.UBAfiuba
FACULTAD DE INGENIERÍA

Sistemas Distribuidos
75.74

Trabajo Práctico:
Coffee Shop Analyzer

Nombre	Padrón
Baldi, Tomás	108317
Katta, Gabriel	105935
Bellido, Santiago	106449

Índice

1. Alcance

El trabajo consiste en el diseño de un **sistema distribuido** para analizar información transaccional de una cadena de cafeterías en Malasia, capaz de atender múltiples clientes de forma concurrente y de mantener su operación ante fallas de procesos.

El sistema debe cumplir los siguientes **requerimientos funcionales**:

1. Filtrar transacciones realizadas en 2024 y 2025, entre las 06:00 y las 23:00 horas, con monto mayor o igual a 75.
2. Identificar productos más vendidos y los que más ingresos generaron, mes a mes durante 2024 y 2025.
3. Calcular el *Total Payment Value (TPV)* por semestre y por sucursal, considerando únicamente transacciones dentro del rango horario establecido.
4. Obtener la fecha de cumpleaños de los tres clientes con mayor cantidad de compras en cada sucursal.

Adicionalmente, el sistema deberá:

- Soportar múltiples ejecuciones de las consultas para un mismo cliente sin reiniciar el servidor, así como la ejecución concurrente de varios clientes, garantizando la correcta liberación de recursos al finalizar cada ejecución.
- Demostrar **resiliencia y tolerancia a fallos** frente a caídas de procesos, manteniendo resultados consistentes e incorporando mecanismos de detección, recuperación y pruebas de inyección de fallas.

A continuación se presentarán los Diagramas correspondientes a la propuesta de solución:

2. Contexto

2.1. Diagrama de Secuencia

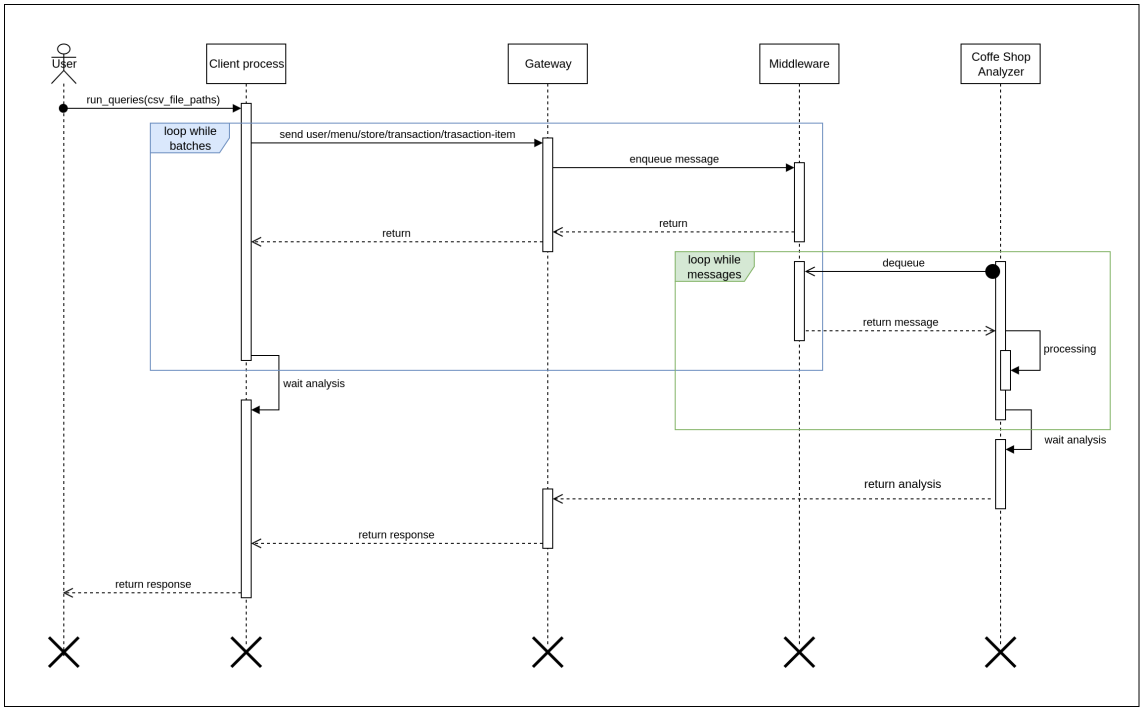

Figura 1: Diagrama de Secuencia del sistema.

En el diagrama se observa que, una vez iniciada la consulta por parte del *User*, el proceso principal del cliente envía la información obtenida de los archivos CSV en *batches* hacia el *Gateway*. Este componente encola los mensajes a través del *Middleware*. El *Coffee Shop Analyzer* desencola la información para ejecutar los *pipelines* y procesar la consulta. Finalizado el análisis, el resultado retorna al *User* a través del *Gateway*.

