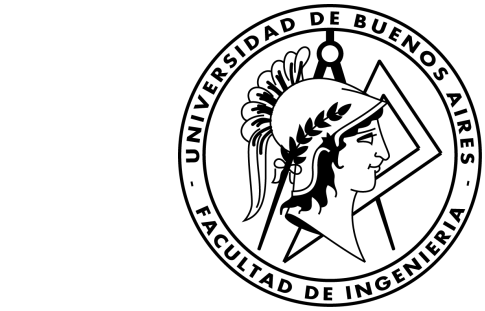

Trabajo Práctico: Tolerancia a Fallos

[TA050] Sistemas Distribuidos I
Segundo cuatrimestre de 2025

Alumno	Nombre y apellido	Padrón	Mail
	Baratta, Facundo	104886	fbaratta@fi.uba.ar
	Kim, Daniel	107188	dakim@fi.uba.ar
	Koo, Hangyeol	108401	hkoo@fi.uba.ar

Fecha de entrega: 4/12/2025

Índice

1. Diagramas

1.1. DAG (Directed Acyclic Graph)

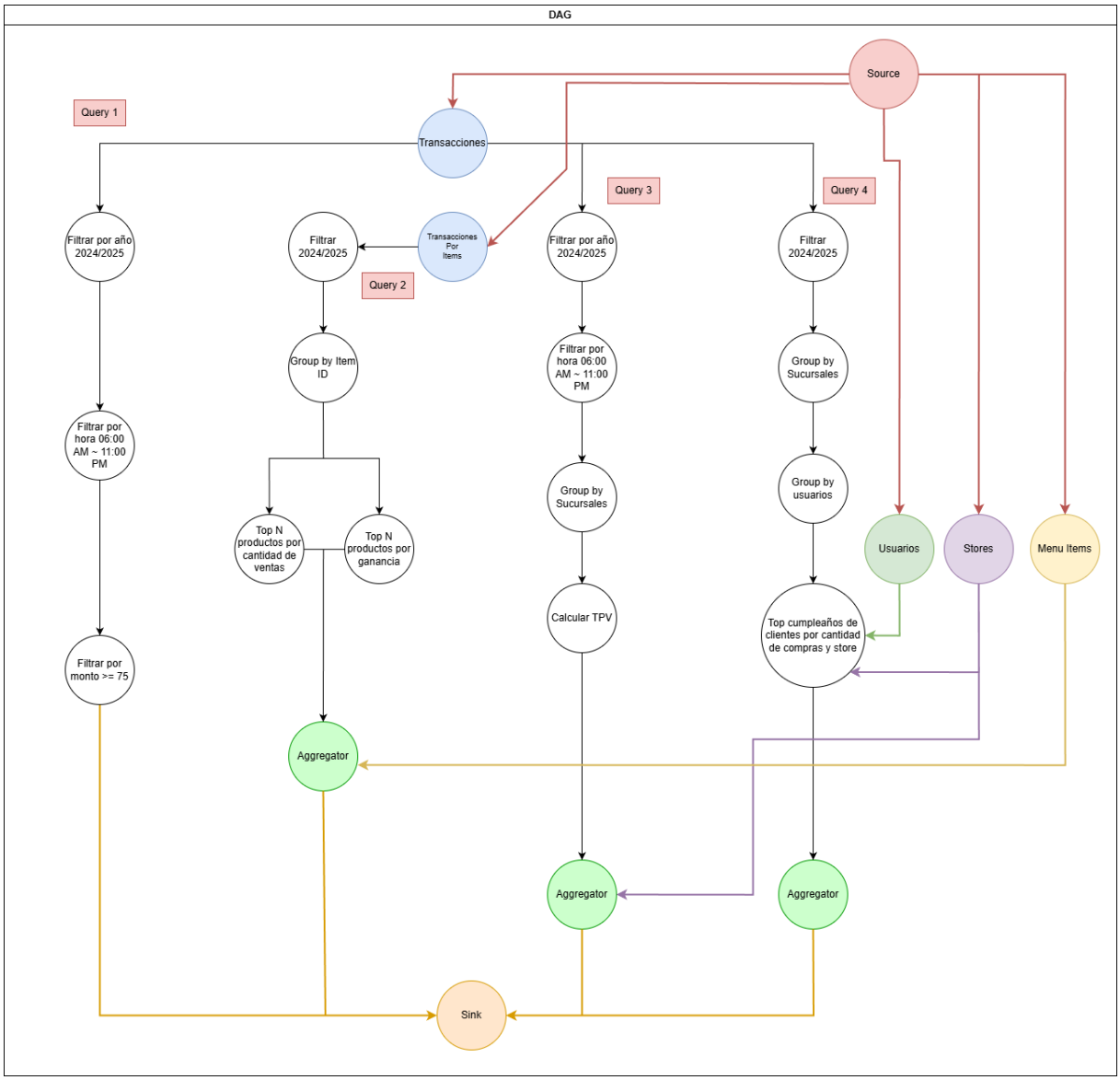

Figura 1: DAG

En el presente diagrama de grafo acíclico es posible ver el flujo general de la resolución de las queries a modo abstracto, con referencia a las fuentes de datos para cada una de ellas:

