos entre los distintos nodos según lo que requiera cada operación. La comunicación entre el nodo coordinador y los nodos de procesamiento se gestiona mediante un *middleware*, el cual se describe en detalle en una sección posterior.

2.1. Operaciones Distribuidas

Las operaciones fundamentales soportadas por el sistema son las siguientes:

1. **Cleaner**: Se encarga de la limpieza de los datos de entrada, eliminando campos que resultan innecesarios para las consultas. Es escalable.
2. **Filter**: Permite seleccionar subconjuntos de datos en función de condiciones específicas. Una misma instancia puede aplicar diferentes filtros según el criterio definido en la consulta (por ejemplo, filtrar por fechas, montos o rangos horarios). Es escalable.
3. **Map**: Transforma los datos de entrada generando nuevas columnas o reformateando la información existente. Ejemplo: Agregar un campo con el mes y año a partir de una fecha, o calcular un contador auxiliar. Es escalable.
4. **Reduce**: Agrupa datos por una clave determinada y aplica una función de agregación (como sumas, conteos o acumulaciones). Existen múltiples variantes, dependiendo de qué métrica se busque consolidar. No es escalable.
5. **Join**: Combina registros de diferentes datasets en función de un campo común, permitiendo enriquecer la información (ejemplo: Unir transacciones con usuarios para obtener la fecha de nacimiento del cliente). Es escalable.
6. **SortBy**: Ordena los registros de acuerdo con uno o más criterios, ya sea de manera ascendente o descendente. Es fundamental para identificar los elementos más significativos en cada consulta (por ejemplo, productos más vendidos). No es escalable.
7. **OutputBuilder**: Es el último paso en la cadena de operaciones, se encarga de formatear la respuesta que va a ser enviada al usuario. Es escalable.

Cada nodo de operación está diseñado para ser genérico y flexible: Puede ejecutar cualquier variante de las funciones mencionadas según el rol específico del nodo, siendo el sistema coordinador quien le envía los datos y la configuración necesaria para que realice la tarea solicitada de la mejor forma posible.

3. Implementación del Usuario

La interacción de los usuarios con el sistema distribuido se realizará mediante un esquema secuencial. Para cada usuario que se conecte, se respetera una secuencia específica de ejecución.

- **Etapa de envío de información:** En esta etapa el usuario envía toda la información en base a la que el sistema debe operar. Esto incluye todos los datasets a procesar para resolver las 4 consultas.
- **Etapa de espera:** El cliente queda a la espera de que el sistema distribuido procese la información y genere las respuestas a las consultas.
- **Etapa de recepción de resultados:** Una vez que el sistema ha procesado la información, el usuario recibe las respuestas a las consultas solicitadas mediante unos archivos generados en la carpeta `'.results/query_results'` en formato '.txt'. Se genera un archivo por cada consulta.

De esta manera se logra una interacción segura y escalable.

La arquitectura está diseñada para soportar múltiples usuarios en simultáneo, escalando el número de conexiones establecidas proporcionalmente a la cantidad de clientes activos.

4. Mecanismo de Optimización del Uso de Recursos

Cada computadora que participa como nodo de procesamiento en el sistema distribuido estará dedicada a ejecutar una operación determinada (por ejemplo, filtrado o reducción).

Si bien en una primera versión se teorizó la posibilidad de generar varios procesos por 'CPU' para buscar algún tipo de optimización del rendimiento de cada computadora, luego de la primera entrega con el corrector se definió que no resultaría necesario para la implementación de la resolución.

Por lo tanto, se descarta esta mejora para esta versión del sistema, pero se deja como recomendación por si se realiza una actualización a futuro en búsqueda de optimizar tiempos de procesamiento de consultas.

De ser así, se debe recordar que al implementar la paralelización, se debe hacer mediante **multiprocesos** y no mediante **multithreading**. La justificación de esta elección radica en que Python presenta limitaciones para tareas de cómputo intensivo debido al *Global Interpreter Lock* (GIL).

La librería de `multiprocessing` permite crear procesos independientes que aprovechan mejor las capacidades de CPUs con múltiples núcleos.

5. Uso de Archivos Intermedios

En el diseño del sistema se contempla la generación de archivos intermedios durante el procesamiento de las consultas. Esta decisión se fundamenta en un aspectos clave:

Respaldo de información: Los archivos intermedios actúan como puntos de control que facilitan las implementaciones realizadas para la tolerancia a fallas. En caso de interrupciones, hacen posible retomar el procesamiento desde un estado parcial previamente almacenado.

El uso de archivos intermedios, si bien introduce un costo adicional de I/O, aumenta la robustez del sistema, garantizando un comportamiento determinístico del sistema, incluso ante diversos escenarios de falla.

Tanto la generación, como el funcionamiento de estos archivos se detalla en profundidad en el apartado 'Tolerancia a Fallas'.

6. Descripción de las Consultas

El sistema debe ser capaz de responder a las siguientes consultas, de acuerdo con la lógica planteada en el enunciado. A continuación, se detalla el paso a paso de cada una de ellas, indicando las tablas necesarias, las columnas relevantes y la secuencia de operaciones distribuidas.

6.1. Consulta 1: Transacciones filtradas

Objetivo: Obtener las transacciones (ID y monto) realizadas durante 2024 y 2025, entre las 06:00 y las 23:00 horas, con un monto total mayor o igual a 75.

- **Tablas requeridas:** `transactions`.
- **Columnas utilizadas:** `transaction_id`, `created_at`, `final_amount`.
- **Pasos:**
  1. Filtrar transacciones entre 2024 y 2025, en el rango horario indicado.
  2. Filtrar transacciones con monto mayor o igual a 75.
  3. Generar el reporte y devolverlo al usuario.

6.2. Consulta 2: Productos más vendidos y más rentables

Objetivo: Identificar, para cada mes de 2024 y 2025, los productos más vendidos (nombre y cantidad) y los productos que más ganancias generaron (nombre y monto).

- **Tablas requeridas:** `transaction_items`, `menu_items`.
- **Columnas utilizadas:**
  - De `transaction_items`: `item_id`, `created_at`, `quantity`, `subtotal`.
  - De `menu_items`: `item_id`, `item_name`.
- **Pasos:**
  1. Filtrar los `transaction_items` de 2024 y 2025.
  2. Generar una nueva columna `year_month` a partir de la fecha.
  3. Para productos más vendidos:
     a) Calcular contador de cantidad (`item_counter`).
     b) Reducir por clave `year_month, item_id` sumando las cantidades.
     c) Ordenar por cantidad descendente.
  4. Para productos más rentables:
     a) Calcular monto acumulado (`total_amount_counter`).
     b) Reducir por clave `year_month, item_id` sumando subtotales.
     c) Ordenar por monto descendente.
  5. Unir con `menu_items` para obtener nombres de los productos.
  6. Generar el reporte y devolverlo al usuario.

6.3. Consulta 3: TPV por semestre y sucursal

Objetivo: Calcular el *Total Payment Value* (TPV) por cada semestre de 2024 y 2025, para cada sucursal, considerando únicamente transacciones realizadas entre 06:00 y 23:00.

- **Tablas requeridas:** `transactions` y `stores`.
- **Columnas utilizadas:** `store_id`, `created_at`, `final_amount`.
- **Pasos:**
  1. Filtrar transacciones entre 2024 y 2025, dentro del rango horario.
  2. Generar nueva columna `year_semester` a partir de la fecha.
  3. Reducir por clave `year_semester, store_id` sumando los montos finales.
  4. Unir con `stores` para obtener nombres de las sucursales.
  5. Generar el reporte y devolverlo al usuario.

6.4. Consulta 4: Clientes más frecuentes y cumpleaños

Objetivo: Obtener la fecha de cumpleaños de los tres clientes con mayor número de compras durante 2024 y 2025, discriminado por sucursal.

- **Tablas requeridas:** `transactions`, `users` y `stores`.
- **Columnas utilizadas:**
  - De `transactions`: `user_id`, `store_id`, `created_at`.
  - De `users`: `user_id`, `birthdate`.
- **Pasos:**
  1. Filtrar transacciones de 2024 y 2025.
  2. Generar clave combinada `store_user` con `store_id` y `user_id`.
  3. Calcular contador de compras (`buys_counter`).
  4. Reducir por clave `store_user` sumando los contadores.
  5. Ordenar en forma descendente por número de compras.
  6. Seleccionar los tres usuarios con más compras por sucursal.
  7. Unir con `users` para obtener las fechas de nacimiento.
  8. Unir con `stores` para obtener nombres de las sucursales.
  9. Generar el reporte y devolverlo al usuario.

7. Vista de Escenarios

7.1. Casos de uso

Para modelar la interacción principal entre los actores y el sistema, se presenta un diagrama de casos de uso. Este diagrama identifica al actor principal/client, denominado Çafetería", y las cuatro funcionalidades clave que el sistema debe proveer, correspondiendo directamente a las consultas de negocio definidas en los requerimientos.

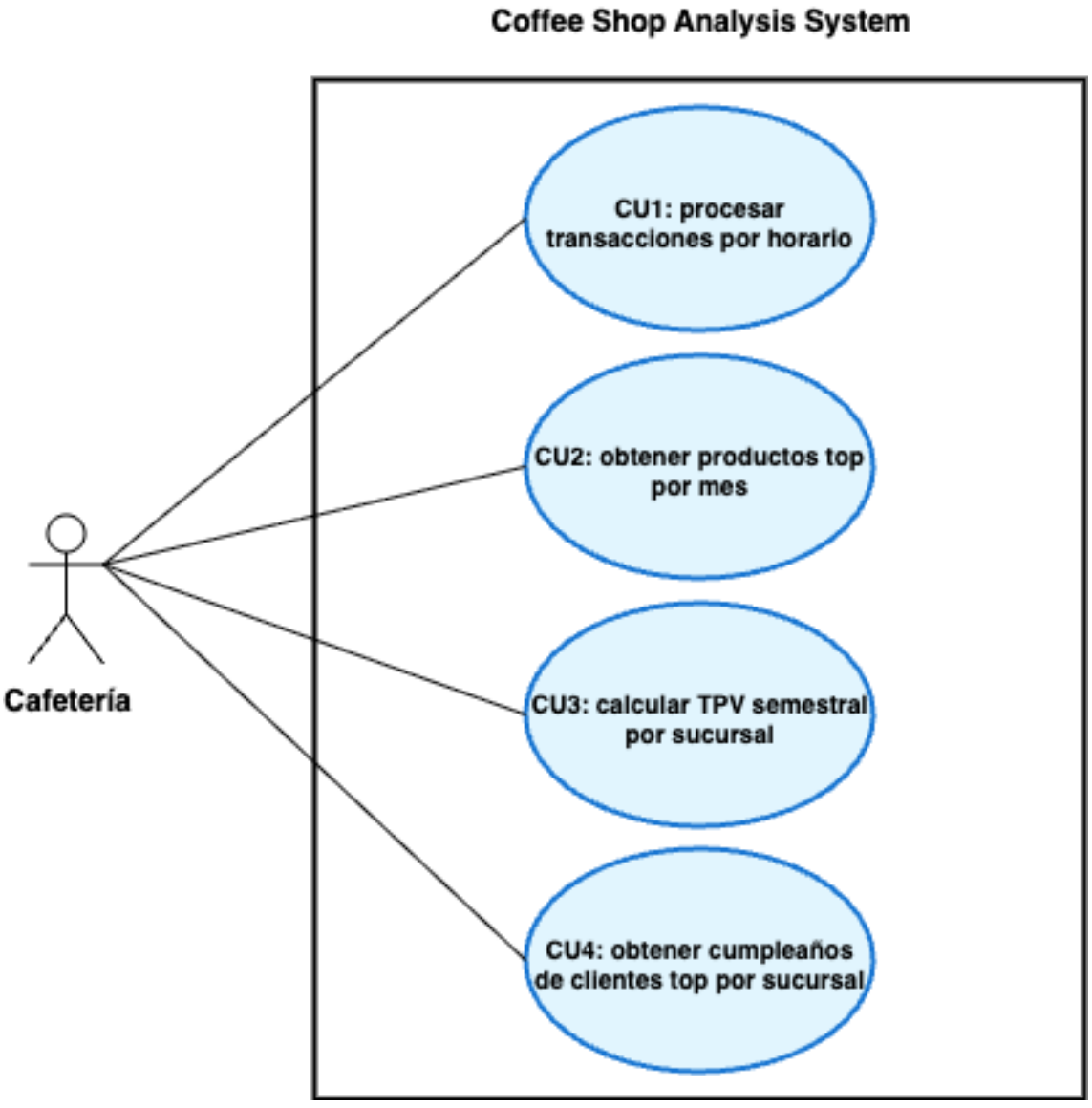


Figura 1: Casos de Uso

8. Vista de Procesos

8.1. Diagramas de Actividades

Los diagramas de actividad describen el flujo de trabajo lógico para cada una de las consultas funcionales. A continuación, se presenta un diagrama para cada consulta, detallando la secuencia de acciones y las decisiones desde el inicio de la solicitud hasta la presentación de los resultados finales.