2.2. DAG

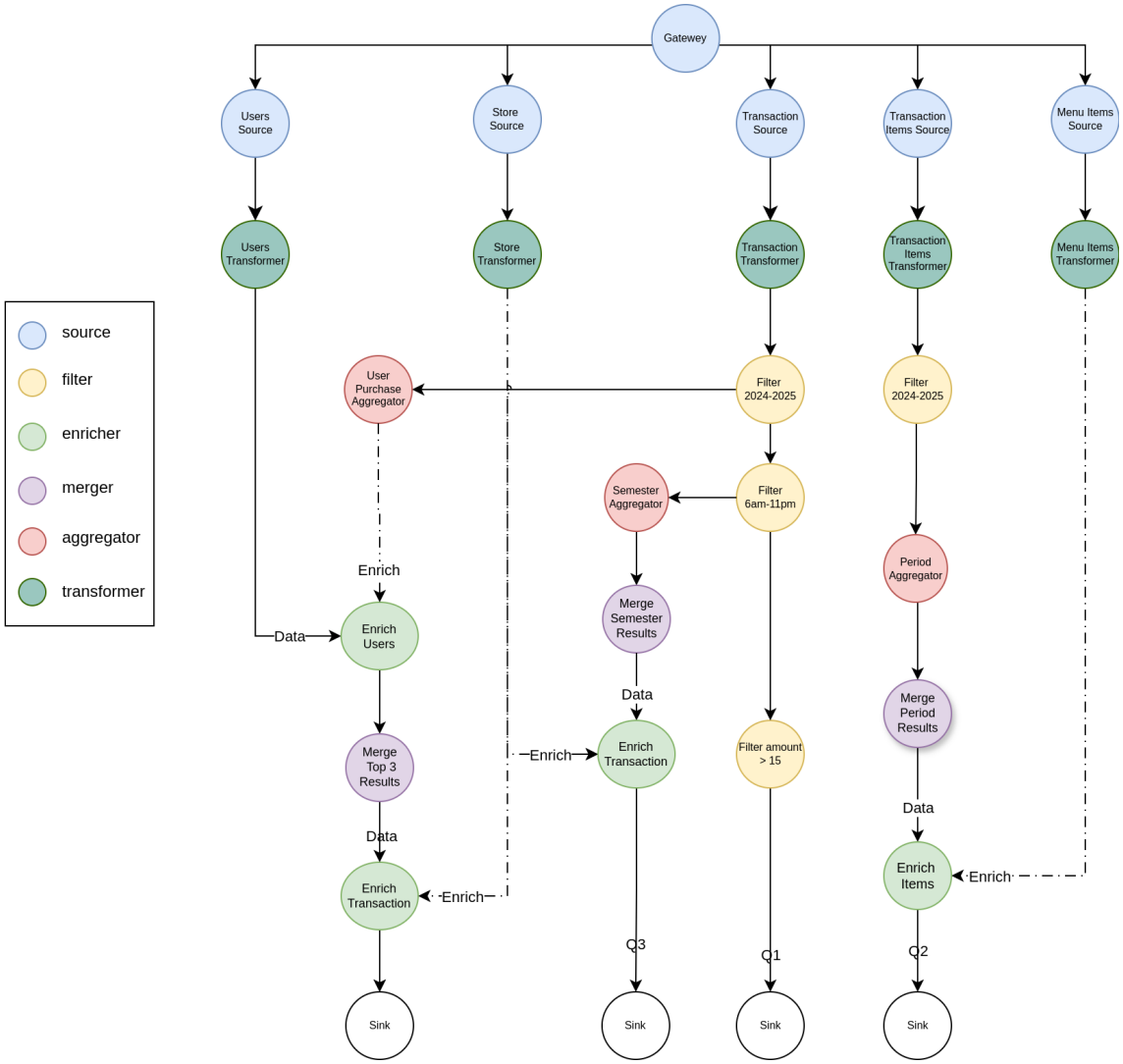

Figura 2: Pipeline de procesamiento (DAG).

Componentes del *pipeline*:

- **Transformers**: encargado de pasar las lineas csv del batch a entidades en un nuevo batch
- **Sources**: colas con la información enviada por el `client_process`, pueden contener batches o registros indivuales.
- **Aggregators**: acumulan en buffer y construyen estructuras que sintetizan información.
- **Filters**: aplican condiciones para filtrar datos.
- **Mergers**: fusionan resultados parciales de nodos anteriores.
- **Enrichers**: combinan datos de dos fuentes para generar una única fuente enriquecida.