- Query 1: Transacciones realizadas durante 2024 y 2025 entre las 06:00 AM y las 11:00 PM con monto total mayor o igual a 75.
  - Se filtran las transacciones por año y hora.
  - Se filtran las transacciones por monto.
- Query 2: Productos más vendidos y productos que más ganancias han generado para cada mes en 2024 y 2025.
  - Se filtran las transacciones de ítems por año.
  - Se calculan los top productos agrupados por meses.

- Query 3: TPV (Total Payment Value) por cada semestre en 2024 y 2025, para cada sucursal, para transacciones realizadas entre las 06:00 AM y las 11:00 PM.
  - Se filtran las transacciones por año y hora.
  - Se calcula TPV por semestre/sucursales
- Query 4: Fecha de cumpleaños de los 3 clientes que han hecho más compras durante 2024 y 2025, para cada sucursal.
  - Se filtran las transacciones por año.
  - Se calculan los top clientes por sucursal.
  - Se agregan los nombres de tiendas y los cumpleaños.

1.2. Diagrama de Secuencia

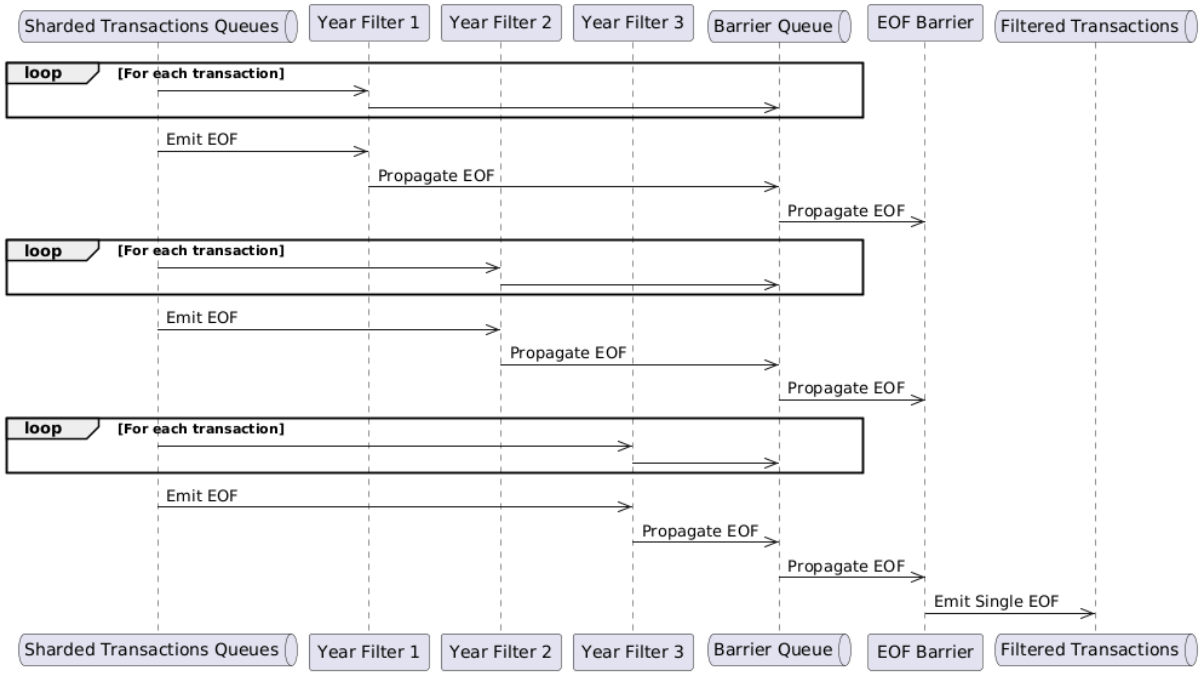

Figura 2: Diagrama de secuencia de manejo de EOF

En el presente diagrama es posible observar el flujo del manejo de EOF para los filtros:

- 1. Se envían todos los registros a los N filtros shardeados por store id.
- 2. Se envía el EOF para cada uno de los filtros.
- 3. Cada filtro envía el EOF al EOF Barrier.
- 4. Una vez que se detectan N EOFs, el barrier emite un único EOF para la siguiente cola.