8.1.1. Diagrama de Actividades de la primera consulta

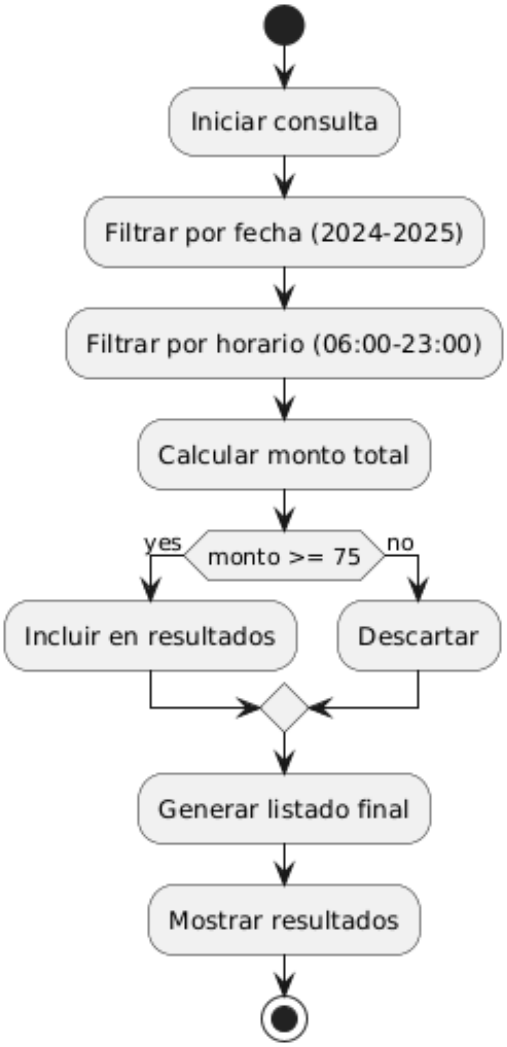


Figura 2: Actividad de la primera consulta

8.1.2. Diagrama de Actividades de la segunda consulta

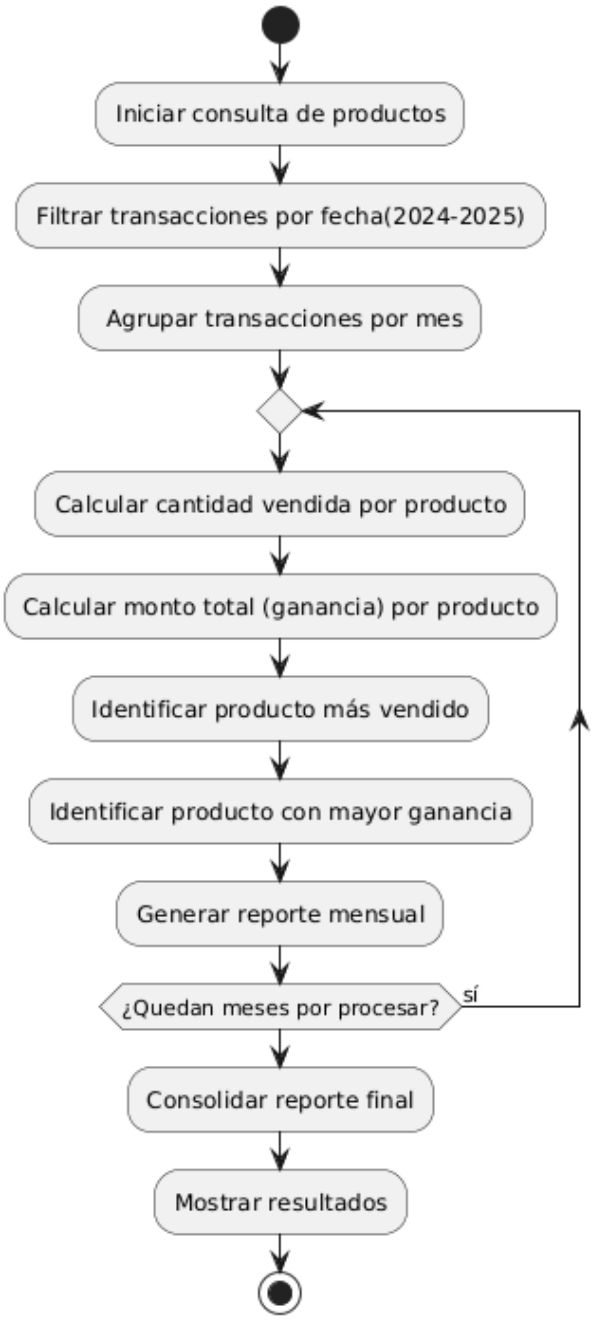


Figura 3: Actividad de la segunda consulta

8.1.3. Diagrama de Actividades de la tercera consulta

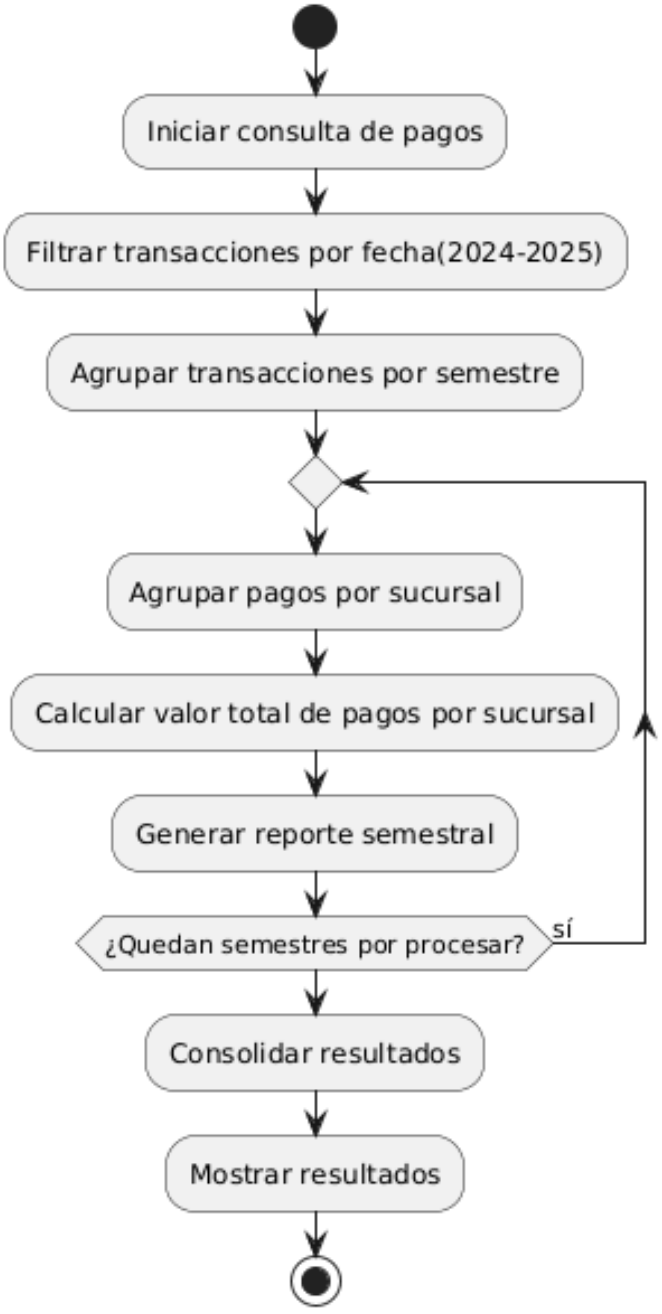


Figura 4: Actividad de la tercera consulta

8.1.4. Diagrama de Actividades de la cuarta consulta

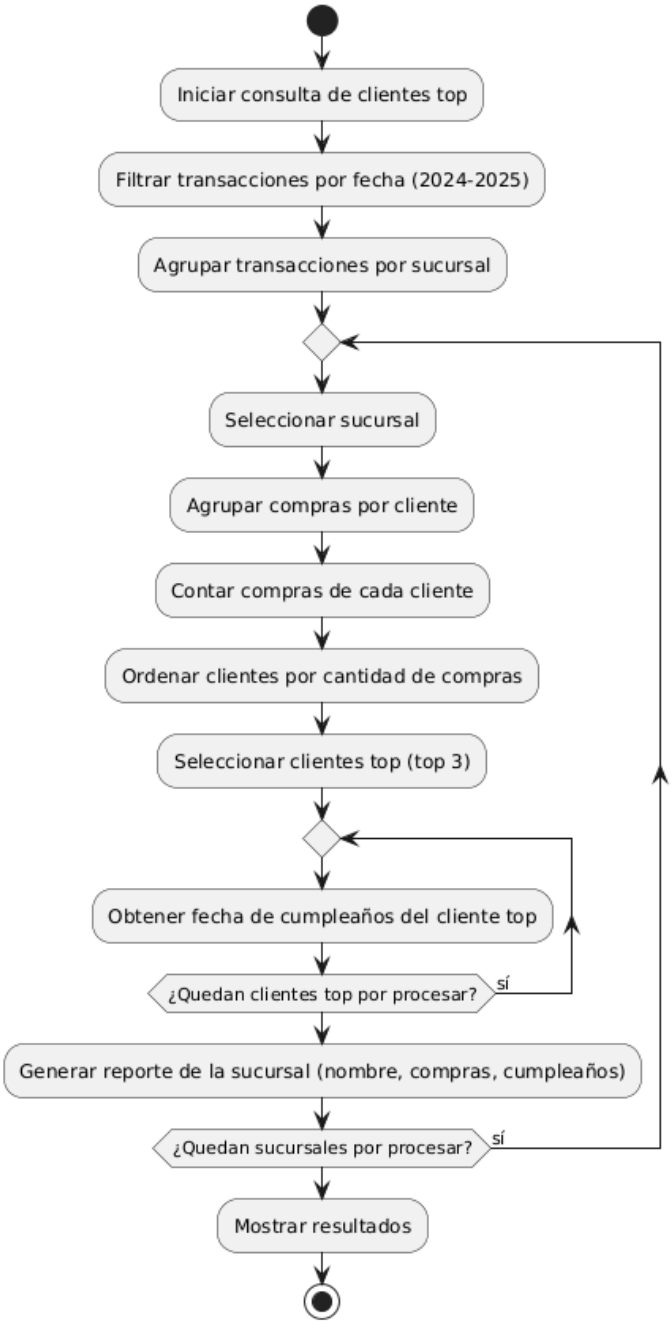


Figura 5: Actividad de la cuarta consulta

8.2. Diagramas de Secuencia

Para ilustrar la interacción temporal y el intercambio de mensajes entre los distintos componentes del sistema, se utilizan los diagramas de secuencia. Cada diagrama modela la ejecución de una consulta, mostrando cómo el Client Process envía lotes de datos al Server, el cual coordina las operaciones distribuidas entre los nodos especializados como Filter, GroupBy y ReduceBy hasta obtener y devolver el reporte final.

8.2.1. Diagrama de Secuencia de la primera consulta

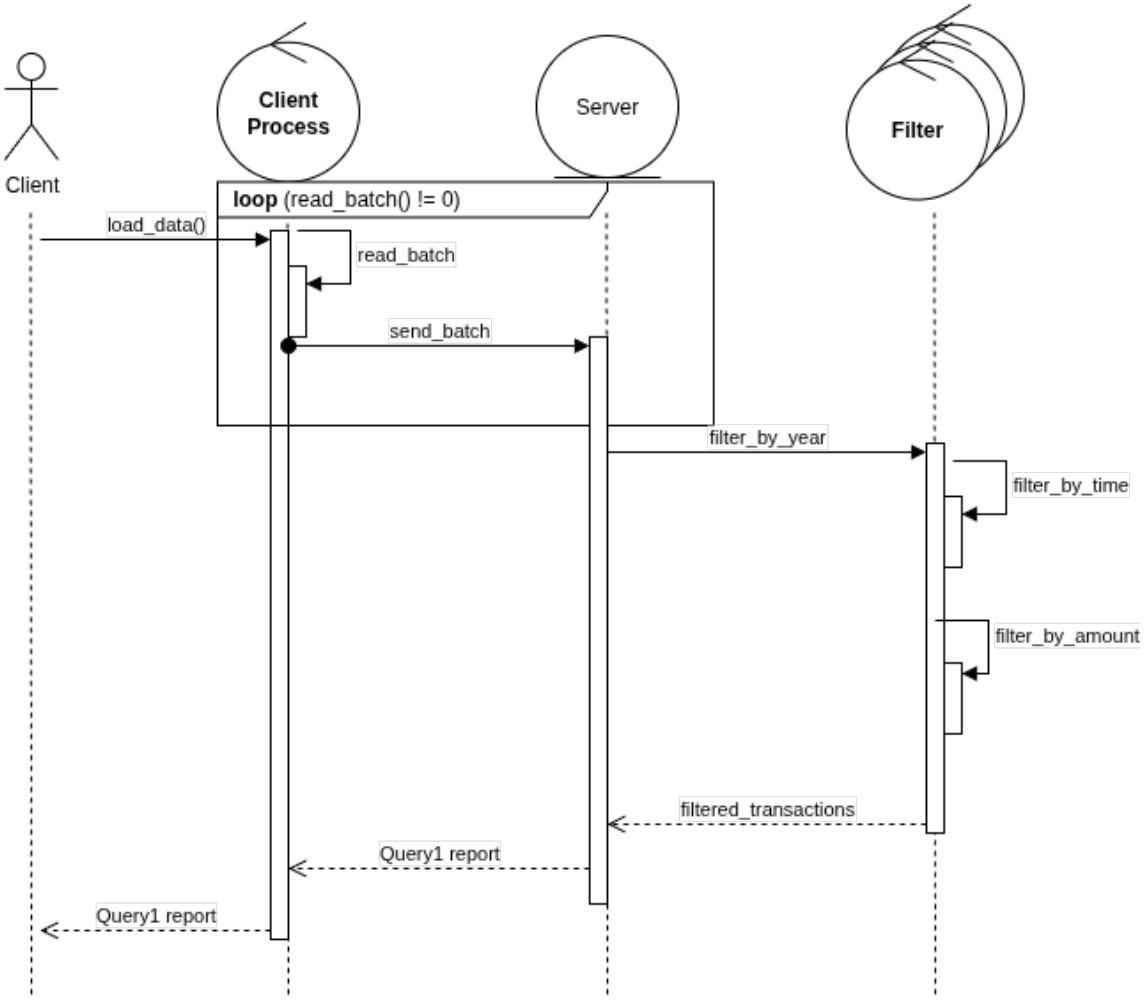


Figura 6: Secuencia de la primera consulta

8.2.2. Diagrama de Secuencia de la segunda consulta

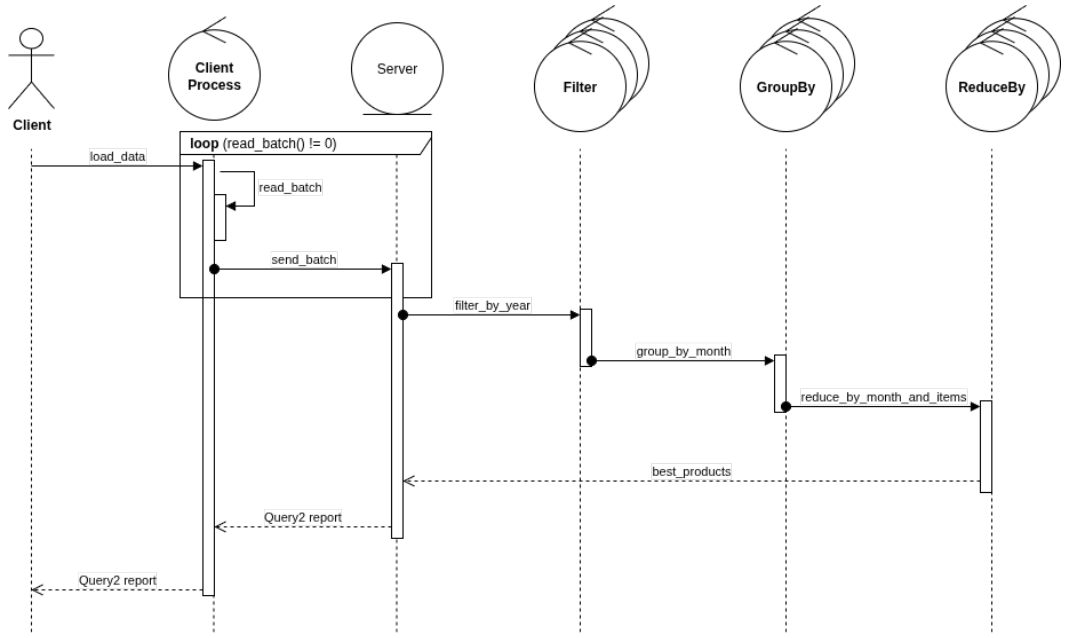


Figura 7: Secuencia de la segunda consulta

8.2.3. Diagrama de Secuencia de la tercera consulta

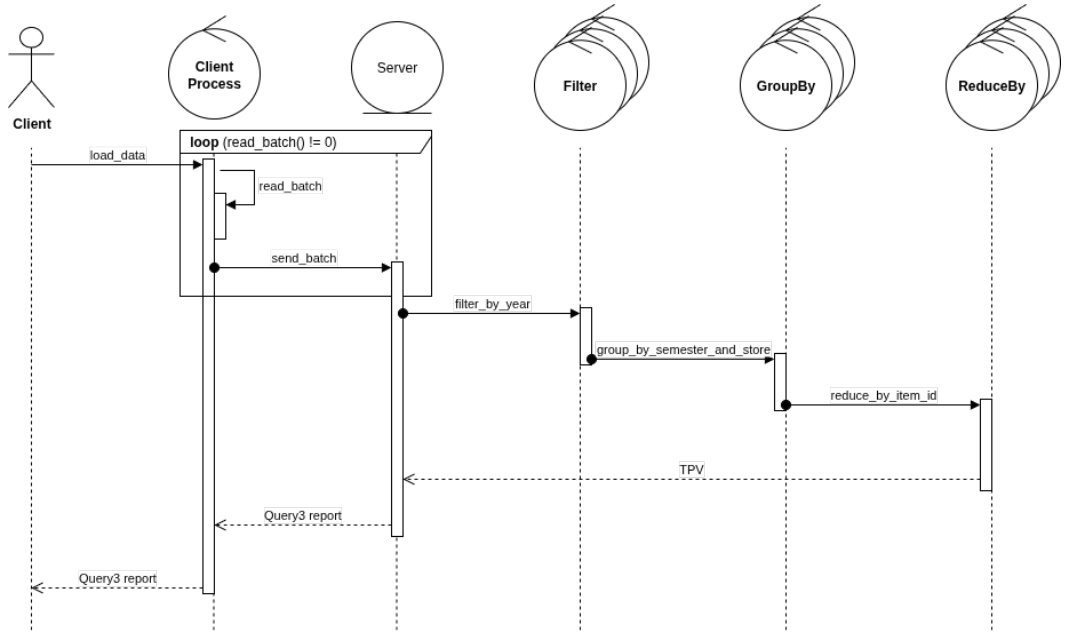


Figura 8: Secuencia de la tercera consulta

8.2.4. Diagrama de Secuencia de la cuarta consulta

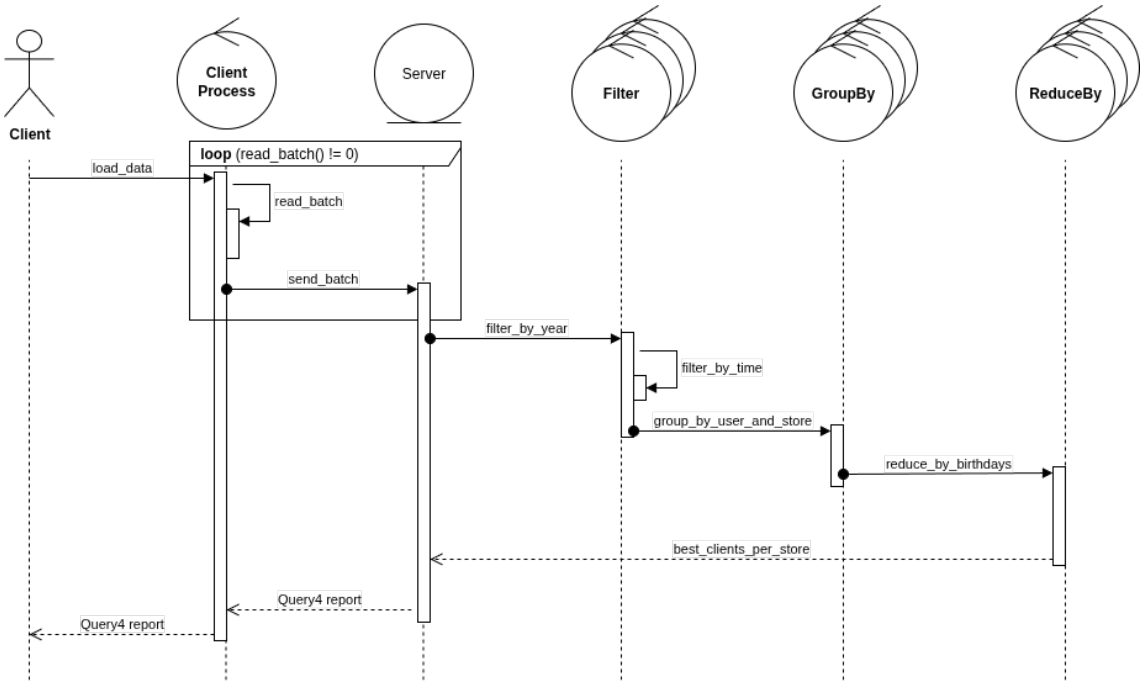


Figura 9: Secuencia de la cuarta consulta

9. Vista Física

9.1. Diagrama de Robustez

El diagrama de robustez forma parte de la vista lógica y sirve como puente entre los casos de uso y el diseño detallado. Este diagrama permite identificar y organizar los principales elementos del sistema, diferenciando entre actores externos, límites de la aplicación y las entidades o controladores que gestionan la lógica de negocio.

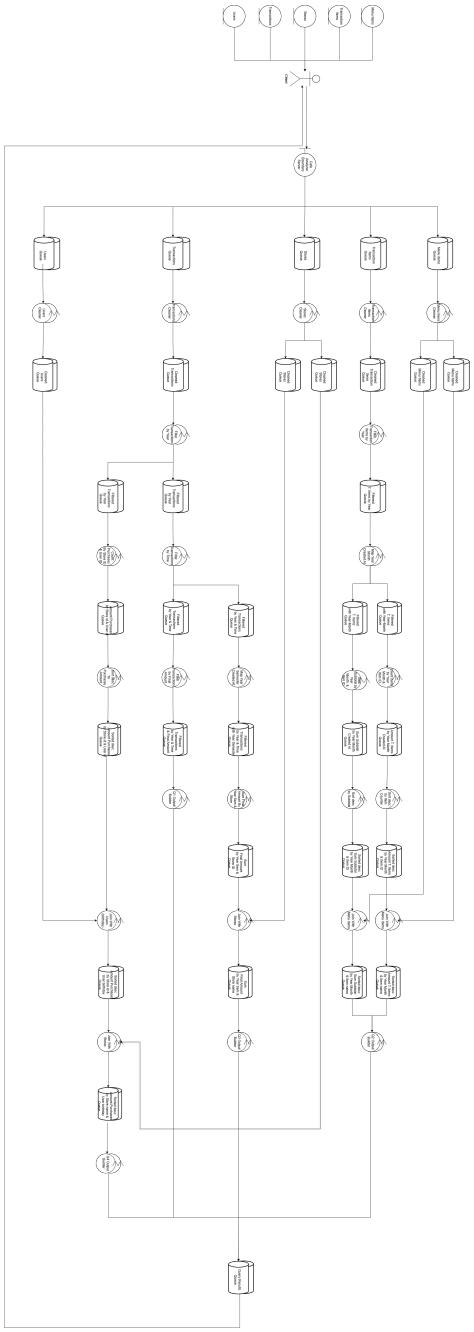
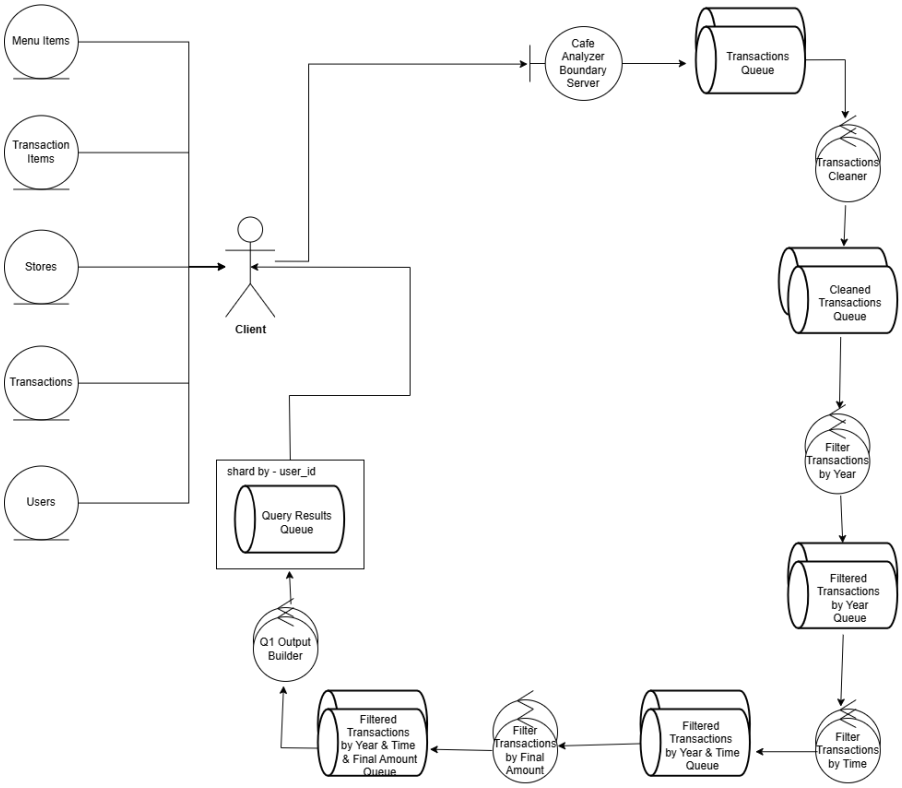
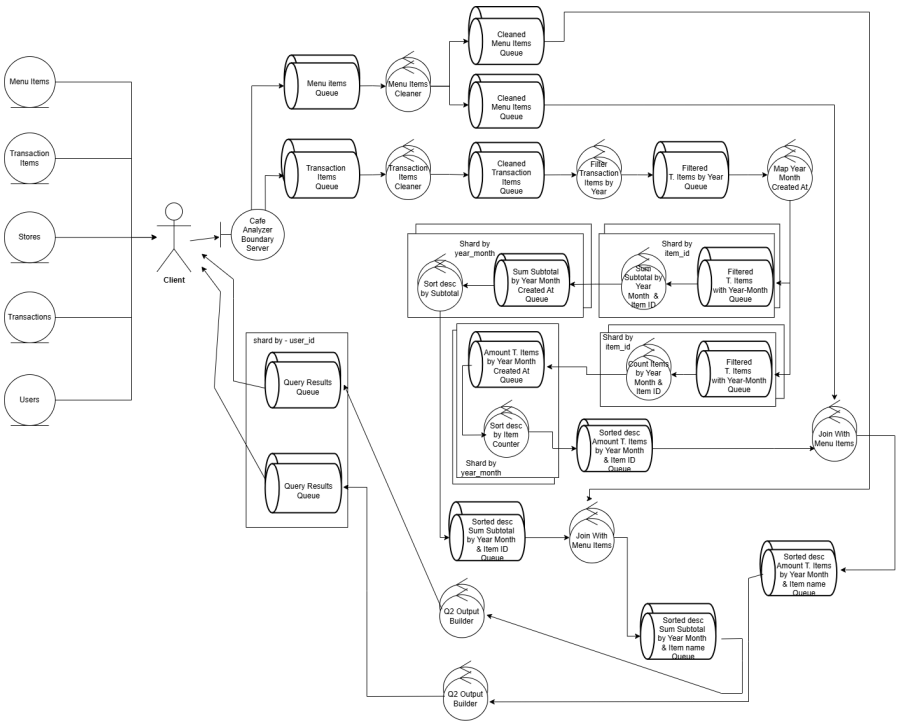


