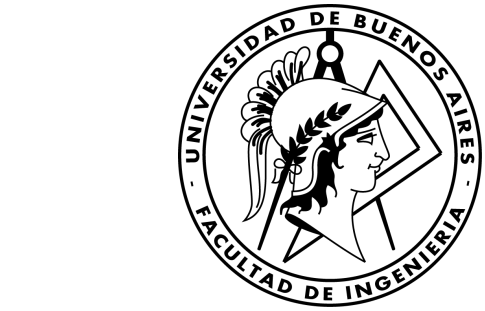

Trabajo Práctico: Tolerancia a Fallos

[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)

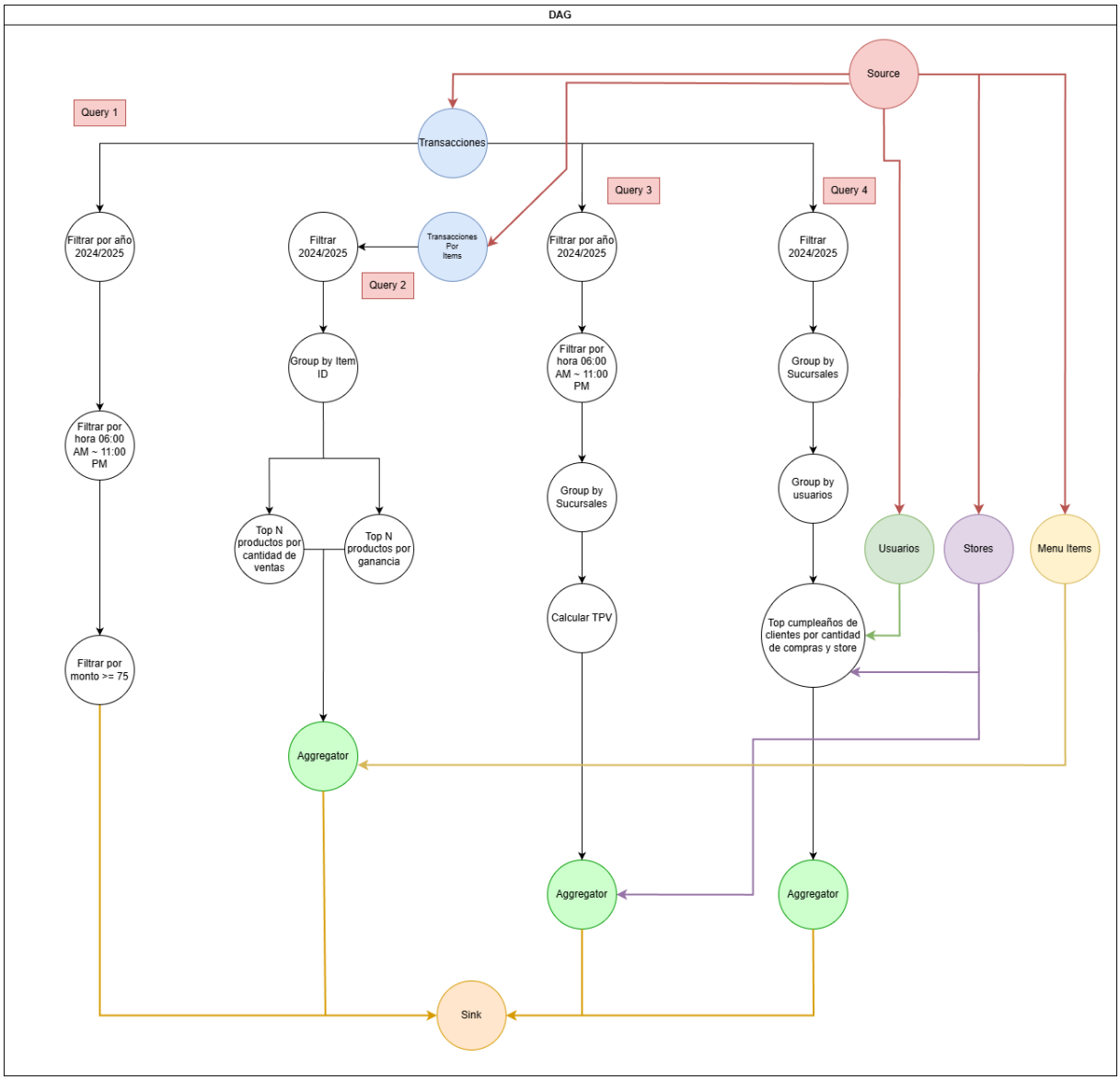