Al tener los filtros shardeados, cada worker consume de su propia cola, y se garantiza que los N EOFs llegan luego de todas las transacciones a la cola del Barrier.

1.3. Diagrama de Actividades

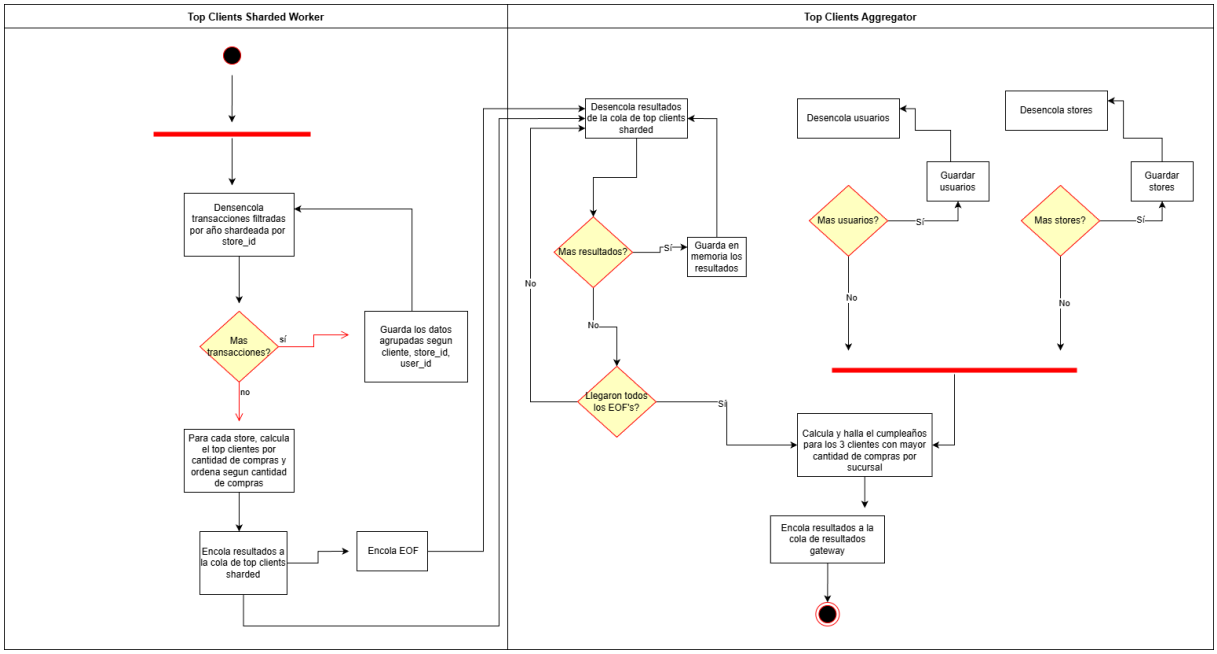

Figura 3: Diagrama de actividades 1

Este diagrama corresponde al hallazgo de las fechas de cumpleaños de los top tres clientes que han hecho más compras durante 2024 y 2025 (última query), para cada sucursal. Se puede ver que el worker top client sharded local, tras desencolar transacciones shardeadas según el parametro store_id y filtradas por año, va bajando a disco los datos que le van llegando. Cuando le llega el EOF calcula el top de clientes agrupados por store y los ordena según la cantidad de compras para el extraer el top 3 local, luego encola los resultados a la cola de top clientes y al final de todo encola el EOF.

Luego el top clients aggregator reciba el output de los top clients sharded local para calcular la fecha de cumpleaños para los 3 clientes con mayor cantidad de compras por sucursal y envíar los resultados finales cuando le llegan los EOF's de todos los sharded workers de top clients.