Figura 10: Diagrama de Robustez - Completo

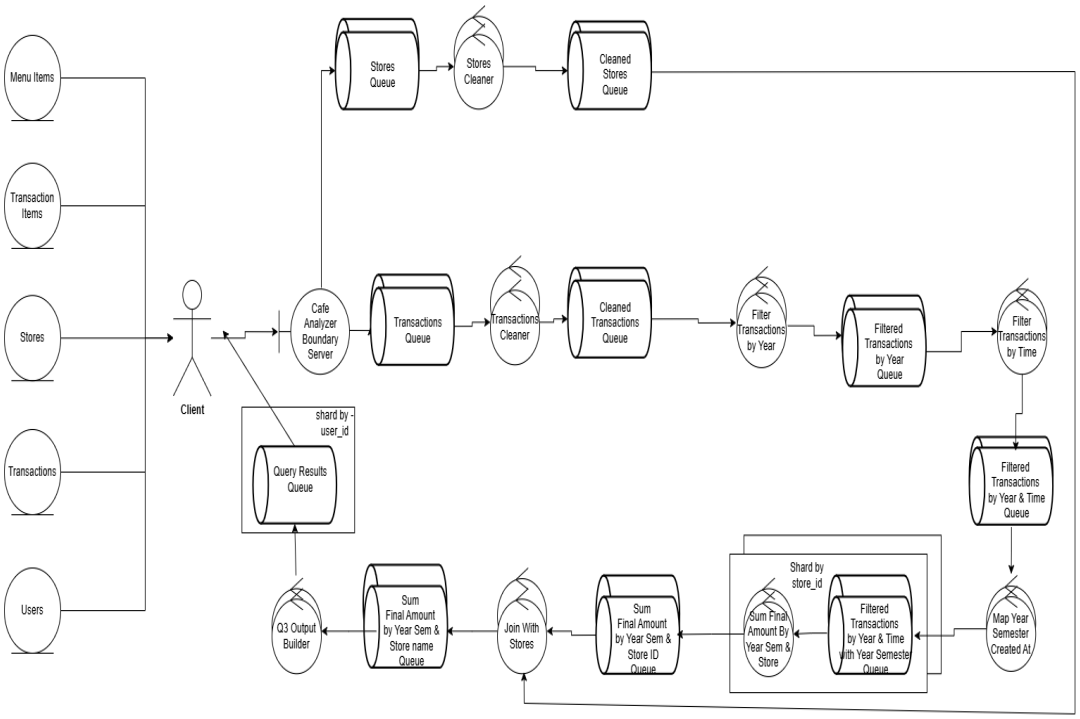
9.2. Diagrama de Robustez - Primera Consulta



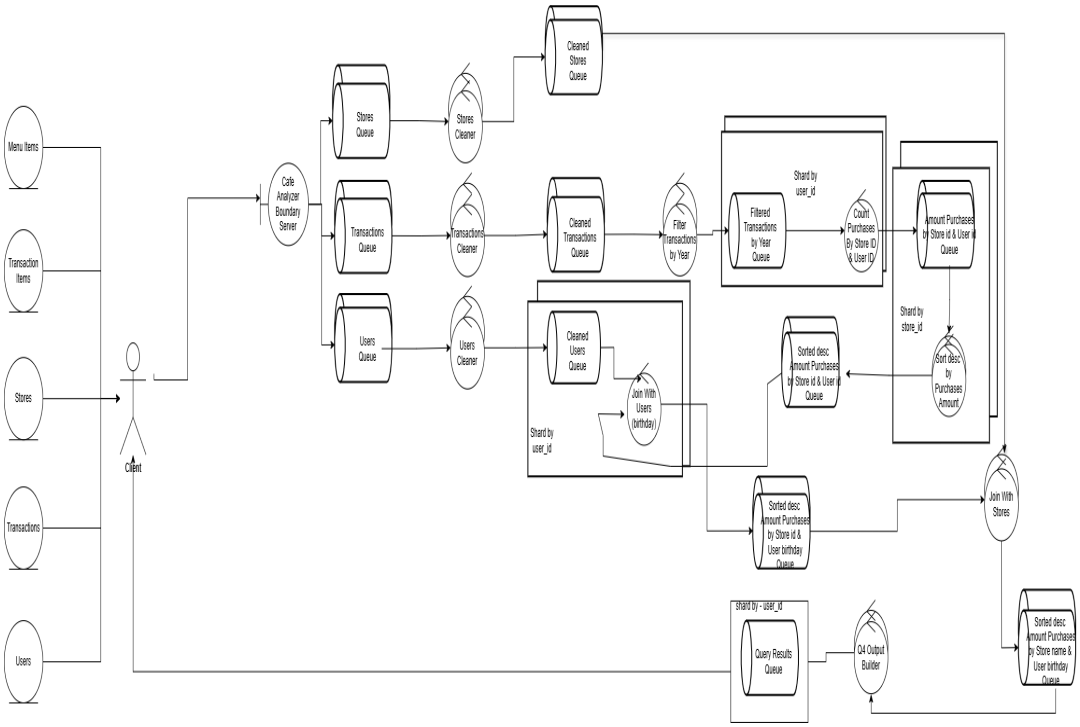
9.3. Diagrama de Robustez - Segunda Consulta



9.4. Diagrama de Robustez - Tercera Consulta



9.5. Diagrama de Robustez - Cuarta Consulta



9.6. Diagrama de Despliegue

Dentro de la vista física realizamos el diagrama de despliegue, el cual ilustra la topología del sistema distribuido en términos de sus nodos computacionales. El diagrama muestra los distintos grupos de nodos especializados (Data Cleaners, Filters, Maps, Joins, etc.) y cómo se interconectan a través de un componente central: el MOM Broker. Esta arquitectura facilita la comunicación asincrónica y el desacoplamiento entre los procesos.

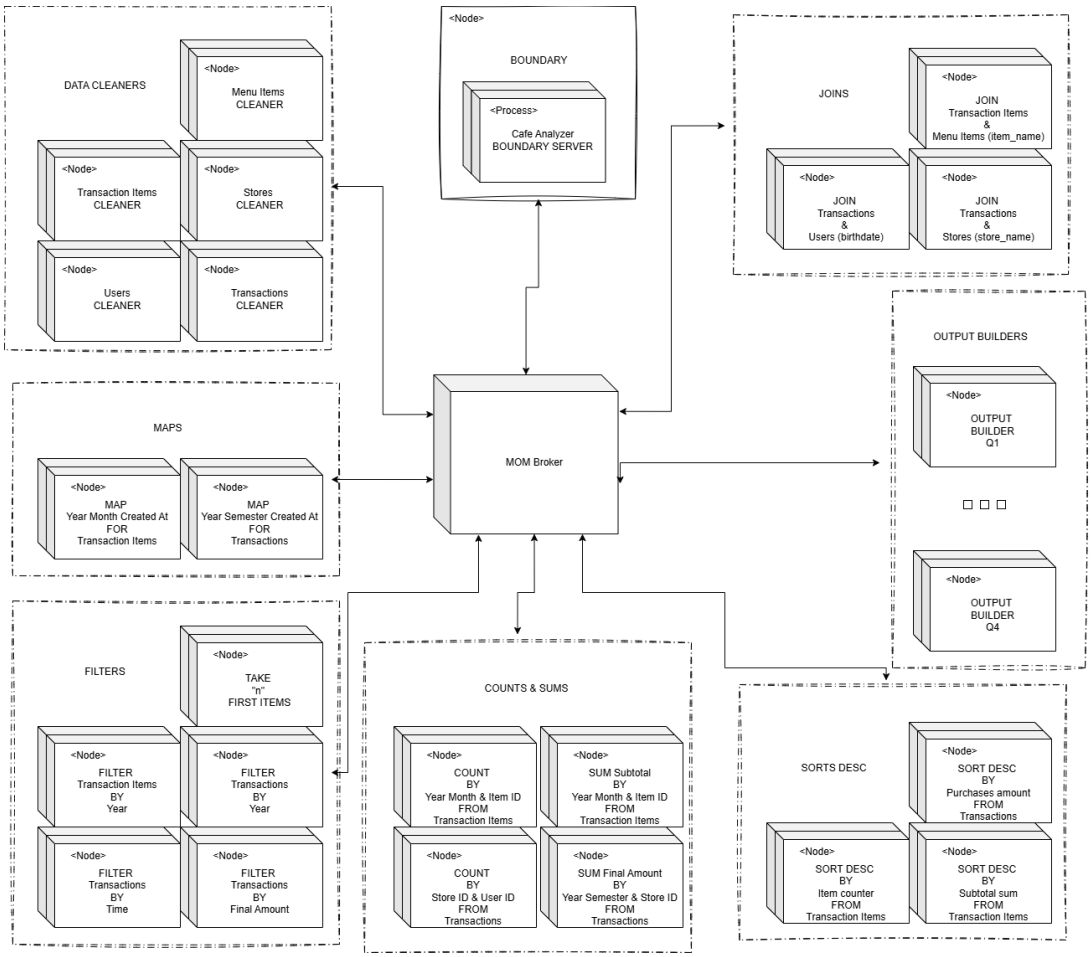


Figura 11: Diagrama de Despliegue

10. Vista de Desarrollo

10.1. Diagrama de Paquetes

La vista de desarrollo organiza el sistema en un conjunto de paquetes lógicos para promover una alta cohesión y un bajo acoplamiento. El diagrama muestra los componentes clave: el paquete shared contiene la lógica de comunicación común, como el MQConnectionHandler; el paquete controllers agrupa las distintas operaciones distribuidas (filtros, mapeos, etc.); y los paquetes client y server definen los puntos de entrada de la aplicación.

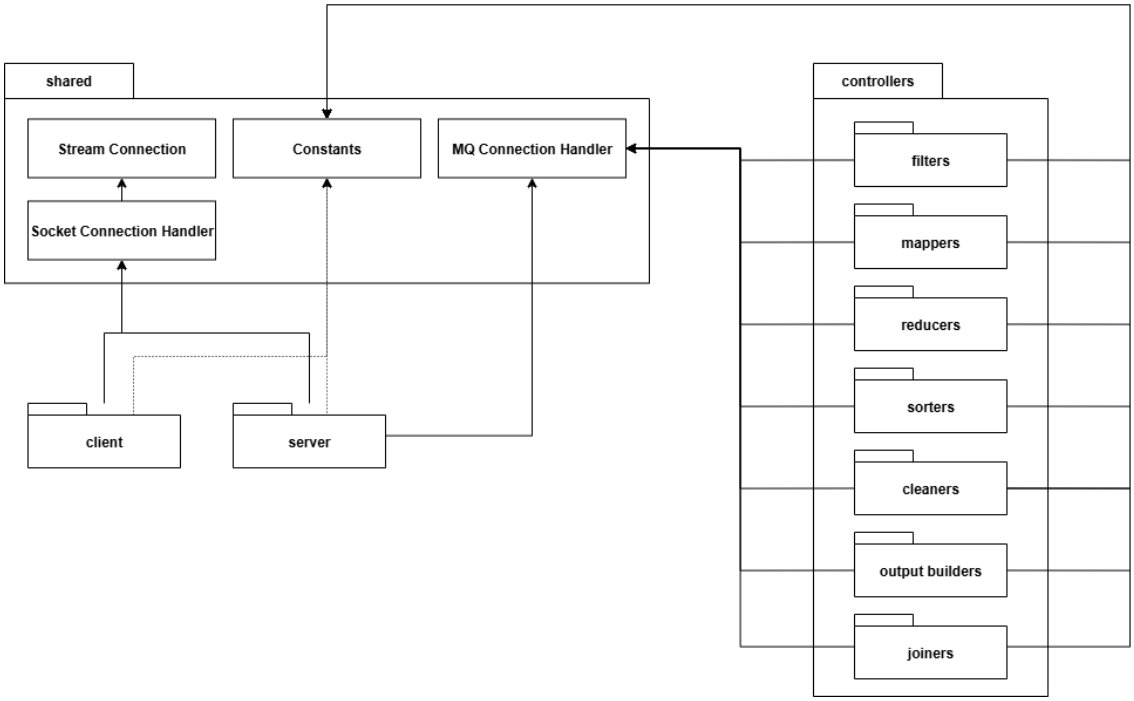


Figura 12: Vista de Desarrollo - Diagrama de Paquetes

11. Vista Lógica

11.1. Diagrama de Clases

La vista lógica detalla la estructura estática del sistema a través de diagramas de clases. El diseño se basa en la herencia y la abstracción para maximizar la reutilización de código. Como se observa en los siguientes diagramas, se define una clase base abstracta Controller de la cual heredan las distintas operaciones especializadas (Filter, Count, Sum, etc.), permitiendo que el sistema maneje diferentes tipos de tareas de manera uniforme y extensible.