Figura 1: DAG

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

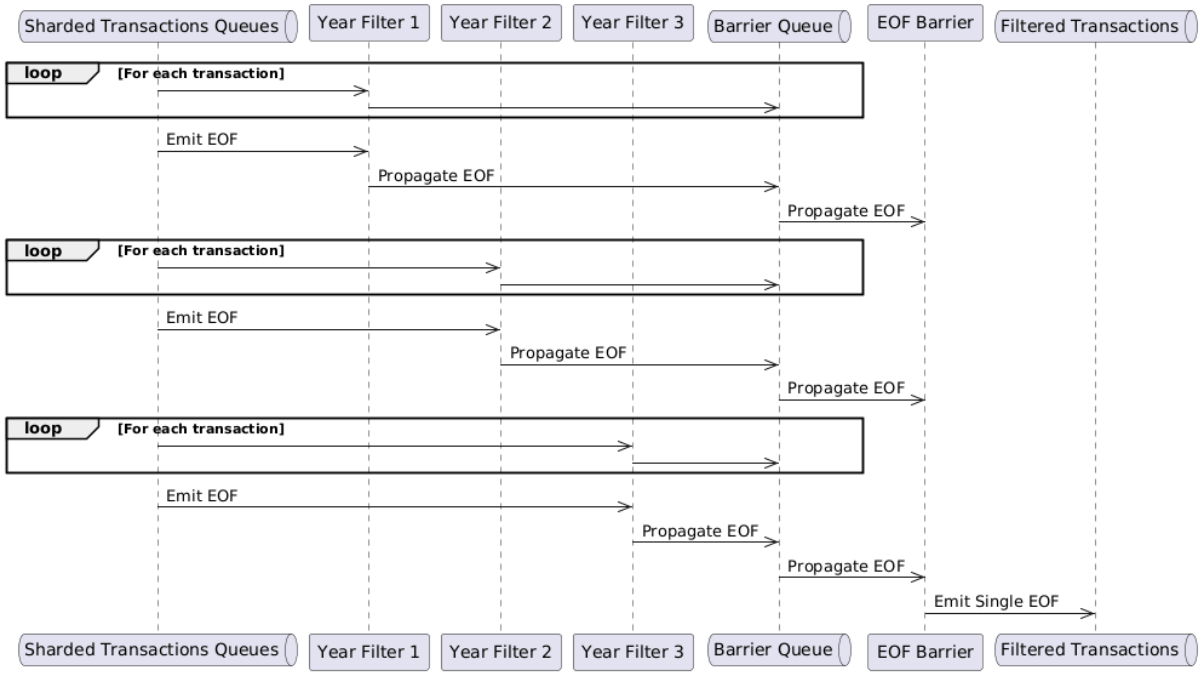

Figura 2: Diagrama de secuencia de manejo de EOF