3. Contenedores

3.1. Diagrama de Despliegue

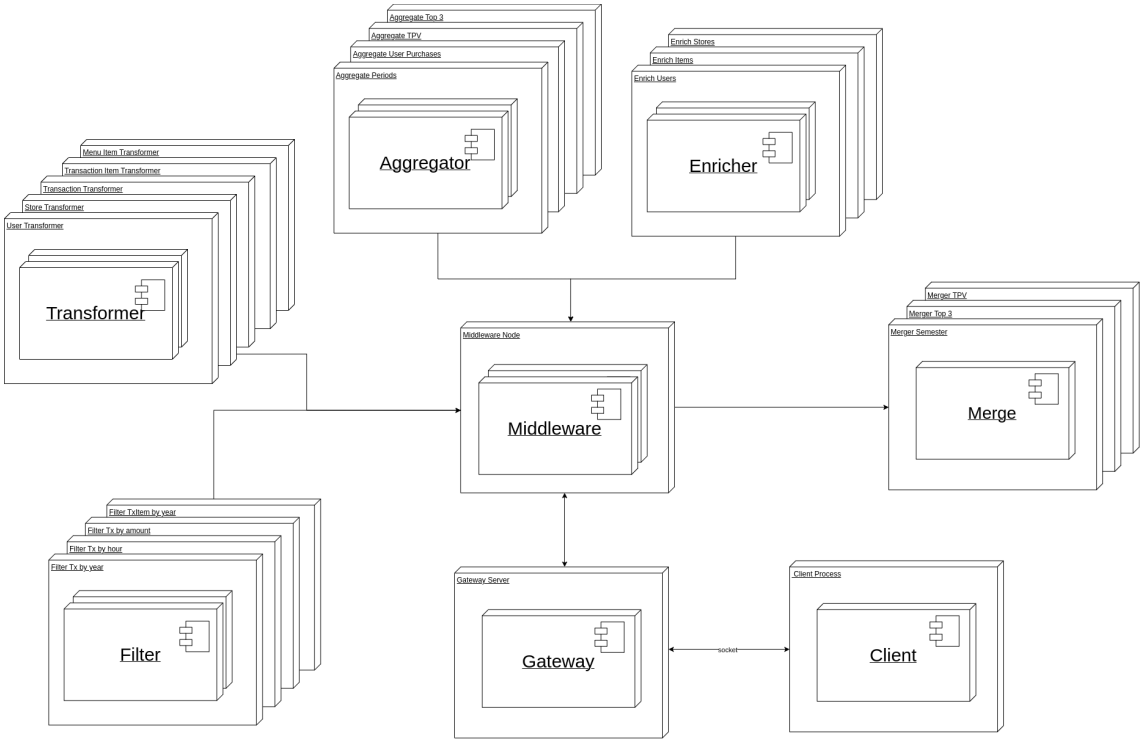

Figura 3: Comunicación de alto nivel entre elementos del sistema.

La mayoría de los componentes se comunican mediante el *Middleware* (RabbitMQ). El *Gateway* y el *Client* manejan su comunicación por sockets TCP.

Para simplificar, se agrupan componentes cuyos *workers* realizan operaciones similares sobre distintos dominios:

- **Transformer**: múltiples *transformer* (User, Transaction, Transaction Item, Store, Menu Item).
- **Filter**: múltiples *workers* (Amount, Hourly, Yearly).
- **Aggregator**: múltiples *workers* (Top 3, TPV, User Purchases, Periods).
- **Enricher**: múltiples *workers* (Stores, Items, Users).
- **Merge**: *worker* único por dominio (TPV, Top 3, Semester) por estado compartido.
- **Middleware**: coordina la comunicación asíncrona (RabbitMQ).
- **Gateway y Client**: interfaz de acceso al sistema.

4. Componentes

4.1. Diagrama de Robustez

El diagrama muestra la distribución de instancias y cómo trabajan en conjunto para responder *queries*. Grupos como filtros, enriquecedores y agregadores usan múltiples *workers* por ser *stateless*,

mientras que los *mergers* requieren instancias únicas para juntar resultados.

Ver Figura 4 en la página siguiente

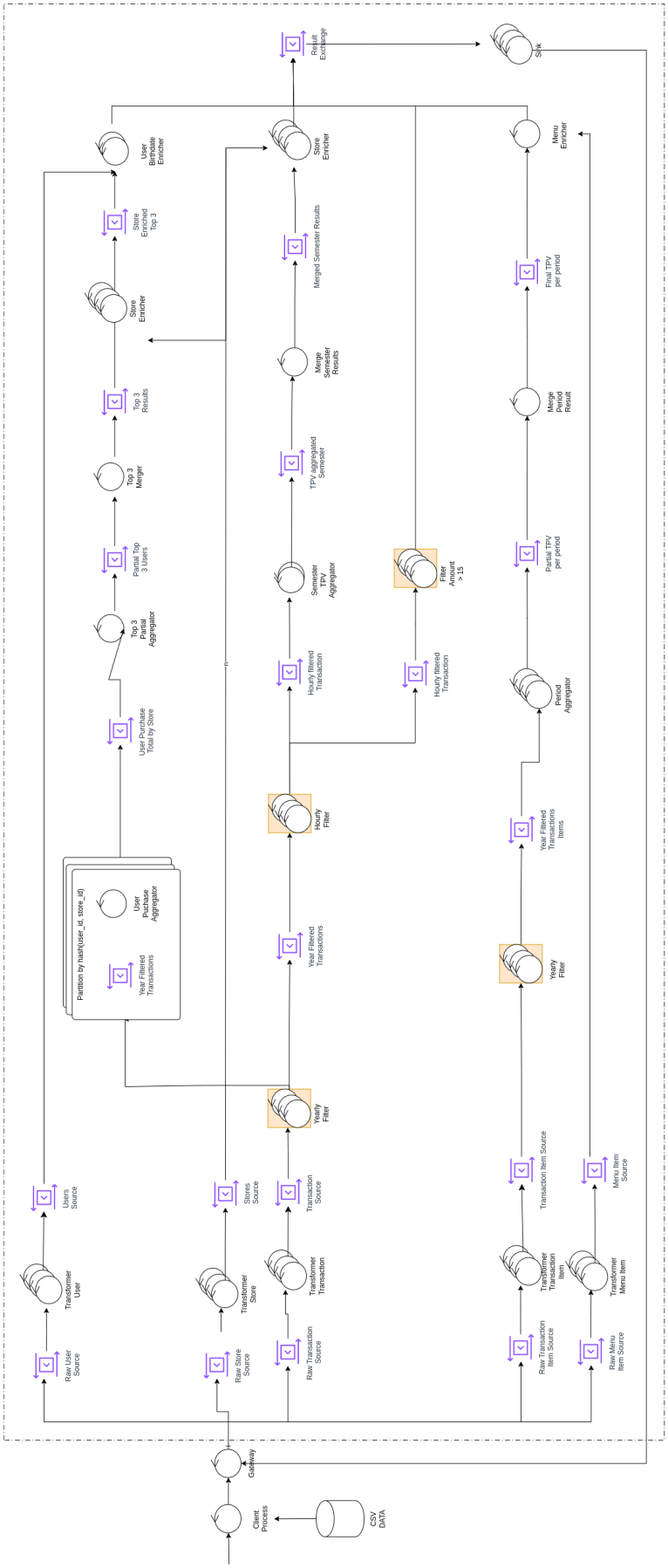

Figura 4: Diagrama de robustez: distribución de instancias del pipeline.