11.1.1. Diagrama de Clases - Cleaner

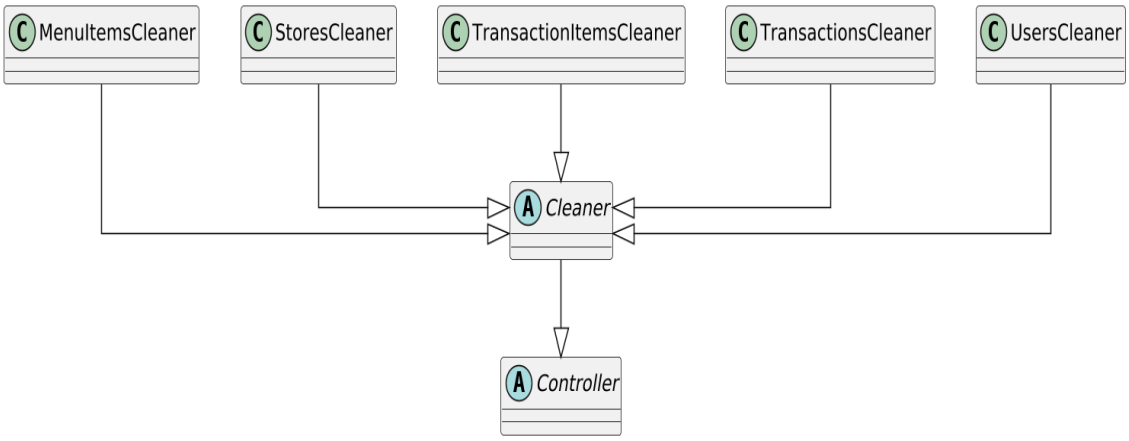


Figura 13: Clase Cleaner

11.1.2. Diagrama de Clases - Filter

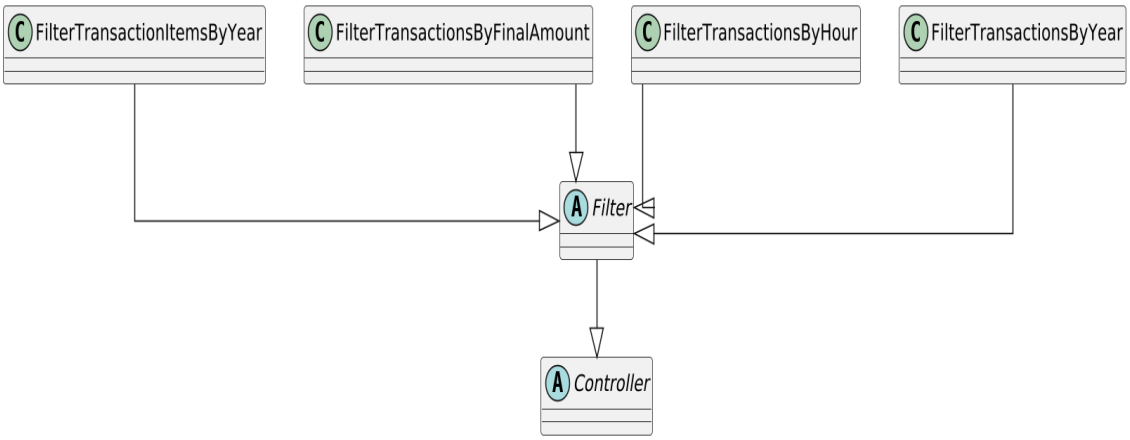


Figura 14: Clase Filter

11.1.3. Diagrama de Clases - Map

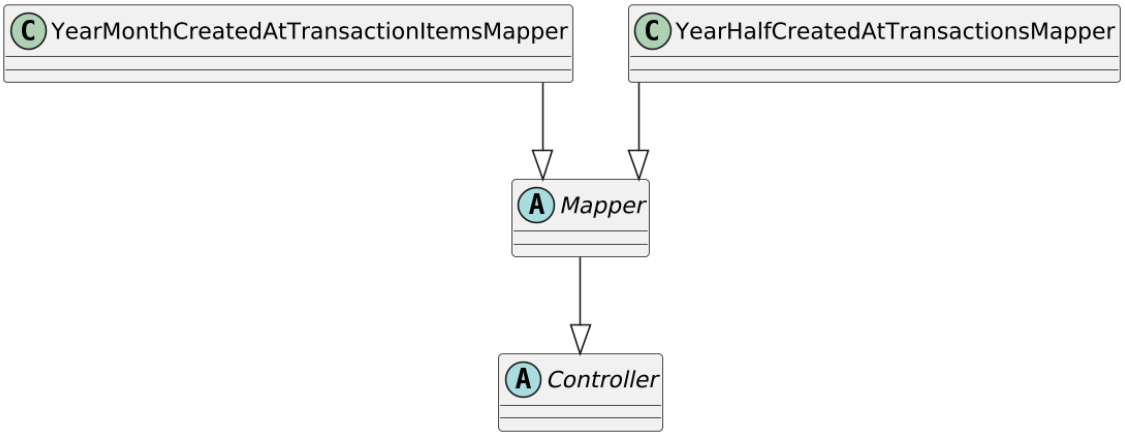


Figura 15: Clase Map

11.1.4. Diagrama de Clases - Reduce: Count y Sum

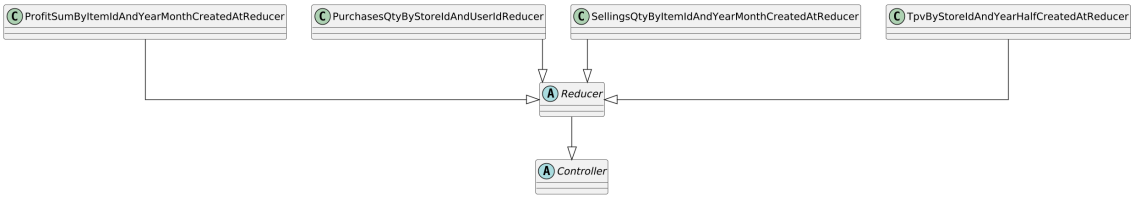


Figura 16: Clases Reduce: Count y Sum

11.1.5. Diagrama de Clases - Sort

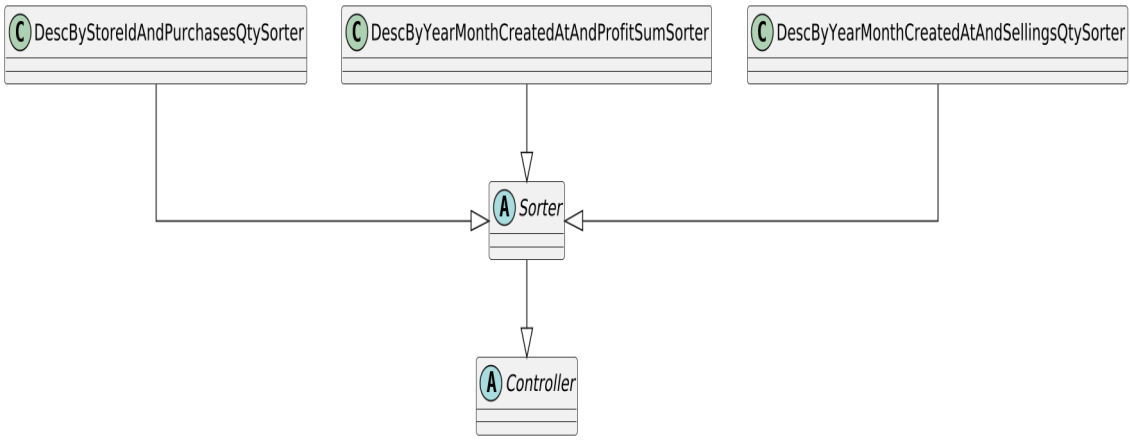


Figura 17: Clase Sort

11.1.6. Diagrama de Clases - Join

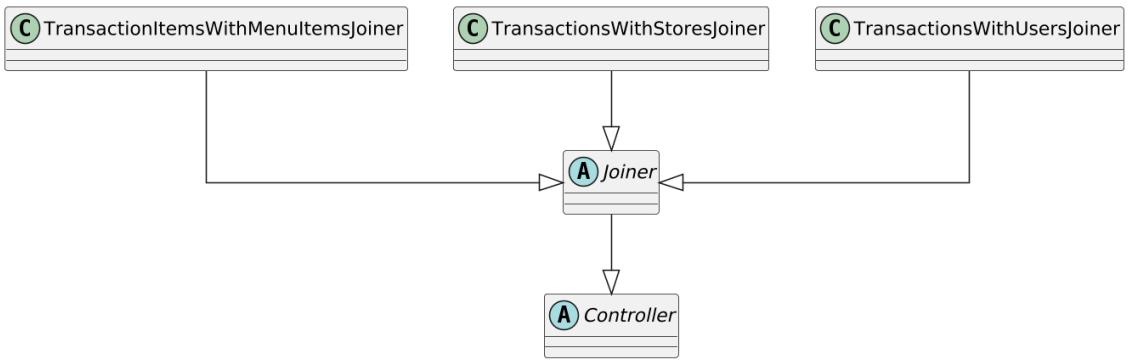


Figura 18: Clase Join

11.1.7. Diagrama de Clases - OutputBuilder

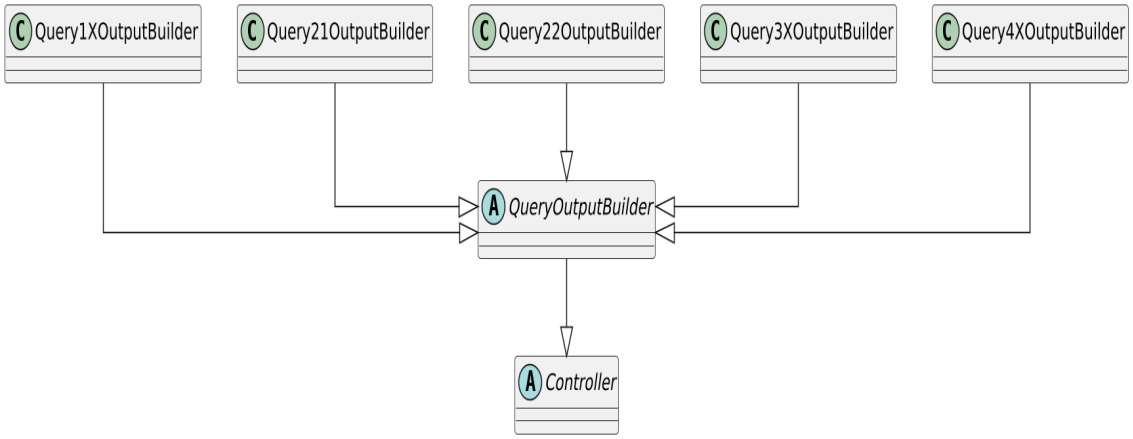


Figura 19: Clase OutputBuilder

12. Diagrama de Grafo Acíclico Dirigido (DAG) - Completo

El flujo de procesamiento completo para resolver las consultas se modela como un Grafo Acíclico Dirigido (DAG). Este diagrama visualiza las dependencias entre las distintas operaciones distribuidas. Cada nodo en el grafo representa una tarea (ej. Clean, Filter by Year, Count Items), y las aristas indican el flujo de datos desde una etapa hacia la siguiente, mostrando cómo se combinan las distintas fuentes de datos para producir los resultados finales (Q1, Q2, Q3, Q4).

12.1. Diagrama de Grafo Acíclico Dirigido (DAG) - Completo

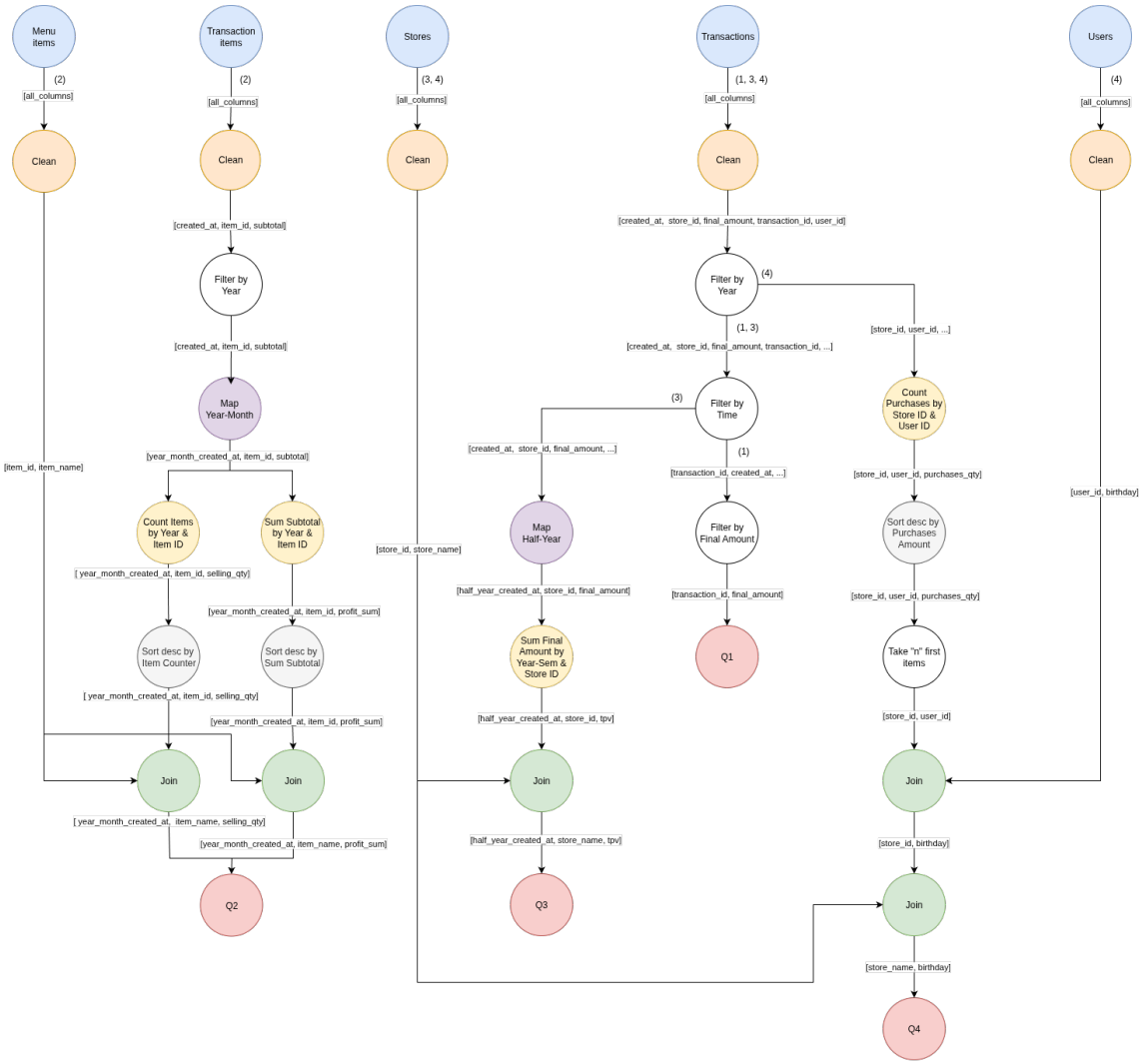


Figura 20: Diagrama de Grafo Acíclico Dirigido (DAG) - Completo

12.2. Diagrama de Grafo Acíclico Dirigido (DAG) - Primera consulta

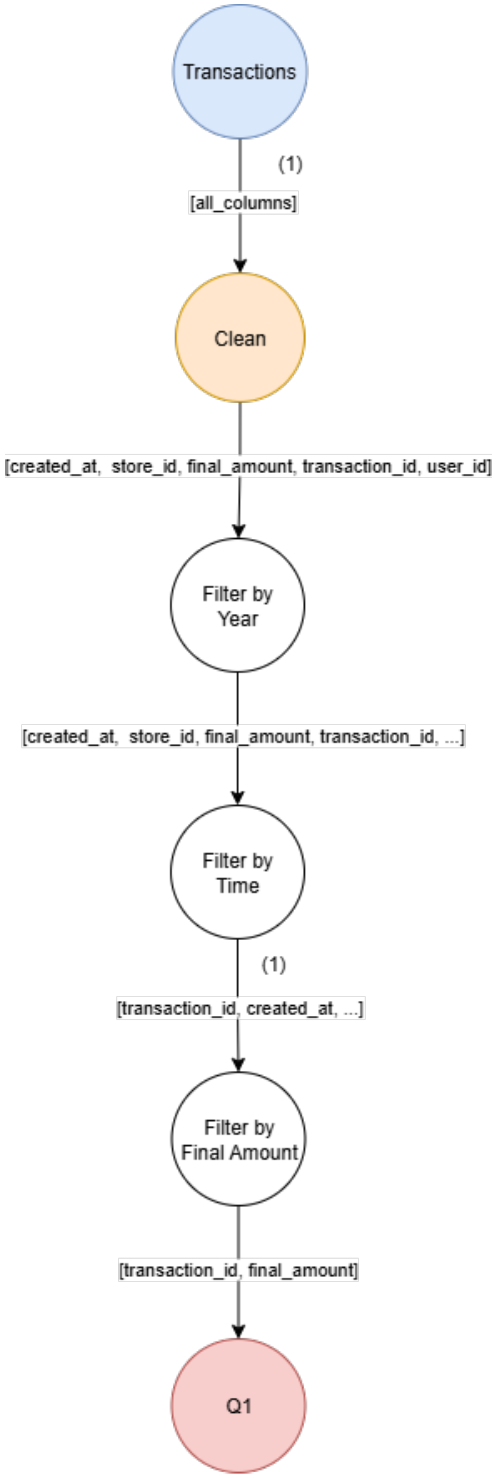


Figura 21: Diagrama de Grafo Acíclico Dirigido (DAG) - Primera consulta

12.3. Diagrama de Grafo Acíclico Dirigido (DAG) - Segunda consulta

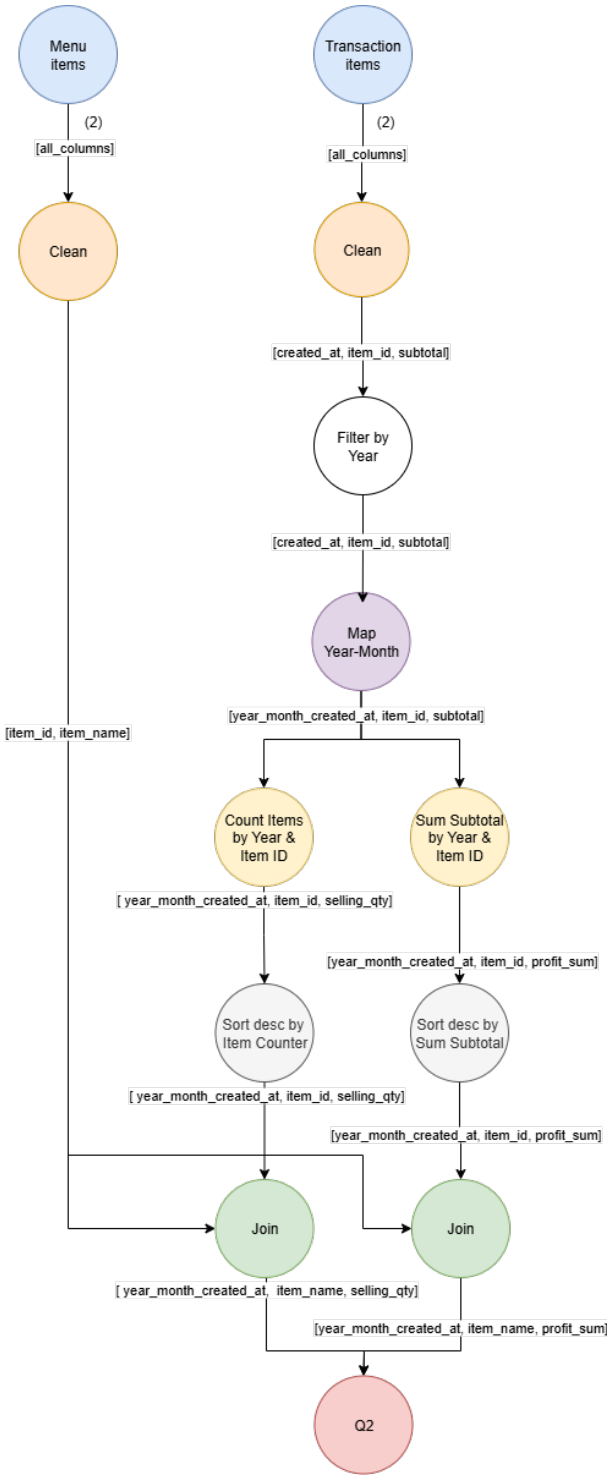


Figura 22: Diagrama de Grafo Acíclico Dirigido (DAG) - Segunda consulta

12.4. Diagrama de Grafo Acíclico Dirigido (DAG) - Tercera consulta

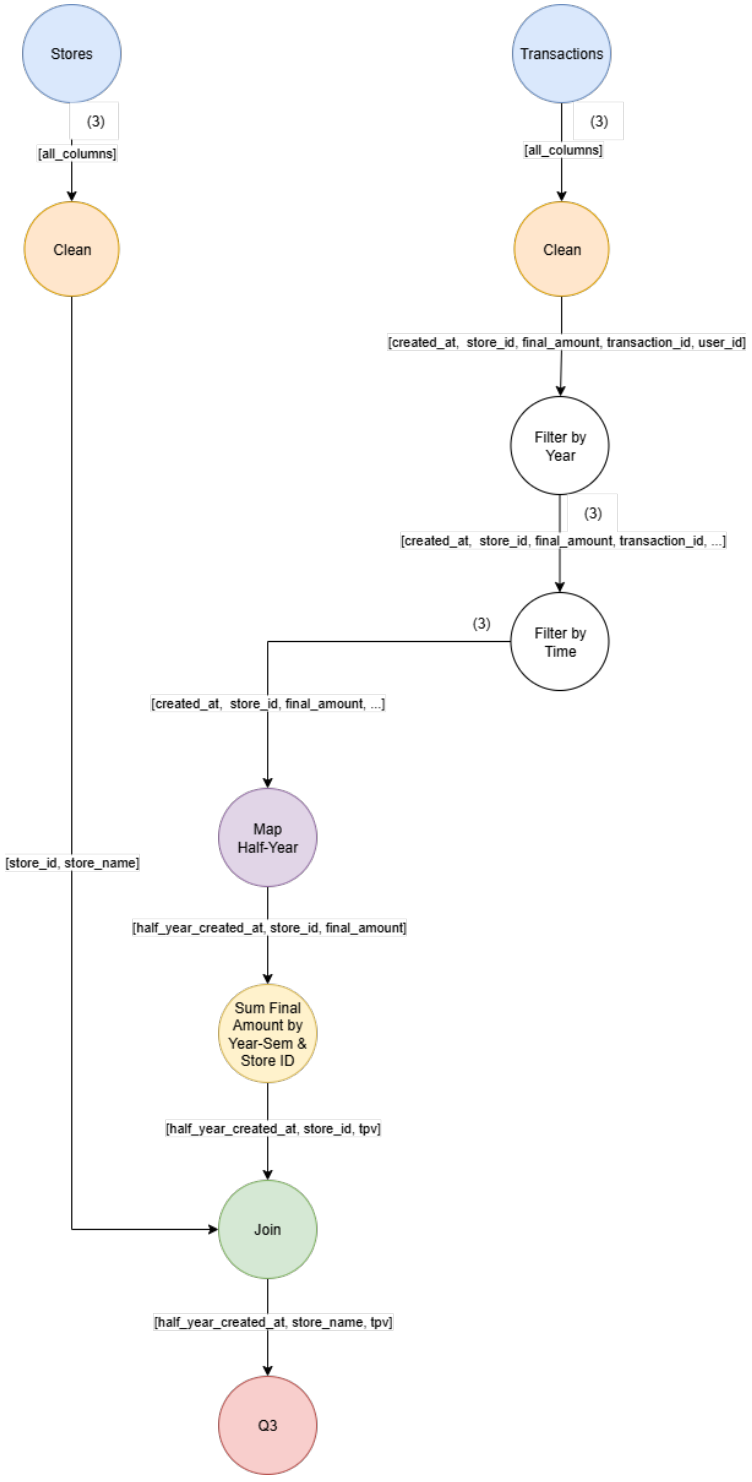


Figura 23: Diagrama de Grafo Acíclico Dirigido (DAG) - Tercera consulta

12.5. Diagrama de Grafo Acíclico Dirigido (DAG) - Cuarta consulta

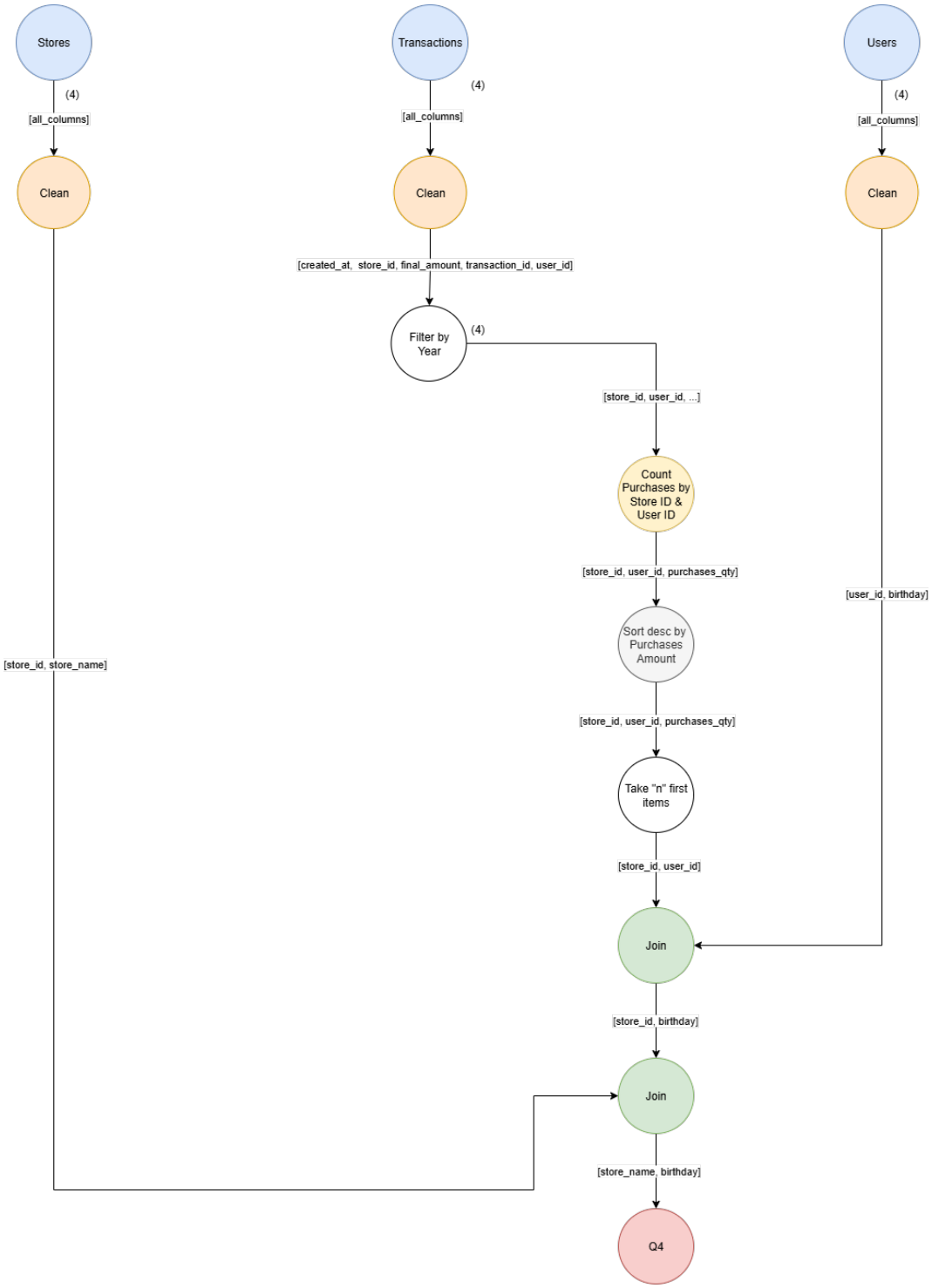


Figura 24: Diagrama de Grafo Acíclico Dirigido (DAG) - Cuarta consulta

13. Middleware

El middleware interno constituye el pilar fundamental de la comunicación en el sistema *Coffee Shop Analysis*, siendo el responsable de orquestar la interacción entre el servidor central y los múltiples nodos de procesamiento. Su arquitectura se basa en un modelo de Middleware Orientado a Mensajes (MOM), implementado mediante la herramienta **RabbitMQ**, que actúa como bróker central y mediador confiable entre los distintos procesos distribuidos, tal como se visualiza en el Diagrama de Despliegue.