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

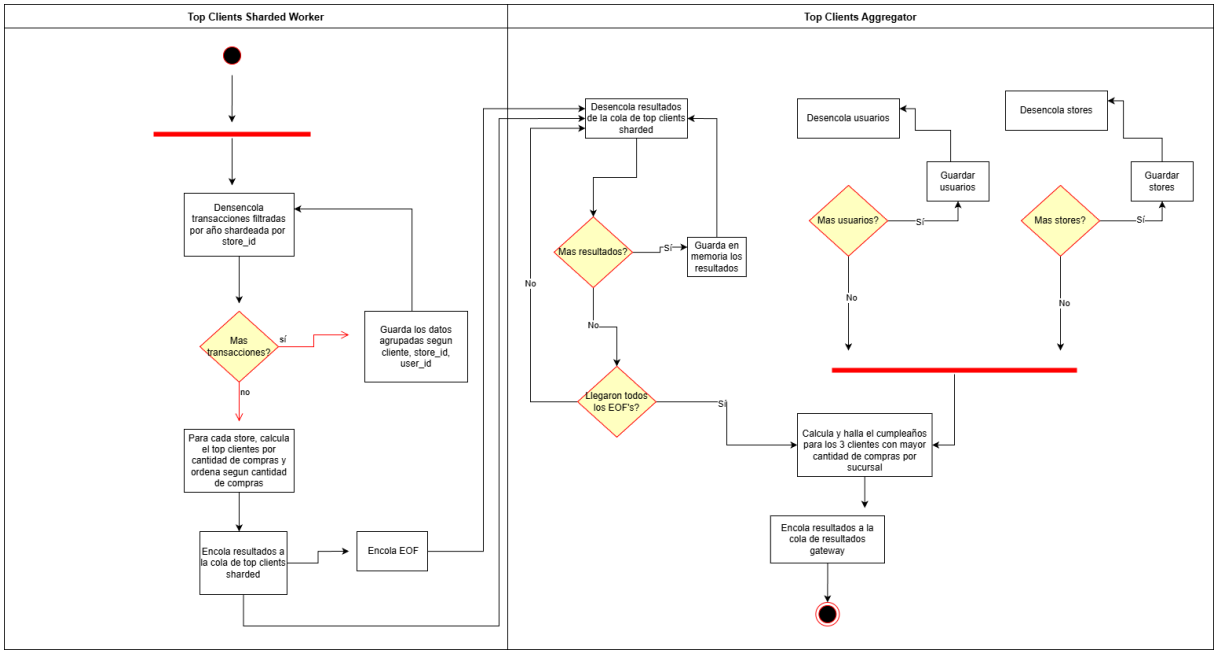

Figura 3: Diagrama de actividades 1

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.