4.2. Diagrama de Actividad

El **User Purchase Aggregator** (múltiples instancias, shardeado por (*user_id*, *store_id*)) recibe transacciones filtradas (años 2024–2025) y construye una estructura interna con la cantidad de compras por usuario y tienda. Al finalizar, sus resultados para esos pares son definitivos.

Luego, el **Top 3 Aggregator** (shardeado por *user_id*) arma el top 3 parcial por tienda. Por último, un *worker* **Merge Top 3 Result** fusiona todos los parciales para obtener el top 3 final por tienda.

Ver Figura 5 en la página siguiente

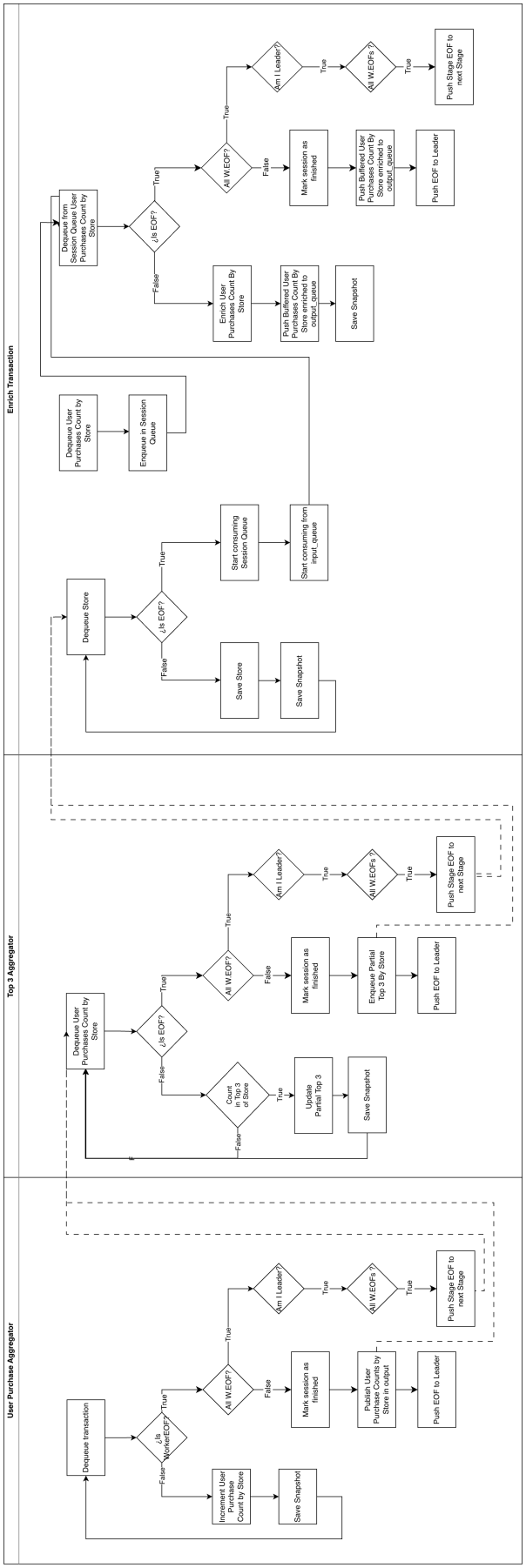

Figura 5: Diagrama de actividad

5. Código

5.1. Diagrama de Paquetes

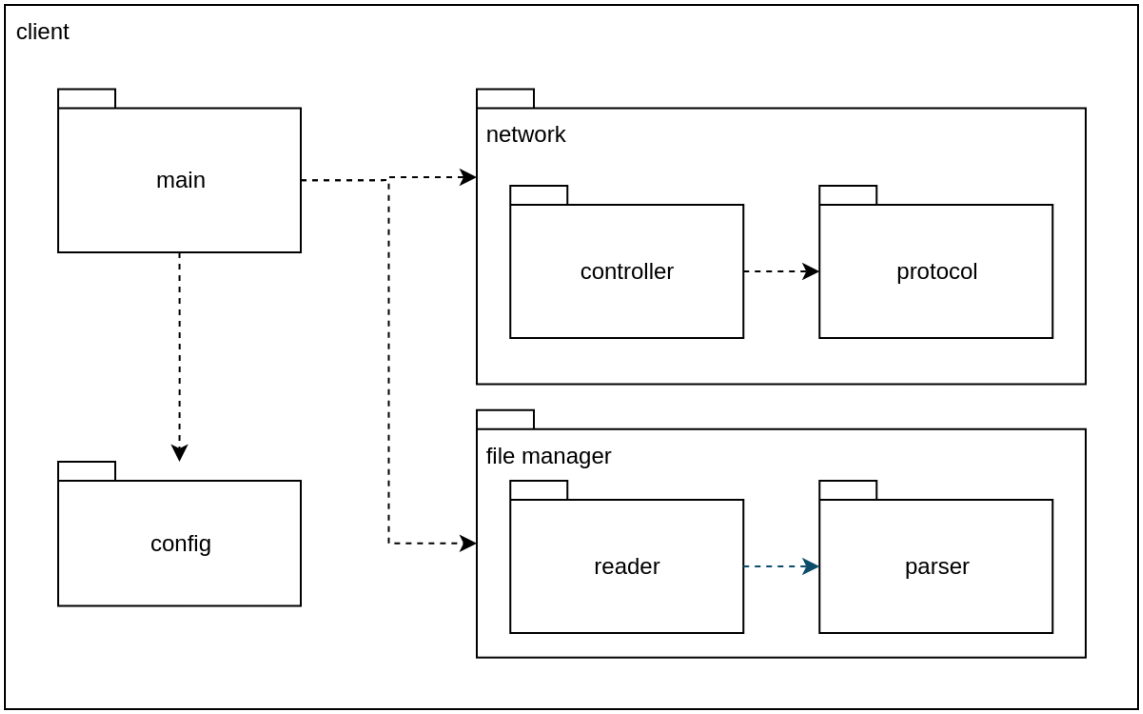

Figura 6: Diagrama de paquetes: Vista Cliente

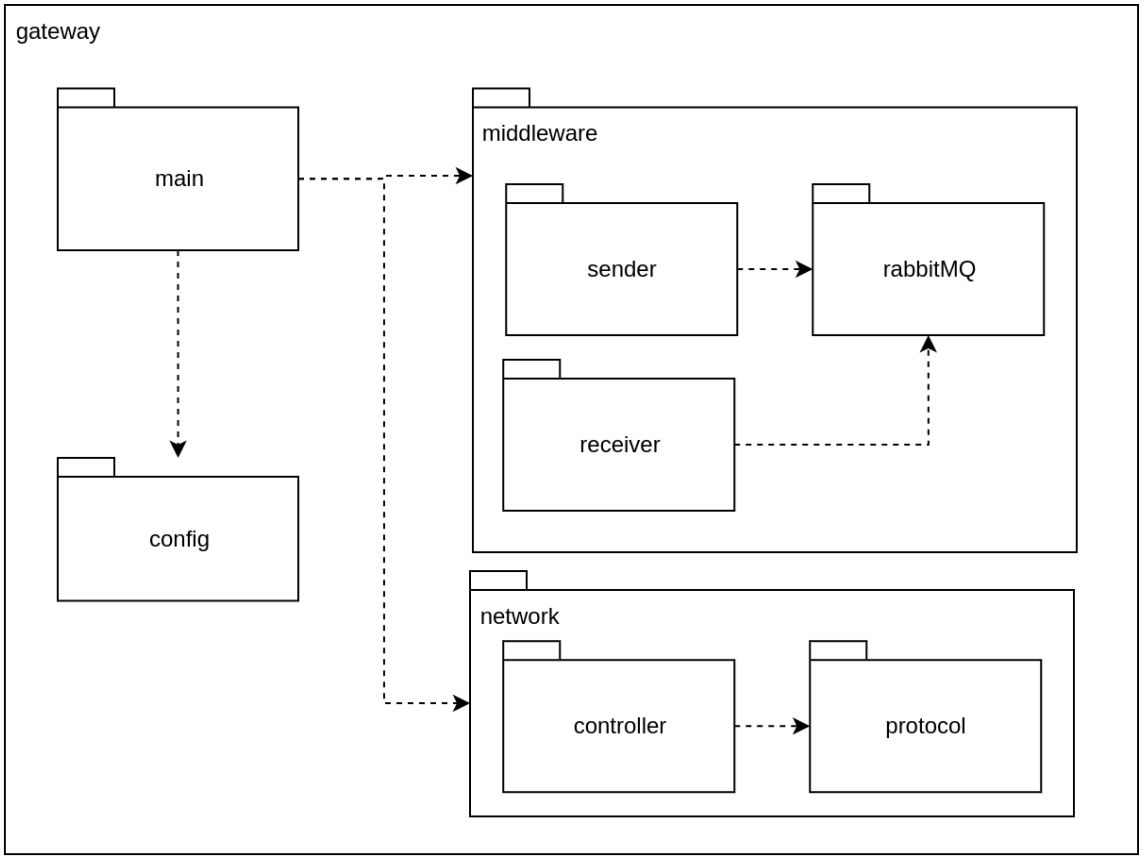

Figura 7: Diagrama de paquetes: Vista Gateway

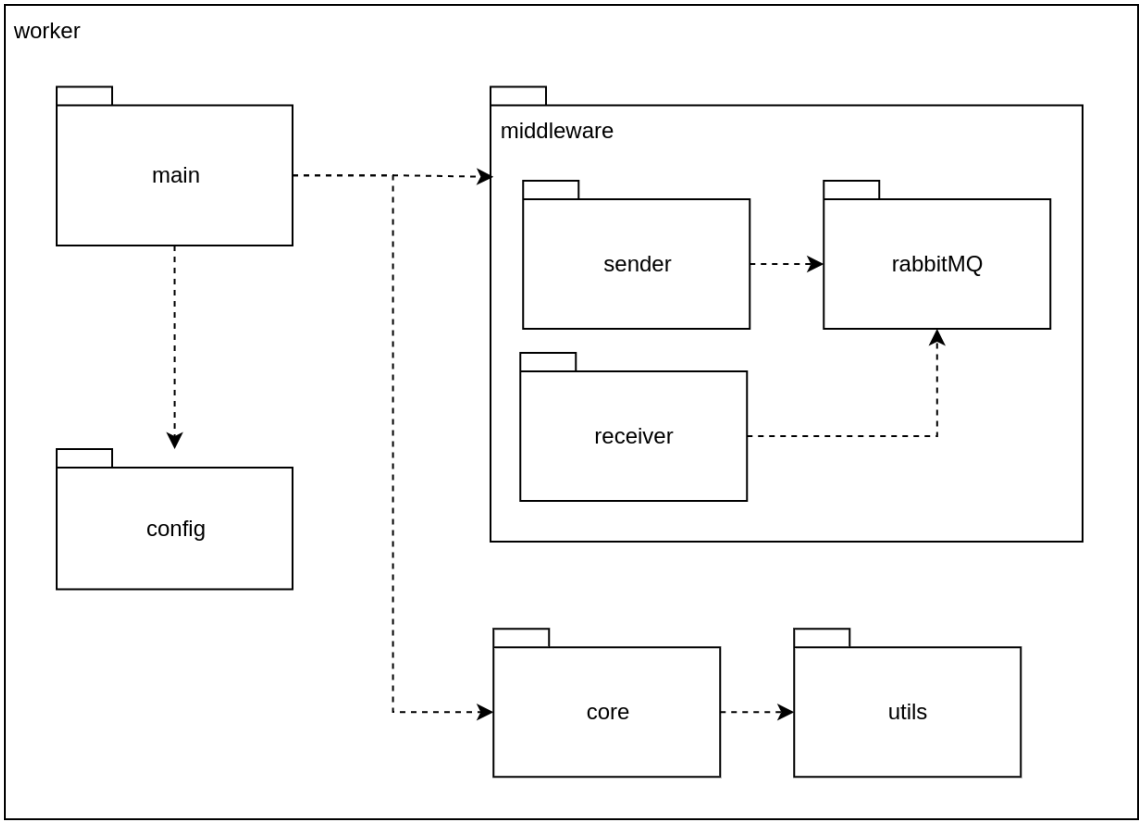

Figura 8: Diagrama de paquetes: Vista Worker

El diagrama muestra agrupación y dependencias entre módulos de cada componente.

Client

- **Main**: usa `Config`, `Network` y `FileManager`.
- **Config**: provee configuración para el funcionamiento de componentes.
- **Network**: gestiona comunicación con *Gateway*; usa `Controller` y `Protocol`.
- **FileManager**: maneja CSV de entrada; submódulos `Reader` y `Parser`.

Gateway

- **Main**: usa `Config`, `Network` y `Middleware`.
- **Config**: configuración de componentes.
- **Network**: comunicación con *Client*; usa `Controller` y `Protocol`.
- **Middleware**: integra RabbitMQ; submódulos `Sender` y `Receiver` para encolar y desencolar mensajes con *Workers*.

Workers

- **Main**: usa `Config`, `Middleware` y `Core`.

- **Config**: configuración de componentes.