La adopción de un middleware de este tipo permite abstraer completamente la complejidad de la comunicación en red, ocultando al programador los detalles de bajo nivel relacionados con sockets, serialización de datos o control de concurrencia. En lugar de conexiones rígidas punto a punto, los componentes se comunican a través de colas y *exchanges* definidos estratégicamente para cada escenario. Este enfoque incrementa notablemente la flexibilidad del sistema, ya que la incorporación de un nuevo nodo de procesamiento, o la eliminación de uno existente, no requiere modificar la lógica de negocio del resto de los participantes.

Otro aspecto central es la asincronía. Gracias al middleware, los nodos no dependen de la disponibilidad inmediata de sus contrapartes, sino que pueden depositar mensajes en una cola para que el receptor los procese cuando corresponda. Esto aporta robustez frente a picos de carga y tolerancia a fallos parciales, ya que los datos no se pierden aunque un nodo se encuentre momentáneamente inactivo. Asimismo, las colas actúan como un mecanismo de *buffering*, desacoplando el ritmo de producción y consumo de datos entre los distintos nodos.

La lógica de conexión se implementa en el componente `MQConnectionHandler`, ubicado en el paquete `shared`, que centraliza la configuración de colas, *exchanges* y canales de comunicación. De este modo, se asegura un manejo uniforme y estandarizado de las operaciones de envío y recepción de mensajes. Sobre esta capa de mensajería se diseñó además un protocolo de aplicación específico que define la estructura de los mensajes para el envío de datasets, la transmisión de resultados y el manejo de errores, lo que permite garantizar la correcta interpretación de la información en todas las etapas del procesamiento.

Finalmente, para interactuar con el middleware se emplearon las interfaces provistas por la cátedra, complementadas con la implementación de pruebas unitarias exhaustivas que validaron su correcto funcionamiento bajo diferentes escenarios de carga. De esta manera, el middleware cumple con el requisito planteado en el enunciado de abstraer la comunicación entre los nodos del sistema distribuido, aportando una base sólida para la escalabilidad, el desacoplamiento y la mantenibilidad del sistema en su conjunto.

Esquemas de comunicación

Durante el desarrollo se identificaron distintos patrones de comunicación, y para cada uno se diseñó una solución particular apoyada en las herramientas de RabbitMQ. A continuación, se describen los principales casos implementados

Comunicación mediante colas dedicadas

En los nodos escalables (Para esta versión todos los nodos de nuestro Sistema Distribuido son escalables), se definió una cola por cada instancia de nodo. Esto asegura la ausencia de *race conditions* y permite direccionar la información de forma controlada, garantizando el funcionamiento correcto y balanceado del sistema.

Para decidir a qué cola enviar cada batch de datos se utilizaron dos estrategias:

1. **Round Robin:** Cada emisor mantiene un contador que se incrementa en cada envío, distribuyendo los lotes de datos de forma equitativa entre las colas disponibles. Una vez alcanzado el último nodo, el contador retorna al primero. Esta técnica fue utilizada en situaciones como:
   - Servidor → Cleaners.
   - Cleaners → Filters.
   - Filters → Filters.
   - Filters → Output Builder.
   - Join Users → Join Stores.
   - Filters → Map Year Month / Map Year Semester.
2. **Hashing por clave (Sharding):** Se empleó en los casos en que los datos debían ser dirigidos a un nodo específico según una clave. Por ejemplo, en la comunicación de:
   - Cleaners de Usuarios → Joins de Usuarios.

   La asignación de transacciones a los nodos de Join se resolvió aplicando una función hash sobre el ID de usuario.

   Concretamente, se utilizó el resto de la división entera del ID por la cantidad de nodos shardeados, lo que asegura que todas las transacciones de un mismo usuario lleguen al mismo nodo de Join.

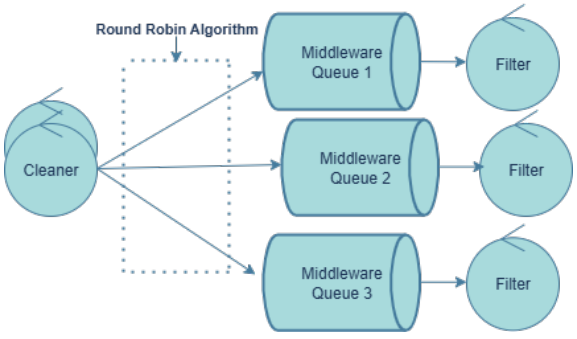


Figura 25: Esquema de comunicación mediante colas dedicadas.

Comunicación mediante exchanges

En situaciones donde la misma información debía ser replicada en múltiples colas, se utilizó un **exchange** de RabbitMQ. Este patrón resultó especialmente útil en la **Query 2**, donde se debía dividir la información para dos subconsultas distintas: los ítems más vendidos y los que generaron mayor facturación.

En este caso, el exchange distribuye la información procesada por el nodo *Map Year Month* hacia las colas de:

- Count Items.
- Sum Items.

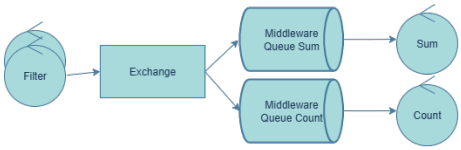


Figura 26: Esquema de comunicación mediante exchanges.

Detección y propagación de EOF

En la nueva versión del sistema, con soporte **MultiClient**, cada cliente opera dentro de su propio *contexto de sesión*, identificado por un `session_id` (UUID). Esto implica que la detección y propagación de los mensajes de fin de flujo (`EOF`) debe realizarse de forma independiente por sesión, garantizando el cierre correcto de cada flujo de datos asociado a un cliente específico.

El mecanismo general conserva la misma lógica que en la versión de un solo cliente, pero se amplía para manejar múltiples sesiones concurrentes:

- Cada nodo mantiene un **registro por sesión** de los emisores de los que espera recibir `EOF`. Es decir, no sólo sabe qué nodos le preceden, sino también a qué *session_id* pertenece cada flujo.
- Por cada `session_id`, el nodo espera recibir la cantidad exacta de mensajes `EOF` correspondientes a sus emisores aguas arriba.
- Cuando un nodo ha recibido todos los `EOF` esperados para una sesión dada, puede concluir que no habrá más datos de entrada para esa sesión en particular.
- En ese momento, el nodo genera y propaga su propio `EOF` —incluyendo el mismo `session_id`— hacia todos los nodos que le suceden, utilizando el mismo esquema de enrutamiento:
  - Enviando a una cola directa (1 a 1).
  - Replicando en un *exchange* compartido.
  - Distribuyendo por *Round Robin* entre varias colas consumidoras.
- Este proceso ocurre de forma independiente para cada sesión activa, permitiendo que distintos clientes finalicen sus flujos en momentos diferentes sin interferir entre sí.

De esta manera, el sistema logra una **propagación de EOF por sesión**, garantizando que cada cliente tenga un cierre completo y ordenado de su pipeline de procesamiento. Además, los nodos pueden liberar recursos por sesión una vez propagado el último `EOF`, mejorando la eficiencia y evitando bloqueos globales entre flujos concurrentes.

14. Mensajes y Protocolos

14.1. Protocolo

Para la implementación del sistema distribuido se actualizó el protocolo de comunicación a un esquema **MultiClient**, que permite multiplexar múltiples clientes concurrentes sobre el mismo backend mediante un *identificador de sesión* embebido en cada mensaje.

Este protocolo define las reglas de interacción entre el nodo coordinador, los nodos de procesamiento y los clientes del sistema. La comunicación se basa en el intercambio de mensajes *auto-contenidos* con un encabezado de tipo fijo, un *identificador de sesión* y un *payload* estructurado. Los mensajes expresan operaciones, datos a procesar y resultados obtenidos, y contemplan el envío por lotes (*batch*) de datasets, una negociación inicial de sesión (`HSK`) y señales de fin de flujo por tipo (`EOF`).

14.2. Mensajes

Como parte del protocolo, se mantiene un conjunto de tipos de mensajes de longitud fija (3 caracteres), orientados a cubrir las operaciones básicas del sistema: envío de datasets por lotes y resultados de queries. Con MultiClient, **todo** mensaje lleva además un *session id* entre el tipo y el payload. Cada mensaje posee:

- Un **tipo** de 3 caracteres (p.ej., `HSK`, `MIT`, `TRN`).
- Un **delimitador** de sesión `|` que separa tipo y *session id*.
- Un **identificador de sesión** (*session id*) opaco, provisto por el cliente.
- Un **delimitador** de inicio de payload `[` y de fin `]`.
- Un **payload** cuyo formato depende del tipo (texto libre o lote estructurado).

A continuación se detalla el funcionamiento concreto, a partir del código provisto.

14.3. Formato de los mensajes

Estructura general. Todo mensaje sigue ahora el formato:

```
<TYPE>|<SESSION_ID>[<PAYLOAD>]
```

Donde:

- `<TYPE>` es un prefijo de **3 caracteres** (`MESSAGE_TYPE_LENGTH = 3`).
- `<SESSION_ID>` es una cadena opaca definida por el cliente. El separador entre tipo y sesión es `SESSION_ID_DELIMITER=|`.
- El **payload** va entre corchetes: `[ ... ]` (`MSG_START_DELIMITER` y `MSG_END_DELIMITER`).

Payload por lotes (batch). Para los mensajes que transportan registros, el payload es una **secuencia de filas**, cada una **encapsulada en llaves** y separada por punto y coma:

```
{<FIELDS>};{<FIELDS>};...
```

Cada `<FIELDS>` es una secuencia de pares clave–valor separados por comas:

```
<FIELD>,<FIELD>,...
```

Un `<FIELD>` tiene la forma (comillas **dobles**):

```
"key":"value"
```

Delimitadores usados en el batch:

- Inicio/fin de **cada fila**: `BATCH_START_DELIMITER={`, `BATCH_END_DELIMITER=}`
- Separador de filas: `BATCH_ROW_SEPARATOR=;`
- Separador de campos: `ROW_FIELD_SEPARATOR=,`

Ejemplo de batch con dos filas (dentro del payload):

```
{"id":"m001","name":"Latte","price":"4.50"};{"id":"m002","name":"Espresso","price":"3.20"}
```

Sin escape de caracteres. No se realiza *escaping*; por lo tanto, las claves y valores no pueden contener comillas dobles `"`, dos puntos `:`, comas `,`, punto y coma `;`, llaves , corchetes `[ ]` ni el delimitador de sesión |.

14.4. Tipos de mensajes y su propósito

Tipo (3 chars)	Uso
`HSK`	*Handshake* de inicio de sesión MultiClient (negociación).
`MIT`	Lote de *menu items*.
`STR`	Lote de *stores*.
`TIT`	Lote de *transaction items*.
`TRN`	Lote de *transactions*.
`USR`	Lote de *users*.
`Q1X`	Resultado para Query 1 (*variant X*).
`Q21`	Resultado para Query 2.1.
`Q22`	Resultado para Query 2.2.
`Q3X`	Resultado para Query 3 (*variant X*).
`Q4X`	Resultado para Query 4 (*variant X*).
`EOF`	Señal de fin de flujo (payload: el tipo de flujo que finaliza).

Nota: en esta versión, `ACK` y `QRY` no forman parte del *wire protocol*. La coordinación se realiza con `HSK` y flujos por sesión. Se expone además el *alias* `ALL_QUERIES = "Q1X;Q21;Q22;Q3X;Q4X"` para expresar capacidades o intereses en el *handshake*.

14.5. Codificación (*encode*)

Mensajes genéricos con sesión. La función privada `_encode_message(type, session_id, payload)` compone:

```
type
|
session_id
[
payload
]
type|session_id[payload]
```

A partir de ella se exponen:

- `encode_handshake_message(id: str, payload: str) -> str`:
  Genera `HSK`id[¡payload¿]— .
- `encode_eof_message(session_id: str, message_type: str) -> str`:
  Genera `EOF\textbar session_id[<message_type>]`, donde el *payload* es el tipo de flujo que se declara finalizado (p.ej., `EOF\textbar s1[TRN]`).

Mensajes por lotes. `encode_batch_message(batch_msg_type, session_id, batch)`:

1. Para cada `row: dict[str, str]`, codifica campos como `"key":"value"` unidos con `,` y los envuelve con `{}`.
2. Une las **filas codificadas** con `;` *(no hay llaves globales del batch).*
3. Envuelve con el tipo, `|session_id` y corchetes de mensaje.

14.6. Decodificación (*decode*)

Acceso a tipo, sesión y payload.

- `get_message_type(msg)`: Devuelve los primeros 3 caracteres. Valida largo mínimo; lanza `ValueError` si el mensaje es demasiado corto.
- `get_message_session_id(msg)`: Extrae el *session id* entre `|` y `[`.
- `get_message_payload(msg)`: Remueve tipo, `|session\_id` y `[]` exteriores, retornando la cadena interna.
- `message_without_payload(msg)`: Verdadero si el payload está vacío.

Validación estructural. `_assert_message_format(expected_type, msg)` verifica:

1. Que el tipo decodificado sea el esperado.
2. Que exista el delimitador de sesión `|` y los corchetes `[`, `]` exteriores.

En caso contrario, lanza `ValueError`.

Decodificación de lotes. `decode_batch_message(msg)`:

1. Obtiene el payload.
2. Separa **filas** por `;`.
3. Para cada fila, aplica `_decode_row`, que:
   - Remueve llaves `{}` en extremos.
   - Separa campos por `,`.
   - Divide cada campo por el primer `:` en clave y valor.
   - Quita comillas dobles exteriores de clave y valor.
4. Devuelve `list[dict[str,str]]`.

Handshake y fin de flujo.

- `decode_handshake_message(msg)` valida `HSK` y retorna `(session_id, payload)`. El payload puede expresar capacidades, p.ej. `ALL_QUERIES`.
- `decode_eof_message(msg)` valida `EOF` y retorna el payload (el *identificador de tipo de flujo* que finaliza; p.ej., `TRN`, `USR`, etc.).

14.7. Ciclos de vida típicos de los mensajes

Establecimiento de sesión (MultiClient).

1. El cliente genera un **session id** (p.ej., `s007`) y emite `HSK|s007[...]` con parámetros de negociación (p.ej., `ALL_QUERIES`).
2. El backend asocia la sesión y, a partir de entonces, **todo** mensaje de/para ese cliente incluirá `s007[,,]` .

Ingesta de datasets por lotes. Para cada dataset (p.ej., `MIT`, `TRN`):

1. El productor envía uno o más mensajes **batch** del tipo correspondiente, incluyendo su *session id* (p.ej., `MIT|s007[...]`).
2. Al finalizar el stream de ese dataset, el productor envía `EOF|s007[<TIPO>]`, por ejemplo `EOF|s007[TRN]`.

Consultas y resultados.

1. Un cliente puede declarar interés/capacidades en `HSK` (p.ej., `ALL_QUERIES`).
2. Los resultados se retornan *por sesión* en mensajes `Q1X`, `Q21`, `Q22`, `Q3X` o `Q4X`, con payload acorde, por ejemplo `Q21|s007[...]`.

14.8. Ejemplos de mensajes codificados

Handshake con capacidades.

```
HSK|s007[Q1X;Q21;Q22;Q3X;Q4X]
```

Batch de menu items (2 filas).

```
MIT|s007[{"id":"m01","name":"Latte","price":"4.50"};{"id":"m002","name":"Doble","price":"3.20"}]
```

Fin de flujo de transacciones.

```
EOF|s007[TRN]
```

Resultado de Query 2.1 (payload libre en este TP).

```
Q21|s007[{"store_id":"s007","metric":"top_seller","value":"m001"}]
```

14.9. Validaciones, errores y robustez

- **Tipo y formato:** Todo decodificador específico verifica que el tipo de mensaje coincida con el esperado, que exista el delimitador de sesión | y que el payload esté entre `[,,]`. Inconsistencias lanzan `ValueError`.
- **Longitud mínima:** `get_message_type` exige al menos 3 caracteres (tipo). Mensajes más cortos disparan `ValueError`.
- **Payload vacío:** Admitido (`message_without_payload`).
- **Complejidad:** Las rutinas de encode/decode son lineales en la longitud del mensaje, con operaciones de `split`/`join` sobre separadores fijos, facilitando el procesamiento por streaming.

14.10. Supuestos y limitaciones deliberadas

- **Sin escape de caracteres:** Claves y valores no deben contener comillas dobles `"`, dos puntos `:`, comas `,`, punto y coma `;`, llaves/corchetes, ni el delimitador de sesión |. Esto simplifica y acelera el parser a costa de restringir el dominio de valores posibles (suficiente para este TP).
- **Tipos de datos:** Todos los valores se tratan como cadenas (`str`); la tipificación semántica queda a cargo de las capas de negocio.
- **Orden y entrega:** El protocolo describe *formato* y *semántica de alto nivel* (handshake, batches, EOF) y la multiplexación por sesión. Las garantías de orden, reintentos o *at-least-once/exactly-once* pertenecen a la capa de transporte y orquestación (p.ej., colas), tratadas en otra sección del informe.

14.11. Extensibilidad

El diseño con prefijos de 3 caracteres y *session id* explícito facilita agregar nuevos tipos de mensajes sin afectar a los existentes. Para incorporar uno nuevo basta con:

1. Declarar la constante del tipo (3 letras).
2. Implementar, si corresponde, *helpers* de encode/decode análogos a los de batch.
3. Documentar el payload asociado (texto libre o esquema de campos del batch).