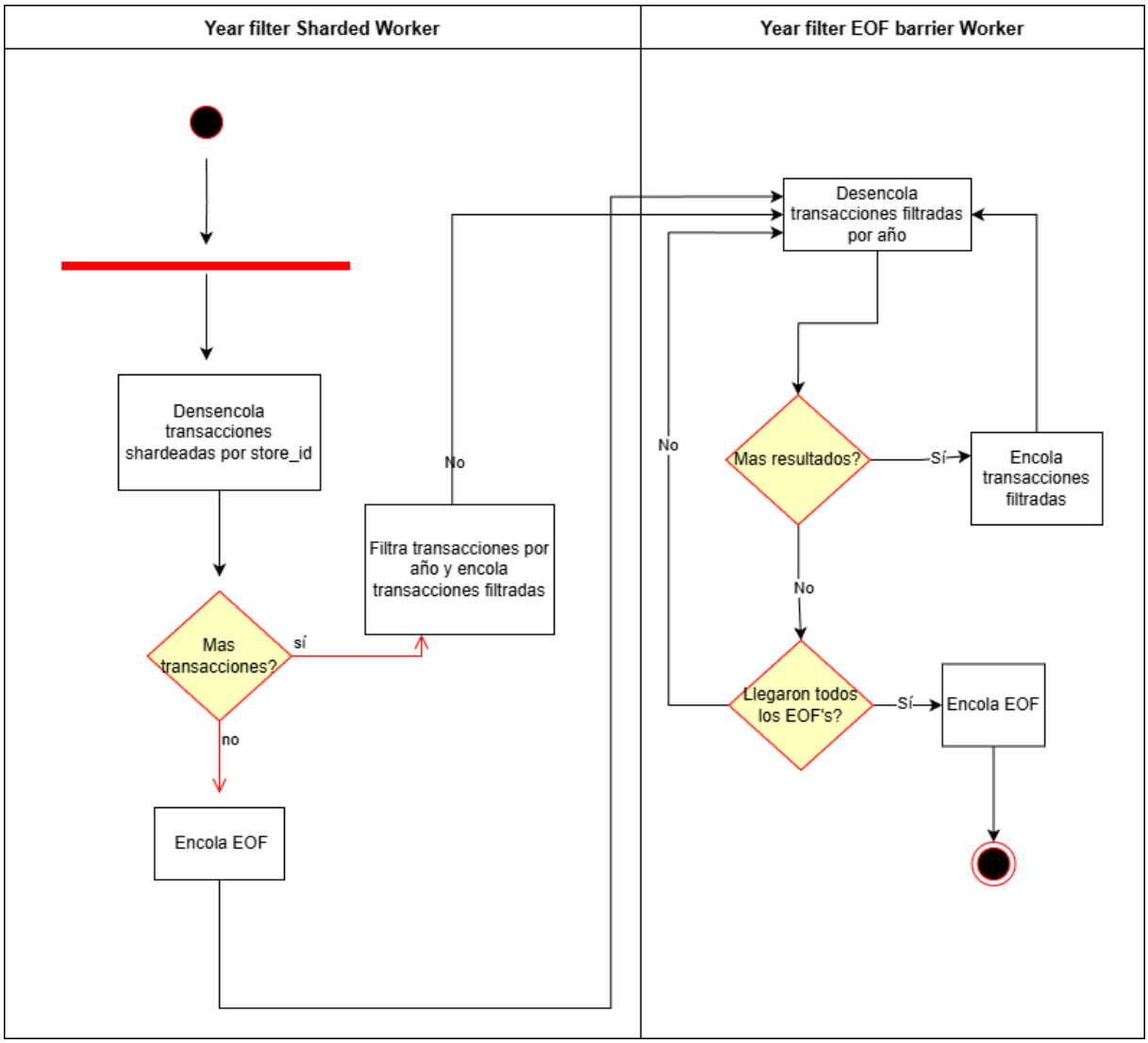

Figura 4: Diagrama de actividades 2

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

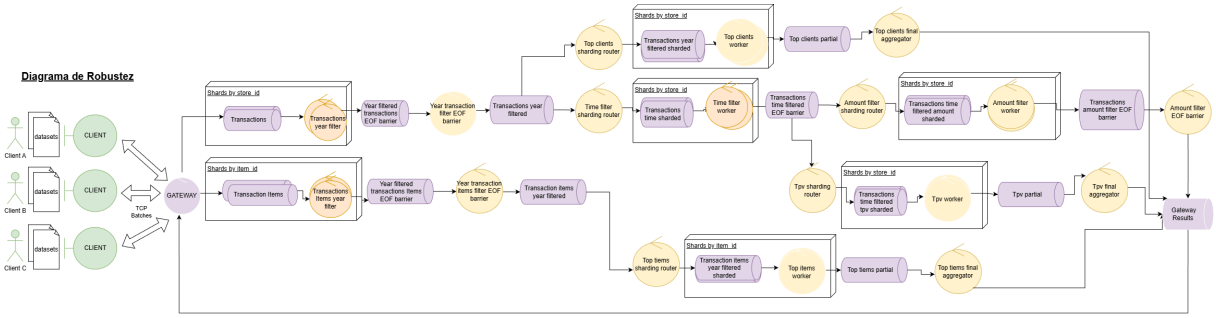

Figura 5: Diagrama de Robustez

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.