- **Middleware**: submódulos `Sender`, `Receiver` y `RabbitMQ`; coordina comunicación con *Gateway* y otros *Workers*.
- **Core**: lógica específica de cada *Worker*; usa `Utils`.
- **Utils**: utilidades para `Core`.

6. Resiliencia

6.1. Precondiciones del sistema

El sistema está diseñado para recuperarse automáticamente de fallos de *workers*, pero requiere ciertas condiciones para funcionar correctamente. A continuación se detallan los requisitos que deben cumplirse.

Infraestructura y persistencia

- **Estado persistido en volúmenes durables**: Los directorios de sesiones deben ubicarse en volúmenes Docker persistentes. Si un contenedor se reinicia y pierde estos directorios, no podrá recuperar su estado previo.
- **HealthChecker con acceso a Docker**: El **HealthChecker** necesita permisos para ejecutar `docker start` y revivir contenedores caídos. Detecta fallos cuando los *workers* dejan de enviar *heartbeats* y los reinicia automáticamente.
- **Filesystem con escrituras atómicas**: El sistema requiere que las escrituras a disco sean efectivamente atómicas (por ejemplo, usando archivo temporal + `fsync` + `os.replace`) para evitar que se corrompa el estado si un *worker* se cae mientras está guardando la sesión.

Protocolo de mensajes

- **IDs de mensaje únicos por *worker***: Cada mensaje tiene un identificador único. Los *workers* almacenan qué `message_id` ya procesaron para detectar duplicados cuando un mensaje se reenvía después de un fallo.
- **Orden de operaciones garantizado**: Los *workers* deben seguir estrictamente la secuencia:

  $$\text{procesar mensaje} \rightarrow \text{guardar estado} \rightarrow \text{confirmar recepción (ACK)}.$$