14.12. Resumen

El protocolo MultiClient define un **formato compacto, determinista y con multiplexación**:

- encabezado de 3 caracteres para el tipo
- separador | y *session id* obligatorio
- delimitadores exteriores `[,,]`
- para datasets, un **batch** como secuencia *fila-a-fila*: `{};` con separadores simples

Se proveen *helpers* específicos para codificar/decodificar los distintos datasets, un `HSK` para iniciar la sesión y una señal `EOF` que marca el fin de cada flujo lógico por sesión. Las validaciones estructurales minimizan errores de parseo y el diseño favorece procesamiento lineal, multi-cliente y extensibilidad controlada para futuras necesidades del TP.

15. Mediciones de rendimiento (Sin fallas)

Para evaluar el rendimiento del sistema distribuido implementado, se realizaron diversas mediciones bajo diferentes configuraciones y cargas de trabajo, poniendo especial énfasis en la capacidad del sistema para manejar múltiples clientes en ejecución simultánea (*MultiClient*) sobre el dataset completo.

A continuación, se detallan los resultados obtenidos y su análisis correspondiente.

15.1. Configuración del entorno de pruebas

Las pruebas se llevaron a cabo en un entorno controlado, utilizando una red local con varios nodos de cómputo interconectados. Durante las mediciones, el sistema fue desplegado de manera que todos los nodos ajustaran automáticamente su cantidad de instancias según lo considerado óptimo para mantener un equilibrio entre rendimiento, utilización de recursos y tiempo de respuesta.

En otras palabras, no se fijó una cantidad estática de nodos de cómputo por servicio, sino que el sistema escaló dinámicamente cada componente según la demanda generada por la cantidad de clientes concurrentes y el volumen de procesamiento requerido en cada fase de ejecución.

15.2. Escenarios de prueba

Se midió el comportamiento del sistema con **1**, **2** y **3 clientes** en ejecución simultánea, todos procesando el **dataset reducido** y ejecutando las **cuatro consultas** definidas por la cátedra.

Cada cliente opera de manera aislada dentro de su propia sesión (`session_id`), manteniendo así independencia lógica entre los flujos y permitiendo evaluar la escalabilidad real del sistema frente a cargas concurrentes.

Además se hizo la medición del tiempo total de procesamiento para un único cliente ejecutando las cuatro consultas sobre el **dataset completo**.

15.3. Dataset completo (100 %) - Único cliente

El tiempo medido para la ejecución de las cuatro consultas sobre el dataset completo con un único cliente fue de **12 minutos y 33 segundos**.

15.4. Dataset reducido (30 %) - Múltiples clientes

El tiempo medido para la ejecución de las cuatro consultas sobre el dataset reducido para múltiples clientes fue el siguiente:

- **1 cliente:** El tiempo total de procesamiento fue **3 minutos y 2 segundos**.
- **2 clientes:** El tiempo total de procesamiento fue **8 minutos y 17 segundos**.
- **3 clientes:** El tiempo total de procesamiento fue **10 minutos y 57 segundos**.

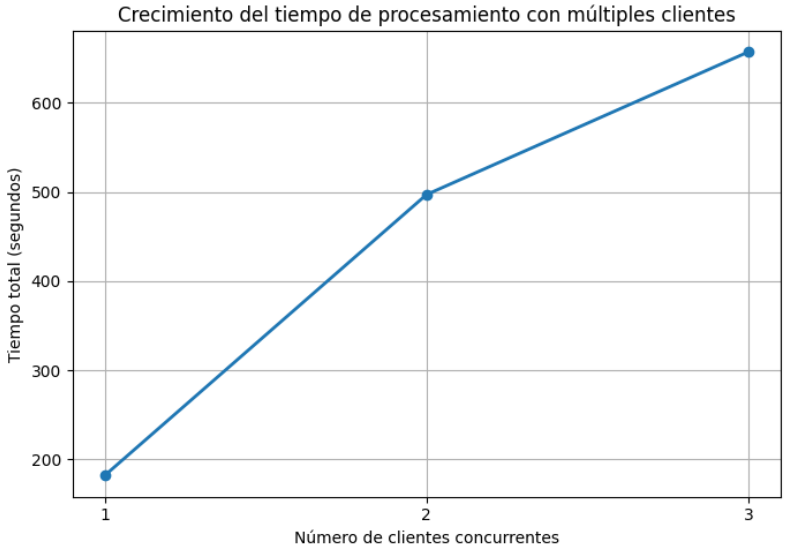


Figura 27: Diagrama de Rendimiento

Podemos notar lo siguiente:

- **Crecimiento no lineal:**
  El tiempo total no aumenta de forma proporcional al número de clientes. Pasar de 1 a 2 clientes incrementa el tiempo en un **173 %**, mientras que pasar de 2 a 3 clientes incrementa en **32 %**.
  Esto indica que los recursos del sistema no escalan linealmente con la carga.
- **Efectos de concurrencia y contención:**
  A medida que más clientes ejecutan consultas en paralelo, el sistema debe compartir recursos. En nuestro sistema distribuido esto genera **sobrecarga de sincronización, bloqueos o esperas en colas de ejecución**, lo que degrada el rendimiento.
- **Tendencia de saturación:**
  El salto entre 1 y 2 clientes es muy pronunciado, pero entre 2 y 3 el aumento es menor. Esto puede sugerir que el sistema llegó a un **punto de uso máximo de recursos**, donde agregar más clientes solo incrementa marginalmente el tiempo de respuesta total (todos esperan en cola).

Conclusión general: A pesar del incremento en el tiempo con más clientes, el sistema muestra un **buen rendimiento general** y mantiene tiempos de respuesta aceptables bajo concurrencia. Esto sugiere que la arquitectura utilizada maneja adecuadamente la carga y ofrece un desempeño estable en escenarios con múltiples clientes.

16. Mecanismos de Control

16.1. Mecanismos de Control de Sincronización

En el diseño del sistema distribuido cuenta con la incorporación de mecanismos de control de sincronización, con el objetivo de evitar problemas asociados a la concurrencia, tales como *race conditions* o *deadlocks*.

Estos mecanismos garantizan un acceso ordenado a los recursos compartidos y la correcta coordinación entre los procesos en ejecución.

16.2. Mecanismos de Control de Señales y Finalización

El sistema incorpora un manejo explícito de señales a fin de asegurar una finalización correcta y segura de los procesos.

En particular, se controla la señal `SIGTERM`, permitiendo un *graceful quit* que libera recursos, cierra todos los archivos abiertos, se asegura de que todos los file descriptors sean cerrados de forma correcta, mediante a los métodos de 'Close' y 'Delete' brindados por el Middleware.

Este mecanismo previene pérdidas de datos, mantiene la consistencia del sistema y asegura el correcto uso de los recursos del sistema operativo.

16.3. Mecanismos de Control de Fallas

Para que el sistema sea robusto, se realizó el desarrollo de diversos mecanismos de control de fallas, los cuales garantizan el determinismo y la confiabilidad del sistema.

En dicha implementación se busca optimizar al máximo estos 2 atributos de calidad, dejando de lado optimizaciones de rendimiento.

Esta decisión de diseño fue tomada para poder controlar la mayor cantidad de escenarios de falla posible, de forma segura y cumpliento con todos los criterios definidos por la cátedra.

Un poco mas adelante en el documento, se encuentra el apartado 'Tolerancia a Fallas', donde se detalla en profundidad todas las implementaciones realizadas.

17. Desafíos al escalar el sistema distribuido

Durante el proceso de escalabilidad del sistema distribuido, se identificaron diversos desafíos técnicos, particularmente en algunos nodos del flujo de procesamiento de datos. Las mayores dificultades se presentaron en los nodos *reducers* (encargados de operaciones de agregación como *sums* y *counts*), en los nodos de ordenamiento (*sorts*) y en los nodos de unión (*joiners*). A continuación se detallan los principales aspectos abordados en cada caso.

17.1. Reducers (Sums y Counts) y Sorts

Los nodos *reducers* y *sorts* representan componentes **stateful**, es decir, mantienen un estado interno durante su ejecución. Este tipo de nodos plantea una mayor complejidad a la hora de diseñar estrategias de escalabilidad, ya que los datos no pueden simplemente distribuirse sin una política clara de particionado que preserve la coherencia del resultado.

A partir de la implementación de la entrega **MultiClient**, se logró una estrategia de escalabilidad efectiva basada en el **shardeo de los nodos** según una clave de partición lógica que permite dividir el cómputo sin afectar la consistencia de los resultados.

Por ejemplo, al calcular el *Total Payment Value (TPV)* de una tienda específica, se puede aplicar un esquema de particionado por `store_id`. De este modo, cada nodo *reducer* procesa únicamente los datos correspondientes a un grupo de tiendas determinado, evitando interferencias y mejorando el rendimiento general del sistema. Esta metodología permitió escalar horizontalmente los nodos *reducers* y *sorts*, manteniendo la integridad de los datos y reduciendo el tiempo total de procesamiento en escenarios de alta concurrencia.

17.2. Joiners

Los nodos *joiners* presentaron un desafío particular al trabajar en entornos con múltiples clientes, especialmente debido a la naturaleza dinámica del flujo de datos en *streaming*. En este contexto, los nodos debían manejar simultáneamente tanto los datos que llegan en tiempo real como la información de nuevas fuentes o clientes que pueden incorporarse durante la ejecución.

Para resolver este problema, se diseñó una arquitectura basada en **hilos concurrentes** dentro del controlador principal, distribuidos de la siguiente manera:

1. Un hilo dedicado a escuchar de forma continua el *stream* de datos entrante, gestionando la transmisión y ejecutando las operaciones de *join* cuando la información requerida está disponible.
2. Un segundo hilo encargado de monitorear la llegada de nuevos conjuntos de datos provenientes de otras tablas o clientes, incorporándolos al flujo de unión en tiempo real.
3. Un tercer hilo que actúa como **coordinador**, responsable de iniciar, supervisar y finalizar los otros dos hilos, garantizando una ejecución ordenada y un cierre seguro del sistema.

Adicionalmente, se implementaron mecanismos de **bufferización y almacenamiento temporal** para retener los datos del *stream* que aún no pueden ser procesados. Esto asegura que, cuando la información complementaria esté disponible, los datos pendientes puedan ser correctamente integrados en el *join* correspondiente sin pérdida ni inconsistencia.

Esta arquitectura permitió alcanzar una ejecución más robusta y eficiente, garantizando la correcta sincronización de los flujos y la consistencia de los resultados, incluso bajo escenarios de concurrencia elevada y múltiples clientes activos.

18. Tests

Con el objetivo de garantizar la correcta operación, consistencia y determinismo del sistema distribuido desarrollado, se implementó un conjunto integral de pruebas (*tests*) que abordan distintos niveles de validación funcional y estructural.

En total, el sistema cuenta con **tres grandes grupos de tests**:

- **Tests de Middleware**
- **Tests de Comparación de Outputs**
- **Tests de Propagación de EOF**

A continuación, se describen los objetivos, el alcance y los criterios de validación de cada grupo de pruebas.

18.1. Tests de Middleware

Estos tests se encargan de verificar la **correcta implementación y funcionamiento del Middleware** desarrollado sobre la biblioteca de comunicación `RabbitMQ`, siguiendo la interfaz y las especificaciones brindadas por la cátedra.

El objetivo principal de este conjunto de pruebas es asegurar que la capa de comunicación entre nodos funcione de manera estable, confiable y conforme al protocolo definido, tanto para el envío como para la recepción de mensajes. Entre los aspectos validados se incluyen:

- Correcta creación y vinculación de colas y *exchanges*.
- Enrutamiento adecuado de mensajes según el tipo de flujo o dataset.
- Manejo de *acknowledgements* y confirmaciones de entrega.
- Comportamiento esperado del middleware bajo condiciones de carga y escalado.
- Gestión de excepciones.

Estos tests fueron desarrollados originalmente para la entrega **"Escalabilidad, Middleware y Coordinación de Procesos"**, y sirvieron de base para validar la estabilidad de la infraestructura de mensajería del sistema.

18.2. Tests de Comparación de Outputs

El segundo conjunto de pruebas está orientado a la **validación funcional de los resultados** obtenidos por el sistema distribuido, comparando las salidas producidas por las consultas con los resultados esperados.

Dado que en un entorno distribuido las tareas se dividen entre nodos escalables sin garantizar un orden específico en la entrega de resultados, fue necesario diseñar técnicas de comparación **insensibles al orden** para validar el correcto comportamiento determinístico del sistema. De esta forma, las pruebas se centran en comprobar la equivalencia semántica de los resultados más allá del orden particular de las filas.

Validación por consulta.

- **Query 1:** Verifica que la cantidad de líneas generadas sea exactamente la misma que la esperada, confirmando que el volumen de resultados es correcto.
- **Query 2:** Compara tanto la cantidad de filas como el contenido de cada una, sin considerar el orden en que fueron entregadas. De este modo, se valida que los resultados sean completos y correctos, independientemente del orden de procesamiento distribuido.
- **Query 3:** Aplica los mismos criterios que en la Query 2, validando que las filas generadas contengan la misma información y que no haya pérdidas ni duplicados, aún cuando el orden de emisión varíe entre ejecuciones.
- **Query 4:** Verifica que:
  - La cantidad total de filas coincida con la esperada.
  - Exista la misma cantidad de respuestas por cada tienda.
  - Los clientes identificados como los más compradores por tienda coincidan en cantidad de compras.

  No se consideran las diferencias en nombres o fechas de nacimiento, ya que en caso de empates el orden de los tres máximos puede variar entre ejecuciones. Lo relevante es que los valores reportados correspondan a los tres mayores por tienda, garantizando así la corrección lógica de la consulta.

Estos tests fueron desarrollados inicialmente para la entrega **"Escalabilidad, Middleware y Coordinación de Procesos"** y luego **mejorados para la entrega "MultiClient"**.

18.3. Tests de Propagación de EOF

Finalmente, se implementó un conjunto de pruebas específicas para validar el **mecanismo de propagación de mensajes** `EOF` dentro del sistema *MultiClient*.

El objetivo de estos tests es comprobar que:

- Cada controlador reciba correctamente los `EOF` asociados a los flujos que le corresponden, diferenciados por `session_id`.
- La propagación de `EOF` se realice de forma completa y ordenada hacia los nodos sucesores en la cadena de procesamiento.
- No existan pérdidas, duplicados ni propagaciones cruzadas entre sesiones.

De esta manera, se valida que el mecanismo de cierre de flujos por sesión funcione correctamente y que el sistema distribuido finalice el procesamiento de cada cliente de forma independiente y controlada, sin afectar el estado de los demás flujos activos.

Estos tests fueron desarrollados como parte de la entrega **"MultiClient"**, a pedido de la cátedra, para asegurar la correcta implementación del sistema de detección y propagación de fin de flujo dentro del entorno multiusuario.

19. Tolerancia a Fallas

19.1. Introducción

En esta sección se detallan los mecanismos implementados en el Sistema Distribuido para tolerar distintos escenarios de falla. El objetivo principal de estas implementaciones es garantizar la **consistencia** y la **robustez** del sistema, priorizando estos atributos por sobre la eficiencia en términos de rendimiento.

En cuanto a la arquitectura general, no se presentan modificaciones en la topología previamente documentada. El único agregado corresponde a la incorporación de un conjunto de nodos denominados *Health Checkers*, encargados de consultar de forma recurrente el estado de cada nodo del sistema, con el fin de validar su correcto funcionamiento o, en caso de detectar fallas, iniciar procesos de recuperación para restablecer las instancias caídas.

19.2. Health Checkers

Los nodos *Health Checkers* se encargan de monitorear continuamente que todos los nodos encargados del procesamiento para la generación de resultados se encuentren activos. Este mecanismo está implementado mediante múltiples instancias dispuestas en una topología en anillo, con el fin de garantizar la tolerancia a fallas y la alta disponibilidad del propio sistema de monitoreo.

Uno de los nodos cumple el rol de **Líder**, siendo el responsable de coordinar las validaciones periódicas. En caso de que el líder falle, se ejecuta automáticamente un algoritmo de elección que designa un nuevo líder, el cual retoma las tareas del anterior.

El proceso de verificación consiste en el envío periódico de mensajes de tipo *Ping* a cada nodo del sistema. Si el nodo responde con un *ACK*, se considera activo; En caso contrario, el líder inicia el proceso de recuperación de esa instancia.

El servidor central también participa de este esquema de *heartbeat*. Por cada cliente que levanta, registra el identificador de sesión en un archivo persistente. Si el servidor se cae y luego se reinicia, utiliza esa información para notificar a todos los controladores que deben limpiar el estado asociado a las sesiones previamente activas, evitando que queden residuos de consultas incompletas. De forma análoga, cuando el servidor detecta que un cliente finaliza o se desconecta definitivamente, envía una orden de limpieza sólo para esa sesión, de modo que se liberen los recursos correspondientes en todos los nodos sin afectar las consultas del resto de los clientes.

19.2.1. Topología del Sistema de Health Checkers

A continuación se dejan unos diagramas que representan:

1. La topología de anillo de los Health Checkers.
2. La interacción entre el líder y los otros nodos Health Checkers.
3. La interacción entre el líder y los nodos del Sistema Distribuido.