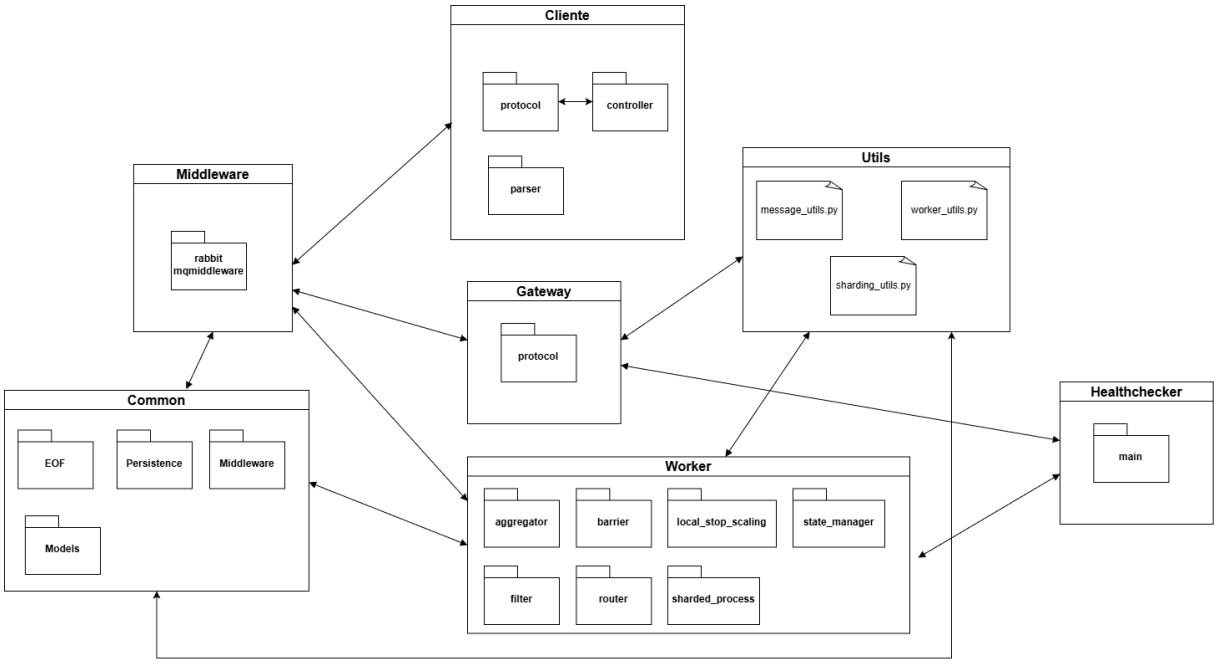

Figura 6: Diagrama de paquetes

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.