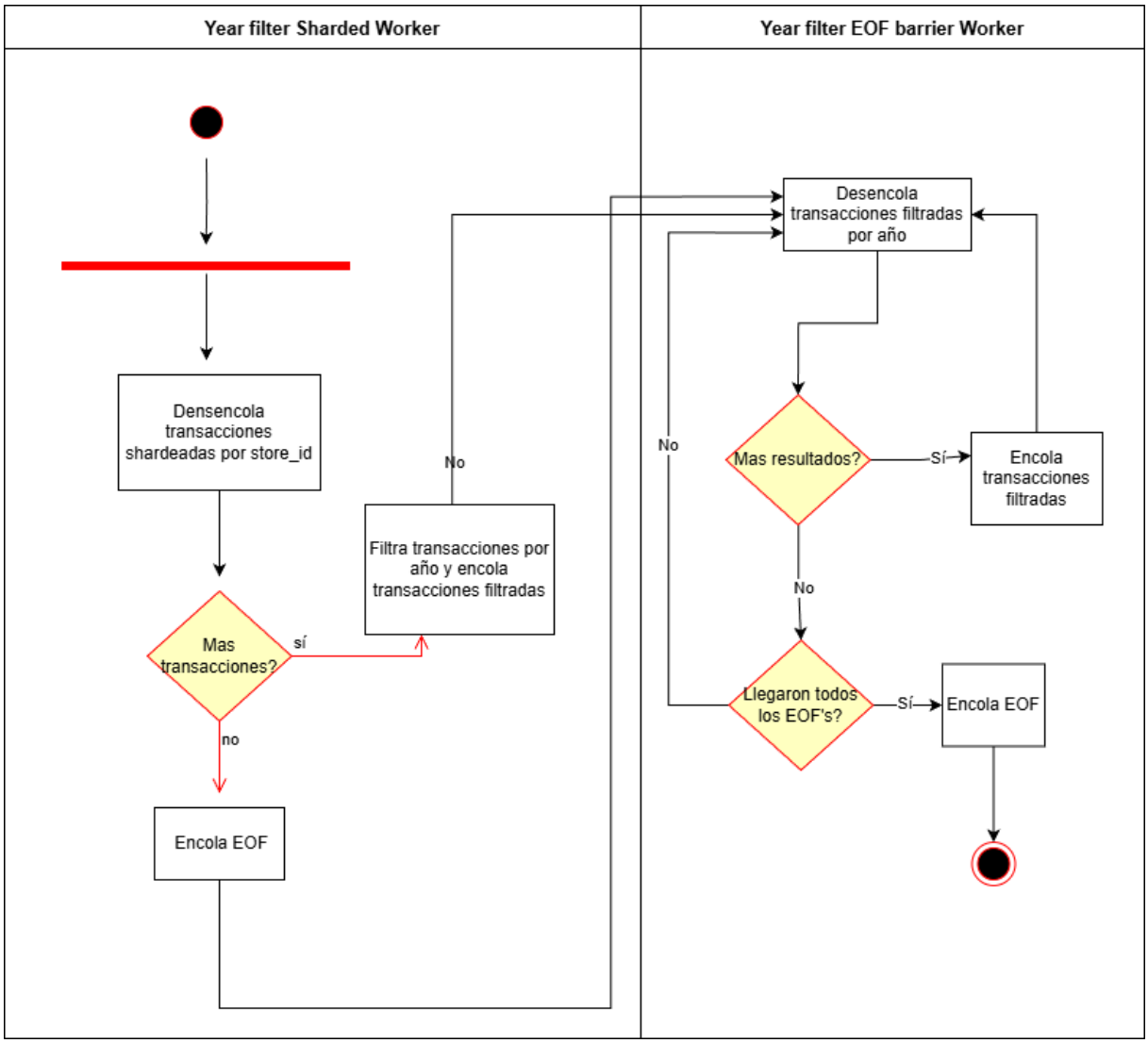

Figura 4: Diagrama de actividades 2

En este otro diagrama se ilustra el flujo YearFilterWorker ->EOFBarrier, en donde el year filter worker a medida que le van llegandos las transacciones, envía bache de transacciones filtradas al EOF barrier. EL EOF barrier worker propaga los baches que le van llegando y espera a los EOF's de todos los YearFilterWorkers antes de propagar su propio EOF al siguiente worker de la pipeline.

1.4. Diagrama de Robustez

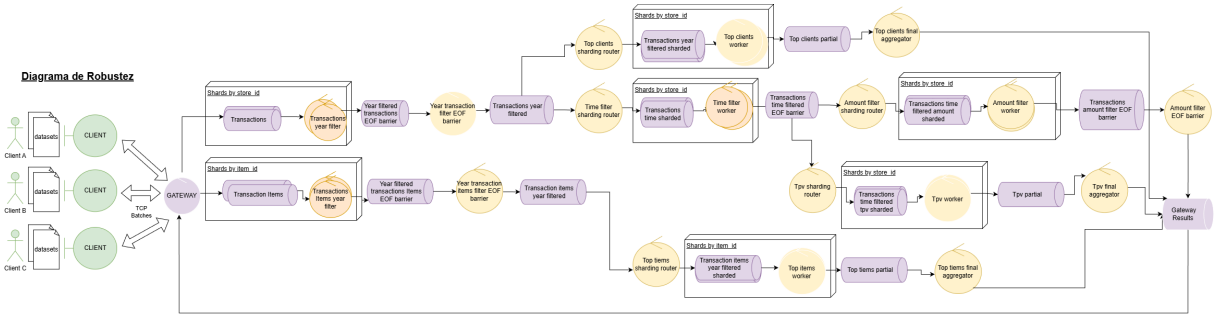

Figura 5: Diagrama de Robustez

En este diagrama se explica el sistema interno, mostrando las colas, los controladores y como interactúan entre ellos.