  De este modo, si un *worker* se cae antes de confirmar, el mensaje se reenvía, pero se detecta como duplicado gracias al `message_id` ya persistido.
- **Ruteo consistente**: Los mensajes con la misma `routing key` deben llegar siempre al mismo *worker*. Esto es necesario para la deduplicación, ya que cada *worker* solo conoce el conjunto de mensajes que él mismo procesó.

Coordinación y recuperación

- ***Heartbeats* periódicos**: Los *workers* envían *heartbeats* al **HealthChecker** para indicar que están funcionando. Si dejan de llegar *heartbeats* dentro del *timeout* configurado, el **HealthChecker** asume que el *worker* falló y procede a reiniciarlo.
- **Recuperación automática**: Cuando un *worker* se reinicia, carga automáticamente su último estado guardado desde disco y continúa procesando desde donde quedó, evitando pérdida de datos y reordenamientos inconsistentes.

6.2. Algoritmo de resiliencia de *workers*

El sistema implementa un mecanismo de persistencia de sesiones que garantiza la recuperación ante fallos. Cada *worker* mantiene un objeto `Session` que almacena el estado de procesamiento por sesión (incluyendo `EOF` recibidos, `ID` de mensajes procesados y datos específicos del *worker*). Al inicio, cada *worker* intenta cargar sesiones previas desde el disco, lo que le permite retomar la ejecución desde el último *checkpoint*.

La deduplicación de mensajes se implementa mediante un conjunto `msgs_received`, que almacena los `ID` de todos los mensajes ya procesados. Cuando llega un nuevo mensaje, primero se verifica si su `ID` ya existe en este conjunto; en ese caso, el mensaje se descarta inmediatamente, pero de todos modos se envía el `ACK` correspondiente a RabbitMQ.