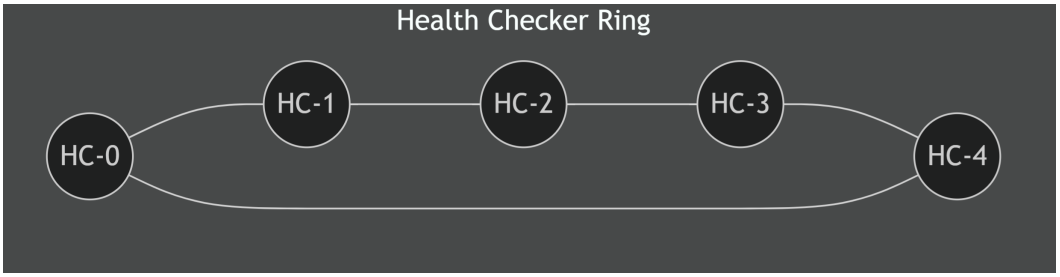


Figura 28: Topología de anillo de los Health Checkers

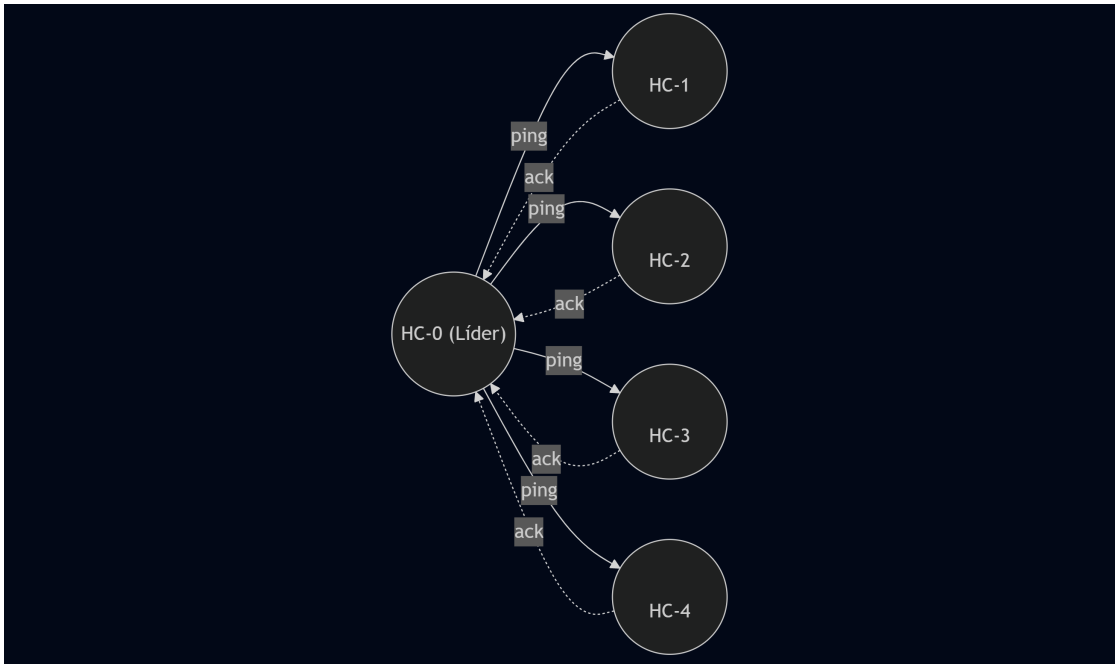


Figura 29: Pings y ACKs que manda el Líder a los otros Health Checkers

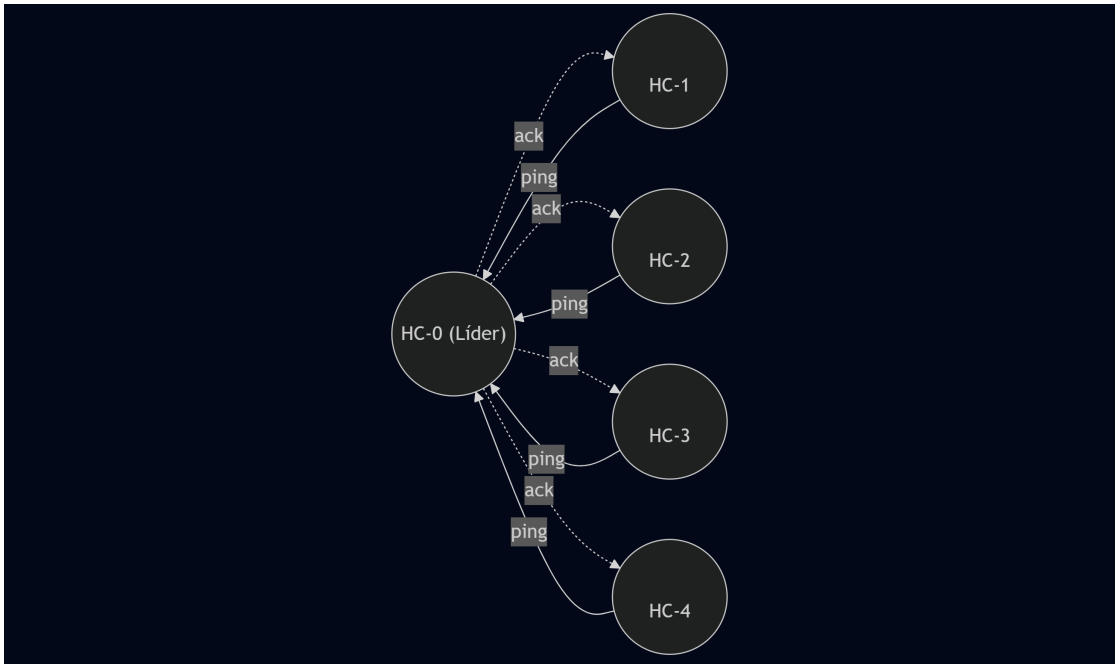


Figura 30: Pings y ACKs que mandan los Health Checkers al Líder

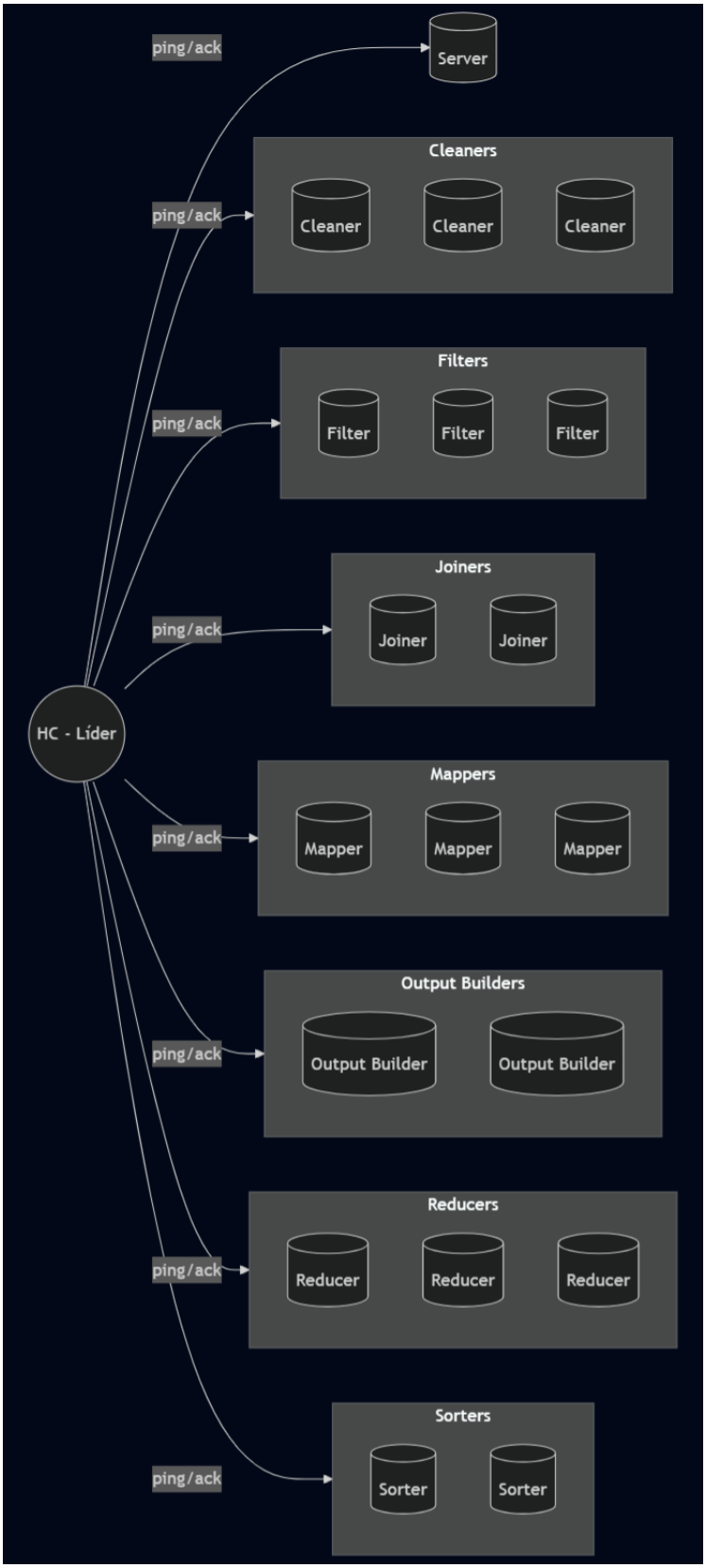


Figura 31: Pings y ACKs que manda el Líder a los nodos del Sistema Distribuido

19.3. Cambio de comportamiento en los nodos

Se introdujeron modificaciones significativas en el funcionamiento de todos los nodos del sistema que consumen mensajes desde las colas de *RabbitMQ*. El objetivo de estos cambios es fortalecer la robustez del sistema ante fallas parciales y garantizar la consistencia de los datos en todo momento.

Cada nodo, al recibir ("popear") un mensaje desde la cola correspondiente, ejecuta su lógica de procesamiento, le envía el mensaje a la siguiente cola de el próximo nodo, escribe el mensaje o el estado actualizado en su volumen persistente, y finalmente envía el *ACK* a *RabbitMQ*. Este orden de operaciones —procesamiento, envío, persistencia y confirmación— es esencial para evitar duplicaciones o pérdidas de mensajes ante fallas inesperadas.

Gracias a la semántica de confirmación de *RabbitMQ*, si un mensaje es consumido pero no se envía el *ACK*, dicho mensaje se reencola automáticamente al inicio de la cola. De esta forma, ante una caída, el nodo podrá retomar su ejecución desde el último estado persistido o reprocesar el mensaje que quedó pendiente, garantizando así la consistencia del flujo.

Observación: Todos los archivos generados se almacenan en volúmenes persistentes asociados a los contenedores de cada nodo. Esto permite que, incluso tras una caída o reinicio, el nodo conserve su información y pueda recuperar su estado previo sin pérdida de datos.

19.3.1. Nodos Stateless

Los nodos *Stateless* procesan mensajes de manera independiente, sin mantener un estado acumulativo entre ellos. Su flujo de ejecución se compone de las siguientes etapas:

1. Recepción del mensaje desde la cola de entrada.
2. Procesamiento del mensaje (limpieza, filtrado u otra operación específica).
3. Envío del resultado al siguiente nodo.
4. Escritura del mensaje procesado en un archivo local del volumen persistente, para disponer de una copia de respaldo.
5. Envío del *ACK* a *RabbitMQ*.

La escritura en disco se realiza **únicamente al final del proceso**, luego de haber enviado el mensaje. Este orden es crítico, ya que evita que un mensaje pueda ser reenviado o computado más de una vez, garantizando la consistencia global del sistema.

19.3.2. Nodos Stateful (Joins)

Los nodos *Joins* presentan un comportamiento intermedio entre los nodos con y sin estado. Durante su ejecución inicial, reciben información de configuración proveniente del servidor, la cual se almacena en su volumen local y se utiliza de manera incremental al procesar los siguientes mensajes.

Una vez recibida toda la información del servidor, el nodo pasa a comportarse de forma análoga a un nodo *Stateless*: Procesa, envía, registra y confirma cada mensaje de manera independiente.

Para facilitar la recuperación ante fallas, una vez completada la fase de inicialización, el nodo *Join* registra un indicador distintivo (Por ejemplo, una marca de finalización) en su archivo local. De esta forma, si el nodo se reinicia, puede detectar fácilmente si ya recibió todos los datos del servidor o si aún resta información por almacenar.

Como optimización adicional, los *Joiners* persisten la información base separada por sesión. Cada consulta mantiene su propio archivo de estado, lo que permite recomponer de forma localizada la información de un cliente en caso de caída, y evita que el estado de una sesión afecte o contamine el de las demás.

19.3.3. Nodos Stateful con estado acumulativo (Sorts, Sums, Counts)

Funcionamiento inicial del nodo

Los nodos con estado acumulativo —como *Sorts*, *Sums* y *Counts*— mantienen información agregada entre mensajes. Su comportamiento fue rediseñado para maximizar la confiabilidad y optimizar la recuperación ante fallas.

Cada nodo sigue el siguiente flujo:

1. Recibe un mensaje desde la cola correspondiente.
2. Actualiza su estado interno con la información del mensaje (operación de cómputo o agregación).
3. Escribe el mensaje procesado en su archivo local (uno por cada nodo antecesor).
4. Envía el *ACK* a *RabbitMQ*.
5. Continúa con el siguiente mensaje.

Cada cierto número de mensajes n, configurable según la carga del nodo o la probabilidad de falla estimada, se realiza un **backup o checkpoint** del estado actual. Este se guarda en un archivo separado dentro del mismo volumen, y actúa como punto de restauración ante fallas.

Envío del reporte generado hacia el próximo nodo

Finalmente, una vez completado el procesamiento de todos los mensajes correspondientes a una consulta o cliente, el nodo realiza el cálculo del estado final acumulado antes de enviar cualquier resultado al siguiente nodo de la secuencia. Ese estado final se persiste en el archivo de **backup/checkpoint**, garantizando la existencia de una versión consistente desde la cual se puedan regenerar los mensajes de salida.

Posteriormente, se procede al envío del estado al nodo siguiente. Entre cada mensaje que se emite (Por ejemplo, cuando el resultado debe fragmentarse en varios mensajes), el nodo actualiza y persiste el estado intermedio asociado a esa salida. De esta forma, si se produce una caída durante el envío, al reiniciarse puede reconstruir exactamente los mismos mensajes a partir del estado almacenado, sin introducir variaciones en el contenido.

Una vez completado el envío de todos los fragmentos, el nodo registra en su volumen una marca o indicador que señala que el estado correspondiente a esa consulta o cliente ya fue transmitido correctamente. Esto permite que, en caso de una caída y posterior recuperación, el nodo identifique que el resultado ya fue enviado y evite duplicaciones o recomputaciones innecesarias.

En los casos en que el envío del estado final requiera dividir la información en múltiples mensajes, se aplica la misma lógica de procesamiento y persistencia que para el resto de los mensajes: cada fragmento enviado se genera de forma determinista a partir del estado persistido y se escribe previamente en un volumen dedicado al manejo de dichos estados parciales. Así, si un mensaje debe reenviarse por un fallo en la confirmación, el contenido es idéntico al original y puede ser detectado como duplicado por el nodo receptor usando el `msg_uuid` o los registros locales.

Consideraciones

Para mantener el uso eficiente del almacenamiento, tras cada backup se eliminan o marcan los registros previos como *COMMIT*, indicando que los mensajes anteriores ya fueron computados y persistidos en el estado actual.

Cada nodo mantiene archivos separados por cada uno de sus antecesores. Esto permite acceder de manera directa al último mensaje procesado por cada fuente, manteniendo la trazabilidad y la consistencia del flujo completo. Se asume además que todas las escrituras en disco son **atómicas**, garantizando que sólo existen dos posibles resultados: escritura completa o ausencia total de la misma, eliminando la posibilidad de estados intermedios inconsistentes.

19.3.4. Procesos e Hilos por nodo

En cuanto a la arquitectura interna de cada nodo, se implementó un modelo basado en múltiples hilos y/o procesos de ejecución para optimizar el procesamiento y posibilitar la gestión de mensajes en paralelo.

A continuación se describen mediante a diagramas los distintos modelos implementados según el tipo de nodo:

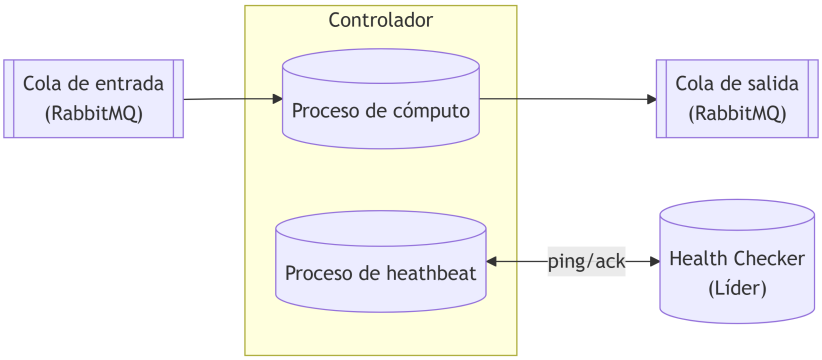


Figura 32: Procesos en los Controladores

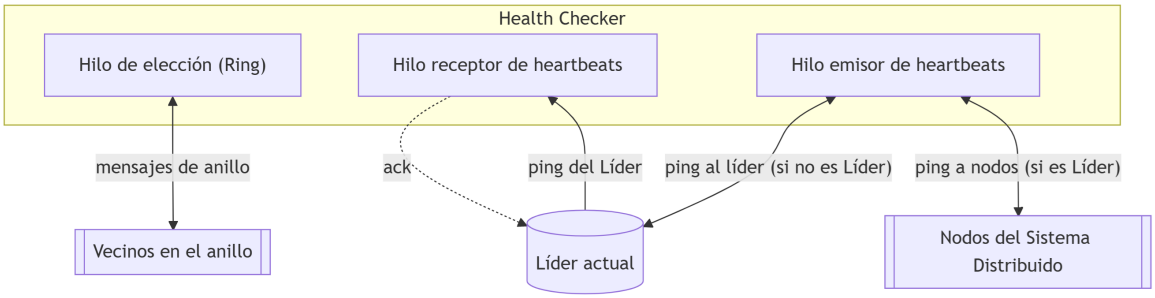


Figura 33: Procesos e hilos en los Health Checkers

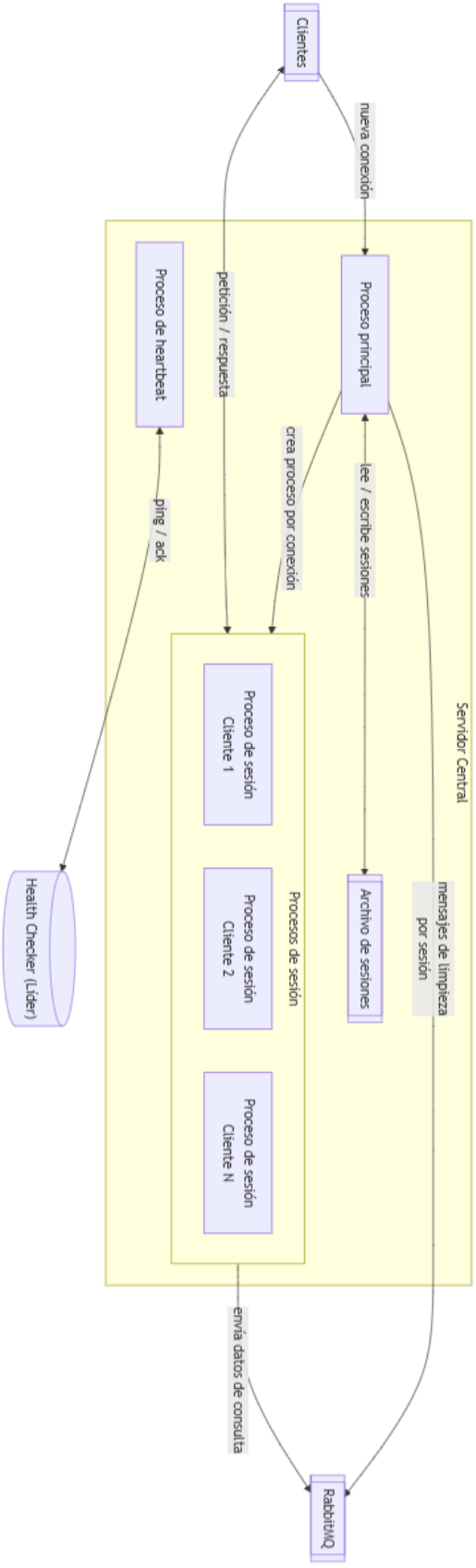


Figura 34: Procesos en el Servidor Central

19.4. Cambios en el protocolo

La nueva implementación introduce un encabezado (*header*) ligero que se agrega sobre el paquete existente, siguiendo un enfoque análogo al apilado de cabeceras entre capas de enlace y red en redes tradicionales. Este encabezado incorpora dos campos adicionales:

- **UUID del mensaje** (`msg_uuid`): Generado por el servidor emisor inicial y *consistente durante toda la vida del mensaje* a través de la tubería de procesamiento. En los nodos que generan reportes/outputs finales, se genera un *nuevo* UUID para el artefacto resultante, el cual se usa a partir de ahí para su trazabilidad.
- **Identificador del nodo origen** (`src_node_id`): Indica el nodo que produjo el mensaje. Se utiliza en el receptor para dirigir la persistencia al volumen/archivo correspondiente a ese origen, preservando trazabilidad y consistencia.

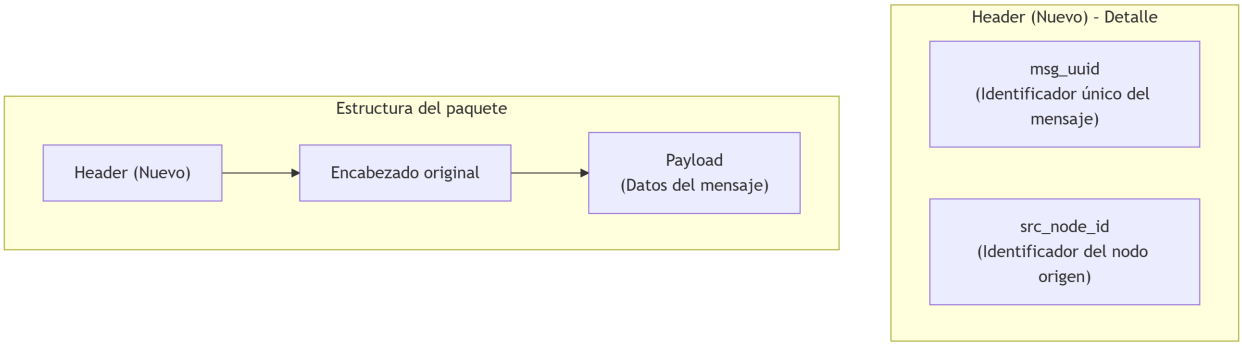


Figura 35: Estructura de los mensajes

Balanceo determinista basado en hash Los nodos que antes distribuían carga con *Round Robin* (contador) migran a **hash por** `msg_uuid`. Este cambio evita desbalances e inconsistencias cuando un nodo cae y reinicia su contador:

- Con *Round Robin*, un reinicio cambia el punto del ciclo y el mismo mensaje podría enrutarse a un destino distinto.
- Con **hash(msg_uuid)**, a igual UUID se obtiene el mismo destino (consistentemente), incluso si el nodo se reinicia. Esto preserva afinidad de mensaje y localización esperada del estado.