1.4.1. Flujo General del Sistema

El sistema inicia cuando múltiples clientes envían sus datasets de transacciones al **Gateway** mediante conexiones TCP en formato de lotes (batches). El Gateway actúa como punto central de entrada y salida, coordinando todo el flujo de procesamiento y garantizando la comunicación bidireccional con los clientes.

1.4.2. Procesamiento Paralelo de Datos

Una vez recibidos los datos, el sistema se divide en dos ramas principales de procesamiento que operan de forma independiente y paralela:

1.4.3. Rama de Transacciones

La primera rama procesa las transacciones completas. Inicialmente, los datos se distribuyen mediante **sharding por store_id** para permitir el procesamiento paralelo. Luego, se aplica un **filtro por año** que selecciona únicamente las transacciones correspondientes a los años 2024 y 2025. Este filtrado utiliza barreras de sincronización (EOF barriers) para asegurar que todos los datos de un cliente sean procesados antes de continuar.

Posteriormente, las transacciones filtradas por año se dividen en dos sub-ramas:

1. **Cálculo de Top Clientes**: Las transacciones se vuelven a distribuir por store_id y se procesan en paralelo para identificar los clientes con mayor cantidad de compras por sucursal. Los resultados parciales de cada worker se agregan en un componente final que consolida la información.

2. **Filtrado Temporal y Cálculos Adicionales**: Las transacciones pasan por un **filtro de tiempo** que selecciona aquellas realizadas entre las 06:00 y las 23:00 horas. Este flujo se divide nuevamente en dos caminos:

   - **Filtro por Monto**: Las transacciones se distribuyen nuevamente y se filtran según criterios de monto, generando un conjunto final de transacciones que cumplen todas las condiciones.
   - **Cálculo de TPV (Total Payment Volume)**: Las transacciones se procesan en paralelo para calcular el volumen total de pagos por semestre y sucursal. Los resultados parciales se consolidan en un agregador final.

1.4.4. Rama de Items de Transacciones

La segunda rama procesa los items individuales de cada transacción. Similar a la rama anterior, los datos se distribuyen mediante **sharding por item_id** y luego se filtran por año. Las transacciones filtradas se utilizan para calcular los **Top Items**, identificando los productos más vendidos y rentables por mes. Los resultados parciales de múltiples workers se agregan para obtener el resultado final.

1.4.5. Sincronización y Agregación

El sistema utiliza barreras de sincronización (EOF barriers) en puntos estratégicos para garantizar que todos los datos de un cliente sean procesados completamente antes de avanzar a la siguiente etapa. Esto previene condiciones de carrera y asegura la consistencia de los resultados.

Los resultados de todas las ramas de procesamiento convergen en el componente **Gateway Results**, que consolida toda la información procesada y la envía de vuelta al Gateway para su entrega final a los clientes correspondientes.

1.4.6. Características de la Arquitectura

La arquitectura presentada en el diagrama refleja un diseño robusto que permite:

- **Procesamiento paralelo**: Múltiples workers procesan diferentes particiones de datos simultáneamente
- **Escalabilidad horizontal**: El sistema puede aumentar su capacidad agregando más workers para cada etapa
- **Eficiencia**: La distribución inteligente de datos minimiza la sobrecarga y maximiza el aprovechamiento de recursos

1.5. Diagrama de Paquetes

El diagrama de paquetes se encuentra dividido en distintos elementos lógicos en función de las responsabilidades de cada nodo. Cada módulo tiene su propio protocolo y usan la interfaz de rabbitmq.