El orden de operaciones es crítico para la *correctness* del sistema:

1. Se procesa el mensaje y se actualiza el estado en memoria.
2. Se persiste la sesión en disco mediante una escritura atómica (archivo temporal + `fsync` + `os.replace`).
3. Solo después de persistir el estado se envía el `ACK` a RabbitMQ.

Con este esquema, si un *worker* falla entre el procesamiento y el envío del `ACK`, RabbitMQ volverá a entregar el mensaje; sin embargo, este será detectado como duplicado gracias al `message_id` ya persistido en la sesión. La implementación actual utiliza `SnapshotFileSessionStorage`, que escribe el estado completo en cada `save`. Esto introduce cierta *overhead* en *workers* con estados grandes (por ejemplo, agregadores), pero garantiza atomicidad y simplicidad en la recuperación.

La coordinación de `EOF` entre réplicas también forma parte de la sesión. Los *workers* no líderes esperan un `EOF` proveniente del *exchange upstream*, mientras que el líder espera tanto su propio `EOF` como los `WorkerEOF` enviados por todas las réplicas de su *stage*. Este estado asociado a los `EOF` se persiste junto con el resto de la sesión, asegurando que no se pierdan datos al reiniciar *workers* y que la propagación del fin de sesión se mantenga consistente.

6.3. Coordinación de *workers* dentro de cada *stage*

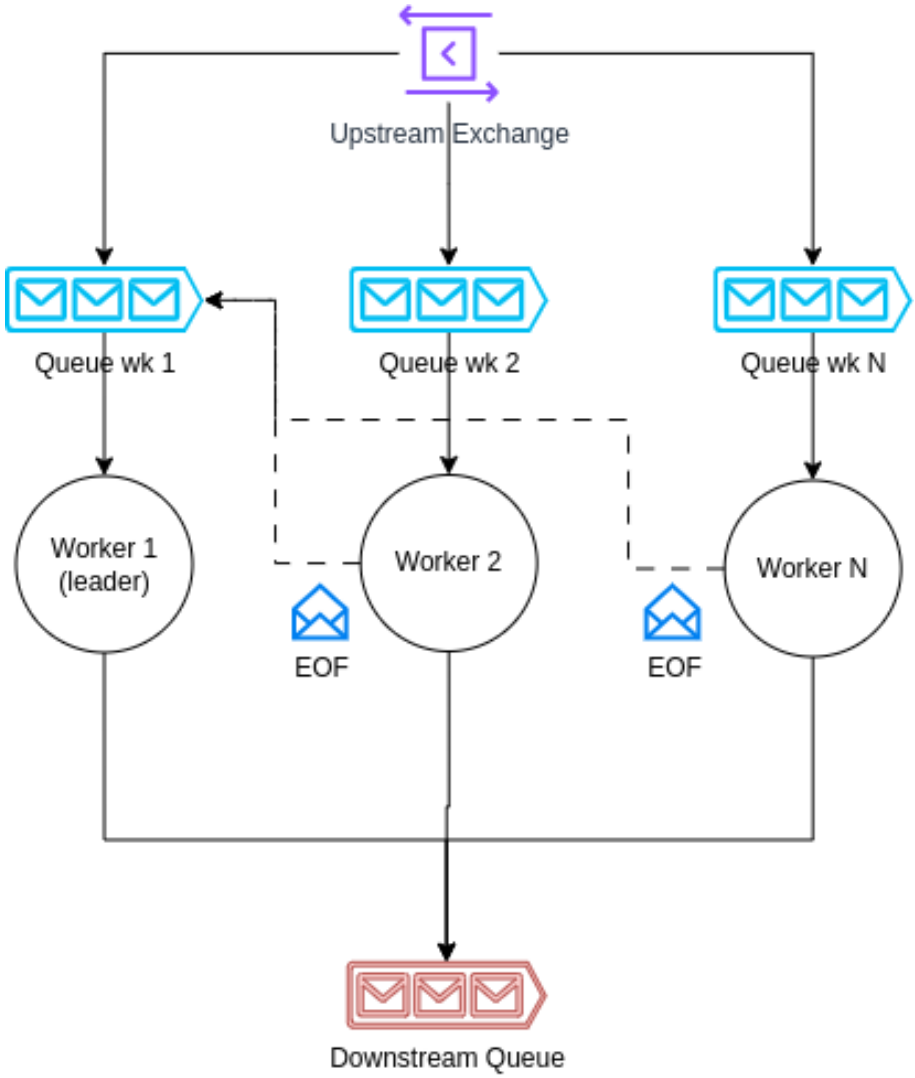

Figura 9: Coordinación de *workers* dentro de una etapa del *pipeline*.

Cada etapa del *pipeline* se implementa como un conjunto de *workers* que consumen mensajes desde un *exchange* de entrada. Para cada *worker* `wk_i` se define una cola exclusiva (*Queue wk i*) asociada a una *routing key* específica, y además todas las colas se suscriben a una *routing key* común denominada `common`. De este modo, el *exchange* puede:

- Enviar mensajes "shardeados" a un único *worker*, utilizando su *routing key* dedicada.
- Difundir mensajes a todos los *workers* de la etapa, utilizando la *routing key* `common`.

El manejo del fin de una sesión (`EOF`) se coordina de la siguiente manera:

1. El *stage* anterior, al finalizar una sesión, publica mensajes `EOF` hacia la etapa actual. Cada *worker* recibe el `EOF` correspondiente a las particiones que procesó.
2. Cuando un *worker* **no líder** recibe el `EOF` de una sesión:
   - **Flushea** sus buffers y envía todos los resultados pendientes hacia la cola de salida del siguiente *stage*.