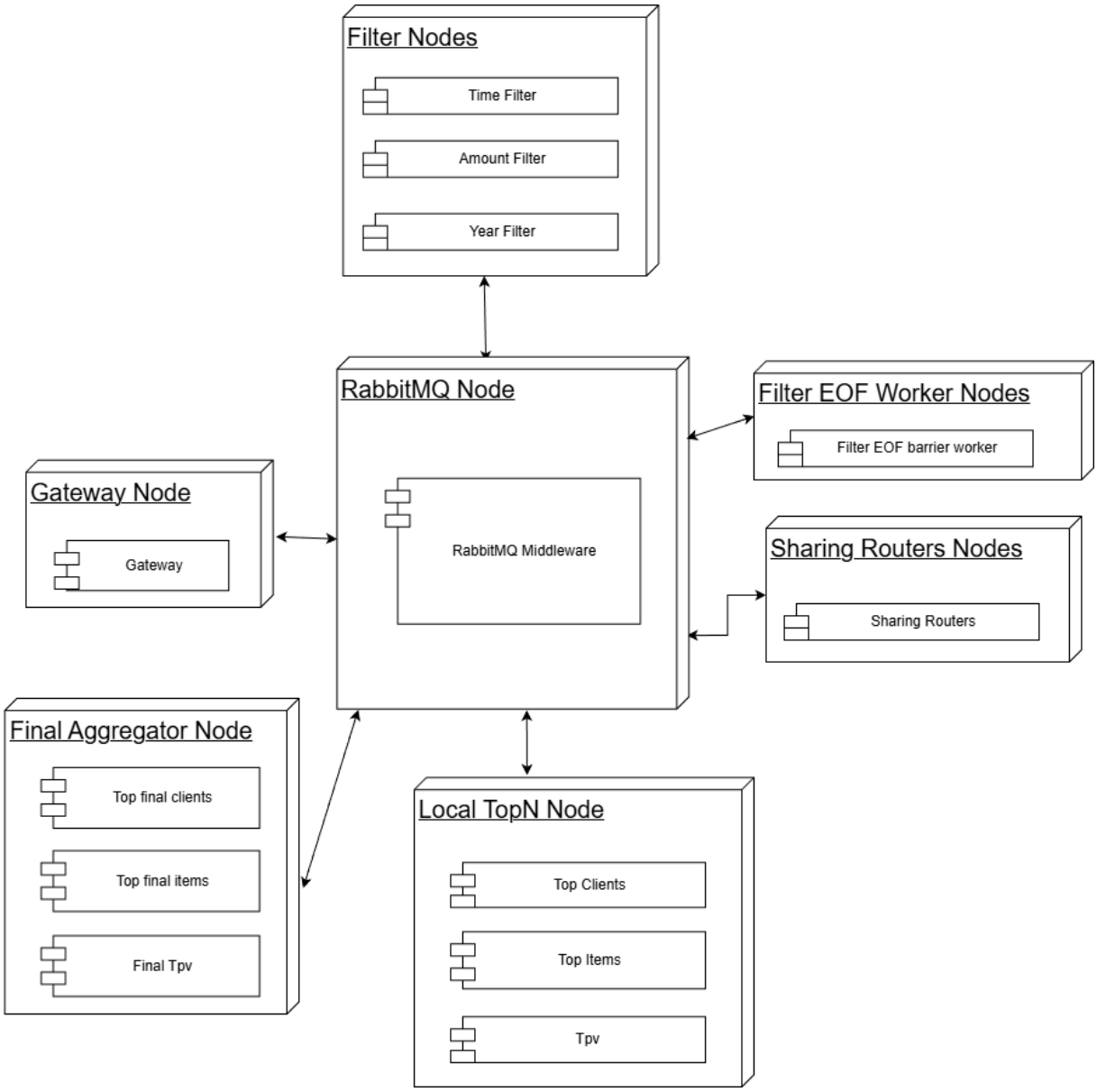

Figura 7: Diagrama de despliegue

2. Desarrollo

2.1. Persistencia y Tolerancia a Fallos

Para lograr tolerancia a fallos se implementaron dos formas atómicas de persistencia en disco, sumadas a detección de duplicados: una basada en tres archivos y *rename*, y otra basada en *append*.

2.1.1. Mecanismos de Tolerancia a Fallos Implementados

El sistema implementa múltiples mecanismos para garantizar la tolerancia a fallos y la recuperación ante interrupciones:

- **Detección de mensajes duplicados**: Cada mensaje incluye un identificador único (`message_uuid`) que se persiste antes de procesar el mensaje. Al reiniciar un worker, se verifica si un mensaje ya fue procesado consultando el almacén de UUIDs procesados, evitando así el reprocesamiento de mensajes en caso de fallos.