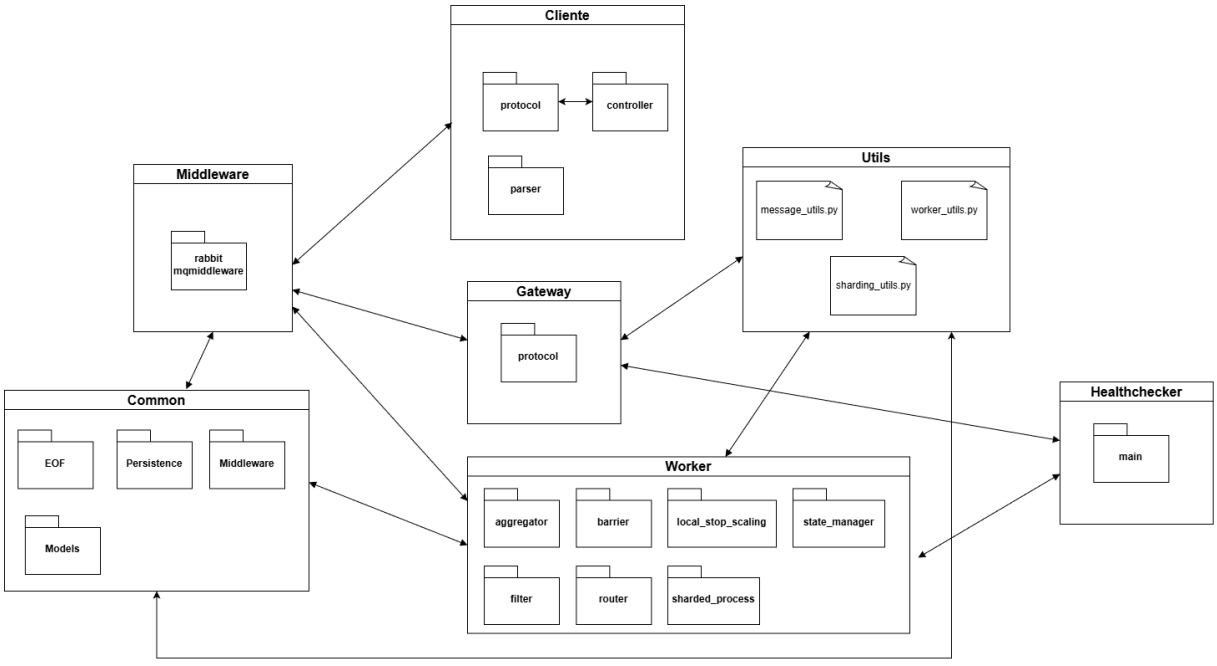

Figura 6: Diagrama de paquetes

1.6. Diagrama de Despliegue

En el diagrama se muestra la distribución de procesos en distintas computadoras. Todos los nodos se comunican a través del middleware RabbitMQ, excepto en el caso de la interacción entre el cliente y el gateway.