- Envía una **notificación de EOF recibido** al *leader* del stage. Para ello publica un mensaje en el *exchange* usando la *routing key* asociada a la cola del *leader*, de manera que dicho mensaje queda shardeado exclusivamente hacia ese *worker*.

3. Dentro de cada etapa el worker con id 0 es el leader. Este *leader*:
   - Procesa sus propios datos normalmente y también recibe su propio `EOF`.
   - Recibe, en su cola, las notificaciones de `EOF` que envían el resto de los *workers* del *stage*.
   - Mantiene un contador por sesión con la cantidad de *workers* que ya confirmaron haber recibido `EOF`.
4. Cuando el *leader* detecta que:
   a) Todos los *workers* de la etapa notificaron haber recibido `EOF` para esa sesión, y
   b) Él mismo ya recibió su propio `EOF`,

   entonces **flushea** sus propios buffers y publica un **único mensaje** `EOF` hacia el siguiente *stage*, utilizando la *routing key* `common` del *exchange* de salida.

   Con este esquema, cada etapa garantiza que:

- Todos los resultados parciales fueron flusheados antes de propagar el `EOF`.
- El siguiente *stage* recibe exactamente **un** `EOF` por sesión, independientemente de la cantidad de *workers* internos.
- La lógica de coordinación se mantiene homogénea a lo largo de todo el *pipeline*, facilitando el escalado horizontal de *workers* y el manejo de sesiones concurrentes.

6.4. HealthChecker

El **HealthChecker** es un servicio distribuido de monitoreo y recuperación automática de contenedores que garantiza la alta disponibilidad del *pipeline*. Está compuesto por múltiples réplicas que implementan el algoritmo de elección de líder *Bully* para coordinar las acciones de recuperación sin introducir un punto único de falla.

Cada *worker* envía *heartbeats* periódicos vía UDP a todas las réplicas del **HealthChecker**, reportando su estado activo. El protocolo UDP se eligió por su bajo *overhead* y su naturaleza *stateless*: los *heartbeats* son mensajes *fire-and-forget* que no requieren *acknowledgment*, y la pérdida ocasional de paquetes no es crítica, ya que los *heartbeats* son frecuentes (cada 5 segundos por defecto).

Por otro lado, la comunicación entre réplicas del **HealthChecker** (mensajes de elección, `OK` y `coordinator`) utiliza TCP, ya que requiere entrega garantizada y orden estricto para la *correctness* del algoritmo de elección. Solo la réplica líder ejecuta acciones de recuperación: reinicia contenedores mediante `docker start` cuando detecta que un *worker* no envió *heartbeats* dentro del *timeout* configurado. Si el líder falla, las demás réplicas detectan su ausencia mediante *timeouts* en los *heartbeats* entre *peers* y disparan una nueva elección.

Este sistema complementa el algoritmo de resiliencia por *worker* al manejar fallos catastróficos donde un *worker* se cuelga o se crashea completamente sin oportunidad de guardar estado. El **HealthChecker** reinicia el contenedor y, al arrancar, el *worker* recupera su último estado persistido desde disco, retomando el procesamiento desde el último mensaje que no recibió `ACK`. Además, el **HealthChecker** se auto-monitorea: cada réplica también revive otras réplicas caídas, garantizando que siempre haya al menos una instancia activa para mantener el sistema en funcionamiento.

6.5. Features exploradas pero no implementadas

Durante el desarrollo exploramos dos optimizaciones importantes que quedaron en *branches* experimentales, debido a restricciones de tiempo de *testing* previo a la entrega.

La *branch* `feature/stateless` (https://github.com/saantim/tp1-distribuidos/tree/feature/stateless) implementa *transformers* y *filters* que no persisten estado intermedio a disco. La idea es reconocer que estos *workers* son inherentemente *stateless*: solo aplican funciones puras (parseo CSV → entidad, o predicado de filtrado) sin acumular datos entre mensajes. La implementación mantiene un *buffer* en memoria que se flushea inmediatamente después de procesar cada *batch*, usando un *hook* `_after_batch_processed` que vacía el *buffer* antes del `save`. Esto eliminaría completamente la escritura a disco en el *path* crítico de estos *workers*, reduciendo latencia e I/O innecesario.

La *branch* `feature/WAL-Stateless` (https://github.com/saantim/tp1-distribuidos/tree/feature/WAL-Stateless) implementa un sistema de *Write-Ahead Log* (WAL) con compactación periódica mediante *snapshots* (`worker/storage/wal.py`). En lugar de escribir el estado completo en cada `save` (problema crítico en agregadores con 100k+ *items*), el WAL solo hace *append* de operaciones al *log* (por ejemplo, `AggregateOp` con *key/value* incremental). Las escrituras son mucho más rápidas (solo *append*, sin reescribir el estado completo) y la recuperación funciona *replayeando* el *log* sobre el último *snapshot*. Cada N *batches* procesados se crea un nuevo *snapshot* y se trunca el *log* para prevenir un crecimiento indefinido. Esta implementación define operaciones tipadas (subclases de `BaseOp`) y un patrón de *reducer*, similar a Redux, para aplicar operaciones al estado durante la recuperación.

Ambas *features* habrían mejorado significativamente la *performance* en el *dataset* (reduciendo el *bottleneck* de I/O en *filters*/*transformers* y en agregadores, respectivamente), pero no pudimos completar el *testing* de integración necesario para validar la *correctness* ante fallos concurrentes. Por este motivo, decidimos entregar con la implementación conservadora y probada basada en *snapshots* completos.