Compatibilidad y consistencia El encabezado es aditivo, no rompe el formato del payload previo. La escritura local sigue siendo atómica, y la lógica de ACK de *RabbitMQ* permanece inalterada: el ACK sólo se emite cuando el procesamiento y la persistencia asociada se completan, manteniendo idempotencia extremo a extremo.

19.5. Escenarios de Falla

A continuación se detallan los distintos escenarios de falla contemplados durante el diseño de los mecanismos de tolerancia y recuperación del sistema. En todos los casos, se garantiza tanto la continuidad operativa ante la caída de uno o más nodos, como la consistencia de los resultados y del flujo de mensajes.

19.5.1. Falla en los Health Checkers

Ante la caída de algún nodo del sistema de *Health Checkers*, pueden presentarse dos situaciones:

- **Falla del líder:** Se ejecuta un proceso de elección para designar un nuevo líder, el cual se reconecta con los nodos activos del sistema y retoma las tareas pendientes.
- **Falla de un nodo no líder:** La instancia se remueve temporalmente del anillo para mantener la coherencia de la topología.

En ambos casos, se intenta levantar nuevamente la instancia caída para restablecer el nivel de redundancia definido y mantener la alta disponibilidad del sistema.

19.5.2. Falla en los Nodos Stateless y Nodos Join

El comportamiento ante fallas es similar al de los nodos *Join*:

- **Falla antes del *ACK*:** *RabbitMQ* reencola el mensaje, que será procesado nuevamente cuando el nodo se recupere. Como esta es la última operación antes de popear otro mensaje, no existe otro instante de falla a considerar.

19.5.3. Falla en los Nodos Statefull

Dependiendo del tipo de nodo, los mecanismos de recuperación difieren:

Nodos con estado acumulativo (Sorts, Sums, Counts):

- **Si el nodo falla antes de procesar el mensaje**: El mensaje se reencola automáticamente por *RabbitMQ*, y al reiniciarse, el nodo lo vuelve a procesar desde su archivo o desde el backup más reciente.
- **Si falla después de procesar el mensaje pero antes del *ACK***: El mensaje se reencola, y al recuperarse, el nodo detecta en su registro que el mensaje ya fue computado, evitando duplicaciones.
- **Si falla durante la acumulación**: Al reiniciarse, el nodo restaura su estado desde el último backup disponible y continúa procesando desde el siguiente mensaje.
- **Si falla en el proceso de envío del reporte hacia el próximo nodo**: Al volver a despertar, se retoma desde el último mensaje enviado, para garantizar la consistencia del sistema, evitando mensajes duplicados. Para identificar los mensajes, el nodo que genera el reporte asigna los IDs correspondientes, haciendose responsable de su correctitud.

19.5.4. Fallas Desestimadas

Durante el análisis se identificaron otros posibles tipos de falla:

- Falla en el **Servidor Central**.
- Falla en los **Clientes**.
- Falla en el **Middleware RabbitMQ**.
- Falla en **todos los Health Checkers simultáneamente**.

El manejo para generar la recuperación de las consultas, ante alguno de estos escenarios fue desestimado en esta versión del sistema, dado que implicaría definir contratos adicionales con los clientes, considerar hipótesis extremadamente remotas y un nivel de complejidad que excede los requerimientos del presente enunciado.

Sin embargo, se documentan estos casos para su consideración en futuras versiones del sistema.

De igual manera, la ocurrencia de alguno de los tipos de falla mencionados se encuentran controlados de forma que el Sistema Distribuido pueda seguir su ejecución, luego de controlar las inconsistencias (Por ejemplo, vaciando los estados de los controladores para reiniciar consultas).

19.5.5. Mecanismos adicionales de consistencia

Para garantizar la coherencia incluso en escenarios de fallas simultáneas:

- El nodo siguiente a uno caído valida que los mensajes recibidos no sean duplicados, comparando el último mensaje recibido con el nuevo.
- Los nodos que mantienen estado lo recomponen desde su último backup; los que no, reprocesan los mensajes pendientes desde la cola.
- Dado que *RabbitMQ* no pierde mensajes durante las caídas, el sistema mantiene su integridad global.

19.6. Sistema de Generación de Errores

Con el objetivo de validar los mecanismos de tolerancia a fallas implementados en el sistema, se desarrolló un **Sistema de Generación de Errores**, diseñado para simular distintos escenarios de falla de manera controlada o aleatoria.

Este sistema cuenta con dos funcionalidades principales:

- **Inyección manual de fallas en instancias específicas:** Esta herramienta permite provocar la caída de instancias puntuales del sistema de forma controlada. Su propósito principal es facilitar las pruebas durante el desarrollo, permitiendo analizar el comportamiento del sistema ante la falla y posterior recuperación de nodos concretos, así como verificar la consistencia de los datos y estados luego de cada reinicio.
- **Módulo de fallas aleatorias tipo *Chaos Monkey*:** Inspirado en las prácticas de ingeniería del caos, este módulo genera caídas y perturbaciones de manera aleatoria en distintas partes del sistema. De esta forma, se evalúa el comportamiento integral del sistema distribuido bajo condiciones de falla no determinísticas, más cercanas a un entorno real de producción. Este enfoque permite demostrar la robustez global y la correcta recuperación de los componentes frente a fallas imprevistas.

La señal que envía el sistema de generación de fallas a los nodos vivos es **SIGKILL**.

En conjunto, ambas funcionalidades permiten validar tanto el funcionamiento individual de los mecanismos de tolerancia a fallas como su efectividad a nivel global, asegurando la resiliencia y estabilidad del sistema frente a distintos tipos de incidentes.

19.7. Mediciones de Rendimiento (Con Fallas)

Con el objetivo de evaluar el comportamiento y la eficiencia del sistema distribuido, se realizaron diversas mediciones de rendimiento bajo distintos escenarios y volúmenes de datos.

El propósito de este análisis es determinar la capacidad del sistema para mantener la consistencia de los resultados y la estabilidad de los tiempos de respuesta ante la presencia de fallas controladas o aleatorias.

Las pruebas se dividen en dos grupos principales: Aquellas realizadas sobre un **dataset reducido**, orientadas a validar la consistencia frente a fallas aleatorias, y aquellas efectuadas sobre el **dataset completo**, destinadas a medir la performance global del sistema con y sin eventos de falla.

19.7.1. Dataset Reducido: Consistencia frente a caídas aleatorias

En este conjunto de pruebas se busca demostrar que el sistema mantiene resultados consistentes ante la ocurrencia de fallas aleatorias, validando así la efectividad de los mecanismos de tolerancia a fallas implementados.

Herramienta de evaluación Para la ejecución de estas pruebas se empleará el **Sistema de Generación de Errores** previamente descripto, en su modo de funcionamiento aleatorio tipo *Chaos Monkey*.

Esta herramienta permitirá simular caídas no determinísticas de distintas instancias del sistema, abarcando tanto nodos *stateless* como *stateful* y componentes del anillo de *Health Checkers*.

Parámetros de prueba Los principales parámetros de configuración de las pruebas son los siguientes (valores a definir en futuras iteraciones):

- Frecuencia de generación de fallas: `120 segundos`.
- Duración total de la prueba: `11 minutos`.
- Cantidad de nodos involucrados: `63`.
- Volumen de datos procesado: `Dataset reducido`.

Criterios de evaluación Los criterios utilizados para validar la consistencia y estabilidad del sistema serán:

- Coincidencia de los resultados obtenidos con los esperados (control de consistencia).
- Tiempo promedio de recuperación ante fallas.
- Número de tareas reintentadas correctamente.
- Impacto en la latencia promedio del sistema.

Resultados esperados Se espera observar resultados consistentes en la salida final, independientemente de las caídas aleatorias introducidas, y una recuperación rápida de las instancias afectadas.

Los valores numéricos y métricas específicas serán completados una vez finalizada la ejecución de las pruebas.

19.7.2. Dataset Completo: Monitoreo de performance con y sin fallas

Este conjunto de pruebas se orienta a analizar el rendimiento del sistema distribuido bajo carga completa, tanto en condiciones estables (sin fallas) como durante la inyección de fallas aleatorias, con el fin de evaluar la degradación del desempeño.

Herramienta y entorno de medición Las mediciones se realizarán utilizando herramientas de monitoreo de recursos y trazabilidad, tales como `htop`.

Esta permitirá recolectar métricas en tiempo real sobre el consumo de CPU, memoria, tráfico de red y latencia promedio de las consultas distribuidas.

Parámetros de prueba Los principales parámetros definidos para este escenario son los siguientes (valores a completar):

- Frecuencia de generación de fallas: `120 segundos`.
- Volumen de datos procesado: `Dataset completo`.
- Cantidad total de nodos en ejecución: `63`.
- Cantidad de consultas simultáneas: `5 (1 Cliente)`.
- Duración de la medición: Aproximadamente se estiman `50 minutos`, no se realizó la medición en su totalidad para evitar daños en el disco de la computadora.

Criterios de evaluación Las métricas principales a analizar serán:

- Tiempo promedio de procesamiento por consulta.
- Throughput del sistema.
- Utilización promedio de recursos (CPU, memoria, red).
- Variación del rendimiento ante la presencia de fallas.

Resultados esperados Se espera observar un desempeño estable del sistema bajo carga completa en condiciones normales, con una degradación controlada durante las pruebas con fallas aleatorias.

Los resultados cuantitativos y gráficos comparativos serán incorporados una vez completadas las mediciones experimentales.

20. Cronograma teórico del desarrollo

20.1. Diseño

Previo al inicio del desarrollo efectivo del diseño, el equipo realizó una instancia de planificación para organizar la distribución de responsabilidades. El objetivo fue asegurar que cada integrante asumiera un conjunto de tareas equilibrado, alineado tanto con sus fortalezas como con las necesidades del proyecto.

20.2. Escalabilidad

El desarrollo de esta entrega cuenta con un tiempo estimado de 2.5 semanas. Durante la primera semana se realizará un análisis detallado de los criterios de escalabilidad, los requisitos técnicos y el alcance de la entrega, así como la planificación y distribución de tareas entre los integrantes del equipo.

En las semanas posteriores se definirán y ejecutarán las labores específicas de cada integrante, las cuales se documentarán en versiones futuras de este informe.

20.3. Multi-Client

El desarrollo de esta entrega cuenta con un tiempo estimado de 2 semanas. La primera semana estará destinada al análisis de los criterios, requisitos y alcance de la funcionalidad de múltiples clientes, junto con la planificación y asignación de tareas a cada integrante del equipo.

En la semana restante se abordará la ejecución de las labores planificadas, que se detallarán en próximas versiones de este documento.

20.4. Paper

El desarrollo de esta entrega cuenta con un tiempo estimado de 1.5 semanas.

Durante la primera semana se llevará a cabo la lectura y análisis del Paper acerca de “Chubby” (Servicio de bloqueo distribuido de Google), abarcando la comprensión de sus conceptos principales, su arquitectura y las conclusiones relevantes para la presentación

Asimismo, se realizará la redacción del documento resumen correspondiente, en el cual se sintetizarán los aspectos técnicos y teóricos más destacados.

El tiempo restante (media semana) será destinado a la preparación de la clase expositiva, incluyendo la elaboración del material de apoyo (Presentación) y la práctica para la exposición de 15 minutos.

20.5. Tolerancia

El desarrollo de esta entrega cuenta con un tiempo estimado de 6.5 semanas.

La primera semana se dedicará al análisis de criterios, requisitos de tolerancia a fallos y alcance de la entrega, además de la planificación y distribución de responsabilidades entre los integrantes.

En la segunda semana se va a pactar una reunión con el corrector, para validar las soluciones planteadas con el corrector.

Una vez sean validadas las implementaciones diseñadas por el corrector, las semanas siguientes estarán orientadas a la implementación progresiva de los mecanismos planificados.

En los últimos días se va a realizar un testeo intensivo del Sistema Distribuido, buscando dejarlo lo mas libre de bugs posible, además de generarse la documentación final del proyecto.

21. Cronograma real del desarrollo

21.1. Diseño (2 semanas)

Durante la **primera semana**, el grupo se enfocó en interpretar en profundidad los requisitos planteados en el enunciado, analizar la lógica de las consultas y definir el diseño conceptual del sistema. Asimismo, en esta etapa se realizó la repartición de tareas entre los integrantes, de manera de optimizar el tiempo de ejecución restante.

En la **segunda semana**, cada integrante se abocó a las actividades asignadas, según el siguiente detalle:

Luciano

- Vista Lógica.
- Vista de Desarrollo.
- Vista Física.
- Vista de Procesos.
- Diseño inicial/conceptual del Middleware.
- Diagrama de Paquetes.

Axel

- Escenarios de uso (Casos de Uso).
- Diagrama de Secuencia.
- Diagrama de Actividades.
- Diagrama de Robustez.
- DAG (Directed Acyclic Graph).

Felipe

- Redacción de las definiciones iniciales.
- Creación de los cronogramas.
- Descripción detallada de la ejecución de las consultas.
- Explicaciones introductorias de los mecanismos de control y protocolos.
- Unión de todas las documentaciones generadas, en el informe entregable.

21.2. Middleware y Coordinación de Procesos (2.5 semanas)

Durante la **primera semana**, el grupo se enfocó en interpretar en profundidad los requisitos planteados en el enunciado, analizar posibles implementaciones para las consultas solicitadas, hacer las correcciones de diseño necesarias, pactar protocolos e identificar bibliotecas necesarias para el desarrollo.

En la **segunda semana**, cada integrante se abocó a las actividades asignadas, detalladas a continuación:

Luciano

- Diseño funcional integral de colas y exchanges en el sistema para la topología diseñada.
- Investigación y selección de la biblioteca para la comunicación mediante RabbitMQ.
- Investigación y selección de la biblioteca para la serialización/deserialización de mensajes.
- Implementación del Middleware.
- Creación de los tests unitarios para validar funcionamiento del Middleware.
- Establecimiento de conexiones entre Cliente, Servidor y Middleware.
- Redacción de protocolos de comunicación.
- Integración del Middleware con los controladores.
- Construcción del ecosistema de nodos de cómputo distribuido mediante a Dockerfiles.

Axel

- Diseño funcional integral de colas y exchanges en el sistema para la topología diseñada.
- Implementación de los controladores:
  - Join.
  - Sorts.
  - Maps.
  - Filters.
- Integración de los controladores con el Middleware.
- Construcción del ecosistema de nodos de cómputo distribuido mediante a Dockerfiles.
- Medición del rendimiento del sistema.

Felipe

- Diseño funcional integral de colas y exchanges en el sistema para la topología diseñada.
- Implementación de los controladores:
  - Cleaners.
  - Output Builders.
  - Reduces.
- Integración de los controladores con el Middleware.
- Redacción de la nueva documentación del sistema.
- Relevamiento y control de los mecanismos de cierre 'graceful' de los nodos de cómputo.

21.3. Multi Client (2 semanas)

Durante la **primera semana**, repitiendo la operativa realizada en entregas previas, el grupo se enfocó en interpretar en profundidad los requisitos planteados en el enunciado, analizar posibles implementaciones para las consultas solicitadas, hacer las correcciones de diseño necesarias, pactar protocolos e identificar bibliotecas necesarias para el desarrollo.

En la **segunda semana**, cada integrante se abocó a las actividades asignadas, detalladas a continuación:

Luciano

- Planificación y diseño de la arquitectura multi cliente.
- Implementación de la lógica multi cliente adaptada al Servidor.
- Implementación de la lógica multi cliente adaptada a los Controladores Statefull.
- Redacción del nuevo protocolo de comunicación multi cliente.
- Refactorización de los controladores para soportar múltiples sesiones concurrentes.
- Adaptación del sistema para configurar la escalabilidad en base al archivo '.env'.

Axel

- Planificación y diseño de la arquitectura multi cliente.
- Implementación de la lógica multi cliente adaptada al Cliente.
- Implementación de la lógica multi cliente adaptada a los Controladores Stateless.
- Refactorización de los controladores para soportar múltiples sesiones concurrentes.
- Mediciones de rendimiento para múltiples clientes concurrentes.
- Redacción de datasets reducidos para la demostración del sistema en la defensa.

Felipe

- Planificación y diseño de la arquitectura multi cliente.
- Análisis de puntos escalables para optimización del rendimiento.
- Redaccion de la nueva documentación del sistema.
- Redacción de los nuevos tests de validación funcional de outputs y propagación de EOF.
- Redacción de la presentación para la defensa de la entrega.
- Relevamiento y control de cumplimiento de los criterios de aceptación.

21.4. Paper (1.5 semanas)

Para esta entrega, en la **primera semana**, todos los integrantes del grupo realizaron la lectura obligatoria sobre el Paper brindado. Una vez terminada la misma, se pasó a la redacción del resumen con los conceptos principales que se identificaron para la exposición.

En la **media semana** restante, se realizó la presentación para la misma, y se practicó varias veces la muestra a realizar.

Luciano

- Lectura completa del Paper.
- Distribución de la exposición.
- Práctica para la muestra.

Axel

- Lectura completa del Paper.
- Redacción del resumen.
- Práctica para la muestra.

Felipe

- Lectura completa del Paper.
- Creación de la presentación.
- Práctica para la muestra.

21.5. Tolerancia a fallas (6.5 semanas)

Durante la **primera semana**, siguiendo con la metodología de trabajo adoptada, el grupo se enfocó en interpretar en profundidad los requisitos planteados en el enunciado, diseñar posibles implementaciones para diversos escenarios de falla, hacer las modificaciones de arquitectura necesarias, pactar protocolos e identificar bibliotecas necesarias para el desarrollo.

En la **segunda semana**, se realizó la validación de las decisiones de diseño tomadas con el corrector, para luego dar paso a la implementacion de los mecanismos ideados.

Durante las siguientes **tres semanas** se implementaron todos los sistemas de tolerancia a fallas pactados.

Para finalizar, en la última **media semana**, se realizó el testo integral el sistema y la documentación final, dando así el cierre total al desarrollo del Sistema Distribuido en su completitud.

Luciano

- Planificación y diseño de la arquitectura tolerante a fallas.
- Rediseño del protocolo de comunicación.
- Implementación de los mecanismos de tolerancia a fallas en los controladores.
- Integración de los 'Health Checkers' con el Sistema Distribuido.
- Implementación de la nueva lógica de envío de mensajes a próximos controladores.
- Agregado de la generación de UUID de los mensajes en el Servidor.
- Verficación de control ante posible duplicación de mensajes.

Axel

- Planificación y diseño de la arquitectura tolerante a fallas.
- Implementación de los mecanismos de tolerancia a fallas en los controladores:
- Testing extensivo del sistema ante fallas.
- Medición del rendimiento del sistema con tolerancia a fallas.
- Investigación de atomocidad en operaciones de archivos.
- Implementación de la reconstrucción de estados de los controladores Statefull ante caídas.
- Adaptación de los controladores a los cambios de protocolo.

Felipe

- Planificación y diseño de la arquitectura tolerante a fallas.
- Redacción de la documentación final del sistema (Añadidos de tolerancia a fallas).
- Redacción de la presentación para la defensa final.
- Implementación de la herramienta 'Chaos-Monkey' para la generación de fallas controladas.
- Implementación del algoritmo de elección de lider para los 'Health Checkers'.
- Implementación inicial del funcionamiento de los 'Health Checkers'.
- Relevamiento de todos los criterios de corrección y aceptación del sistema.