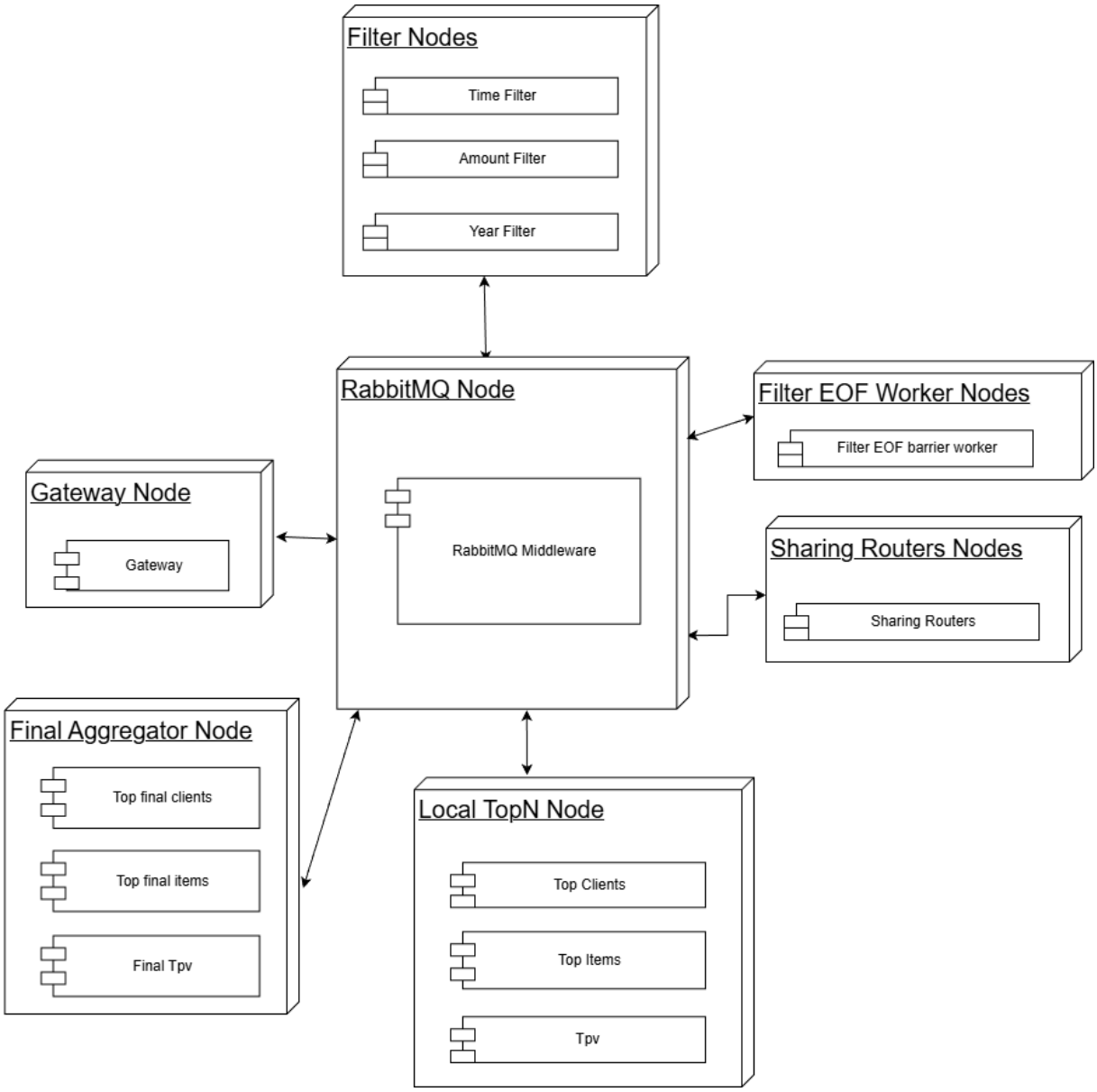

Figura 7: Diagrama de despliegue

2. Desarrollo

2.1. Persistencia y Tolerancia a Fallos

Para lograr tolerancia a fallos se implementaron 2 formas atómicas de persistencia en disco sumado a detección de duplicados: una en base a 3 archivos y *rename*, y otra en base a *append*.

2.1.1. Persistencia en base a Rename

Para simular escritura atómica de archivo en disco, se utilizó un algoritmo que utiliza 3 archivos y *os.rename* (que es atómico según la documentación oficial de Python) en el que para todo caso de corte posible se garantiza la consistencia del estado persistido:

```
Inicialización:
    - Establecer conexión con la cola de mensajes.
    - Intentar cargar el estado desde state.bin:
        * Si state.bin no existe:
```

```
            - Si existe state.backup.bin, restaurar desde allí.
            - En caso contrario, inicializar estado por defecto.
        * Si state.bin está corrupto, restaurar desde state.backup.bin.

Loop de procesamiento:
    - Leer mensaje desde la cola.

    - Si el mensaje posee el mismo ID que el último procesado:
        continue (evitar reprocesamiento).

    Caso: mensaje a procesar
        - Actualizar diccionario de estado y último ID procesado.
        - Escribir state.temp.bin con:
            id: UUID del último mensaje procesado
            state: diccionario por cliente
            checksum: verificación de integridad
        - Si existe state.bin:
            renombrar a state.backup.bin
        - Renombrar state.temp.bin → state.bin (operación atómica)
        - Enviar ACK del mensaje.

    Caso: EOF
        - Emitir resultados al siguiente worker
       (puede enviar duplicados y se debe implementar detección de duplicados de la misma forma en el siguiente
        - Actualizar estado: eliminar cliente e ID correspondiente.
        - Escribir state.temp.bin con:
            id, state, checksum
        - Si existe state.bin:
            renombrar a state.backup.bin
        - Renombrar state.temp.bin → state.bin (atómico)
        - Enviar ACK del mensaje.
```

En la implementación real se reemplazó *state* de los nombres al id del cliente, manteniendo así un archivo por cada cliente.

Este approach tiene dos problemas: por un lado se debe garantizar el orden de los mensajes (que lo resolvemos por routing y colas individuales), y por otro, exige reescribir los archivos por completo. Para mitigar la caída de performance debido a esto último, implementamos también un método de persistencia alternativo en base a *append* (atómico dado que la implementación interna de Python utiliza la bandera de sistema operativo $O_APPENDenUnixqueesatmicosegnLinuxmanualpage$), $paracasosdondenosesnicamentedeinterslosi$

2.2. Multi-Client

Para poder manejar múltiples clientes a la vez, se implementó el ClientSessionManager que mantiene un diccionario thread-safe de las sesiones activas de los clientes; donde cada sesión contiene el client ID (UUID), la referencia al socket y el estado actual de las queries según EOFs recibidos. Éstas sesiones son creadas por el ClientHandler al aceptar una conexión.

Por otro lado, cada mensaje enviado a RabbitMQ incluye metadata con el client ID, para garantizar que todos los workers puedan tener y pasarse referencias del cliente sobre el cual está trabajando.