- **Persistencia atómica de estado**: El estado de cada worker se persiste de forma atómica utilizando un protocolo de dos fases: primero se escribe en un archivo temporal con `fsync`, luego se valida la integridad mediante checksum, y finalmente se reemplaza el archivo original de forma atómica usando `os.replace`. Esto garantiza que ante un fallo durante la escritura, el estado anterior permanezca intacto.
- **Recuperación de estado al reinicio**: Al iniciar, cada worker carga automáticamente el estado persistido desde disco. Los agregadores restauran tanto el estado de agregación intermedia como los contadores de EOF, permitiendo continuar el procesamiento desde el punto donde se interrumpió.
- **Reconexión automática al middleware**: El middleware implementa lógica de reconexión automática con backoff exponencial ante fallos de conexión con RabbitMQ. Si un mensaje no puede ser confirmado (ACK) debido a una desconexión, se fuerza la reconexión y el broker reenvía el mensaje, preservando la semántica *at-least-once*.
- **Requeue de mensajes en caso de error**: Cuando ocurre un error durante el procesamiento de un mensaje, el middleware envía un NACK con `requeue=True`, permitiendo que el mensaje sea reintentado posteriormente. Si el worker es interrumpido (por ejemplo, durante un shutdown), también se solicita el requeue para evitar pérdida de datos.
- **Rollback de estado ante errores**: Si ocurre una excepción durante el procesamiento de un batch, el worker restaura el estado anterior al inicio del procesamiento, garantizando que no queden datos parciales o inconsistentes.

2.1.2. Persistencia en base a *rename*

Para simular escritura atómica de archivo en disco, se utilizó un algoritmo que emplea tres archivos y *os.rename* (que es atómico según la documentación oficial de Python). Este mecanismo garantiza consistencia del estado persistido ante cualquier falla posible:

```
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; el siguiente worker debe implementar
           el mismo mecanismo de detección de duplicados).
        - 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* en los nombres por el ID del cliente, manteniendo un archivo por cada cliente.

Este enfoque tiene dos limitaciones: por un lado, depende críticamente del mantenimiento del orden de los mensajes (resuelto mediante *routing* y colas individuales); por otro, exige reescribir los archivos completos en cada actualización. Para mitigar el posible impacto en la performance, se implementó también un método alternativo de persistencia basado en *append* (atómico dado que la implementación interna de Python utiliza la bandera O_APPEND en Unix, la cual es atómica según la página del manual de Linux), para casos donde únicamente es necesario registrar los IDs de mensajes ya procesados.

2.1.3. Sistema de Healthcheck

El sistema incluye un servicio de monitoreo de salud basado en UDP que detecta y reinicia automáticamente servicios caídos:

- **Arquitectura**: Cada worker y el gateway exponen un servicio UDP que responde a verificaciones de salud. Los procesos *healthchecker* monitorean estos servicios periódicamente enviando mensajes de healthcheck y ejecutando `docker restart` cuando detectan que un servicio ha fallado.
- **Protocolo UDP**: El protocolo es simple y eficiente: el healthchecker envía un byte con valor 1 (HEALTHCHECK_REQUEST) al puerto UDP configurado (por defecto 9290), y el servicio responde con un byte con valor 2 (HEALTHCHECK_ACK). Si no se recibe respuesta dentro del timeout configurado, se incrementa un contador de errores consecutivos.
- **Topología Ring**: El sistema soporta múltiples healthcheckers organizados en un anillo. Cada healthchecker monitorea un subconjunto de servicios asignados mediante hash consistente y también monitorea al siguiente healthchecker en el anillo. Si un healthchecker cae, el anterior en el anillo lo detecta y lo reinicia, eliminando el punto único de fallo.
- **Reinicio automático**: Cuando un healthchecker detecta `max_errors` errores consecutivos (configurable, por defecto 3), ejecuta `docker restart` sobre el contenedor correspondiente. El contador de errores se resetea después del reinicio, y el healthchecker espera un intervalo configurado antes de verificar nuevamente.
- **Integración transparente**: El servicio de healthcheck se integra automáticamente en todos los workers (a través de `BaseWorker`) y en el gateway, sin requerir configuración adicional en el código de cada componente.

2.2. 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.