2.3. Cómo manejamos los EOF's

2.3.1. Cliente → Gateway

El cliente, una vez que termina de enviar todos los archivos CSV correspondientes a un dataset específico (`users`, `stores`, `menu_items`, `transactions`, `transaction_items`), envía un mensaje **EOF** al gateway indicando el tipo de datos que ha completado. Este EOF incluye un código de respuesta que el gateway debe confirmar antes de proceder.

2.3.2. Gateway → Workers

El gateway, al recibir un EOF del cliente, lo procesa y lo propaga selectivamente a los workers que requieren esa información específica.

- Cuando recibe un EOF de `transactions`, lo propaga a la cadena de procesamiento de transacciones.
- Cuando recibe un EOF de `stores`, lo envía a todos los workers que necesitan información de tiendas.

Una vez que el gateway termina de transmitir todos los datos correspondientes a un dataset, envía un **EOF final** a los workers que estaban procesando esa información.

2.3.3. Workers → Gateway

Los workers, cuando terminan de procesar todos los datos recibidos y detectan el EOF, realizan las siguientes acciones:

1. Completan el procesamiento de cualquier lote pendiente.
2. Generan y envían resúmenes o resultados finales si corresponde.
3. Envían un EOF de confirmación al gateway indicando que han terminado su procesamiento.
4. Detienen su consumo de mensajes de la cola de entrada.
5. Liberan los recursos asociados.

2.3.4. Gateway → Cliente

Finalmente, el gateway recopila todos los EOF de los workers y, una vez que confirma que todo el procesamiento ha terminado, envía un **EOF final** al cliente junto con los resultados procesados, cerrando así el ciclo completo de procesamiento.

Problema detectado. Si existen 2 (o más) réplicas de un worker consumiendo de la misma cola, cada una recibirá el mismo EOF y enviará 2 (o más) EOF's hacia adelante. Esto puede causar problemas de sincronización.

Solución implementada. Los EOF ahora contienen un contador en su metadata. Cada worker agrega su `worker_id` al contador y lo reinserta en la cola de entrada. Sólo cuando todas las réplicas han agregado su ID (la última sabe cuántas réplicas hay configuradas), se envía el EOF a la cola de salida, evitando la propagación múltiple.

Alternativas consideradas.

- Definir un **worker líder** dentro de cada grupo de réplicas.
- Asignar a cada réplica su propia cola de salida, y que el gateway consolide los EOF.

2.4. Test EOF Algorithm

Nuestra implementación del test de nuestro algoritmo de EOF es de caja negra dado que utilizamos RabbitMQ y nuestra implementación verdadera de MiddlewareConfig y EOF Handler. Pero aprovechando que tenemos esas implementaciones abstraidas del resto del worker, mockeamos el worker con variables de entorno virtuales y un moqueamos el pasaje de mensaje desde un client-gateway externo. En definitiva estamos probando la implementación específica de MiddlwareConfig y EOF Handler en conjunto, indiferente del resto del código del worker.

2.5. Testing del broker

Los tests del broker se encuentran en el directorio `/tests`. Las instrucciones para correrlos son:

1. Tener RabbitMQ levantado con Docker.
2. Ejecutar `source venv/bin/activate` para levantar el entorno virtual.
3. Instalar dependencias con `pip install -r requirements.txt`.
4. Correr `pytest tests/test_middleware_basic.py`.

2.5.1. Problema Identificado en los Tests

Durante el desarrollo de los tests para la interfaz del broker, observamos que los tests fallaban sin usar `time.sleep`, a pesar de que el sistema funcionaba correctamente dentro de Docker.

Análisis. En desarrollo dockerizado:

- Los workers se ejecutan continuamente y mantienen conexiones persistentes.
- El procesamiento es asíncrono y estable.

En testing local:

- Los consumers se levantan en hilos efímeros.
- Hay un delay entre la inicialización y el consumo efectivo.
- Los mensajes pueden perderse si se publican antes de que el consumer esté listo.

Rol de `time.sleep`. Garantiza que:

- Las colas y bindings estén activos.
- Los consumers estén listos para recibir mensajes.
- No se pierdan mensajes por asincronías.

Por tanto, consideramos su uso justificado en ambiente de testing, donde no existen workers persistentes.

Resumen de mejoras recientes.

- La metadata de los mensajes ahora incluye **client ID** y todos los datos asociados.
- Cada worker almacena la información en memoria bajo un diccionario `client_id → data`.
- Se implementó **automatización de chequeos de resultados**.

2.6. Anexo: Mejoras posibles

- Se podría haber hecho escalable los **aggregators** a nivel clientes, aunque lo dejamos por fuera del scope.
- Actualmente **EOF Barrier** recibe tanto las transacciones como EOFs, pero las transacciones podrían directamente omitir pasar por este worker.