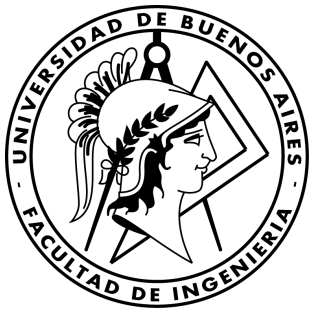

Trabajo Práctico: Multi-Client

[TA050] Sistemas Distribuidos I
Segundo cuatrimestre de 2025

Alumno	**Nombre y apellido**	**Padrón**	**Mail**
	Baratta, Facundo	104486	fbaratta@fi.uba.ar
	KIM, Daniel	107188	dakim@fi.uba.ar
	Koo, Hangyeol	108401	hkoo@fi.uba.ar

Fecha de entrega: 16/10/2025

Índice

1. Diagramas

1.1. DAG (Directed Acyclic Graph)

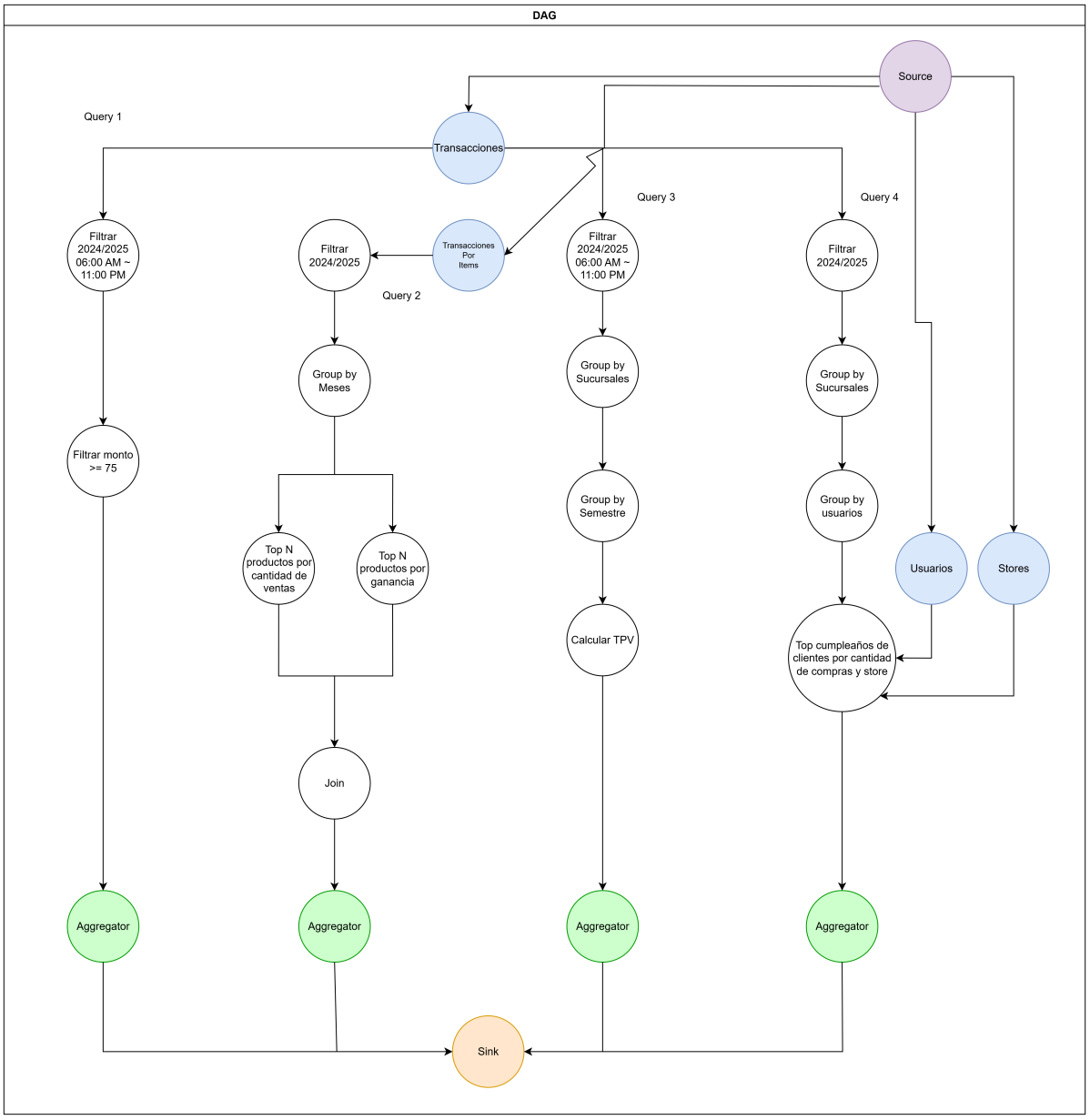

Figura 1: DAG

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

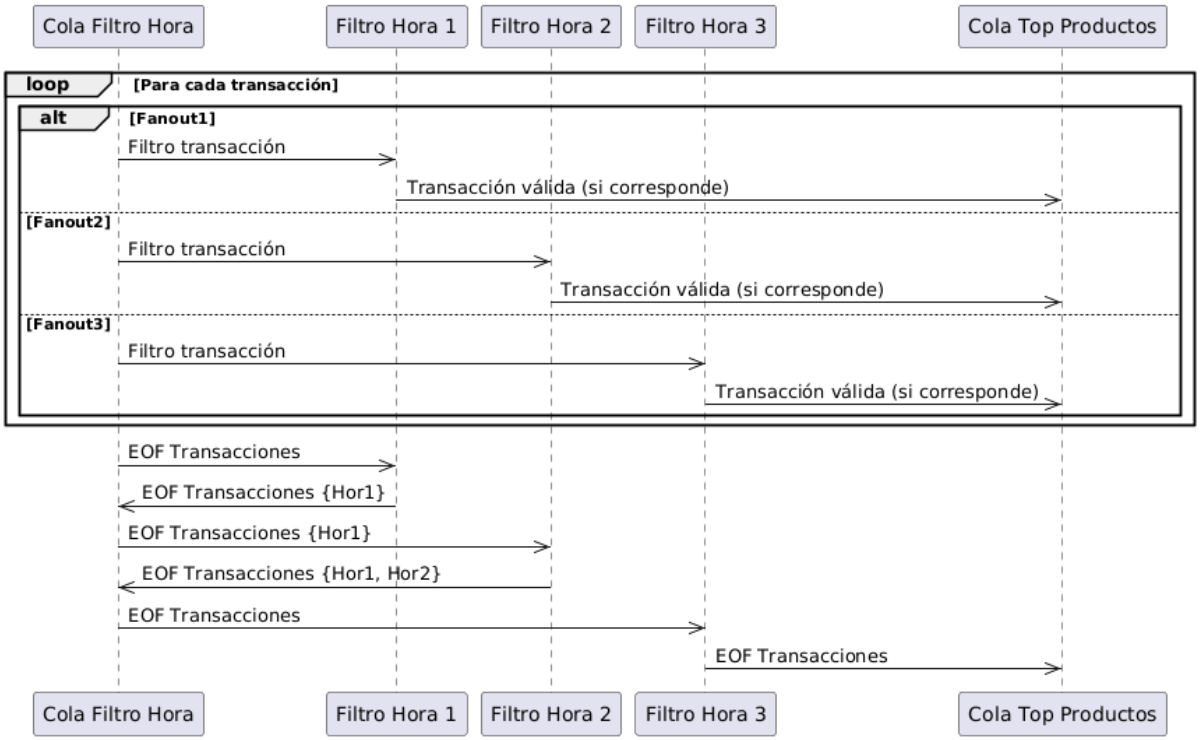

Figura 2: Diagrama de secuencia de manejo de EOF

En el presente diagrama es posible observar el flujo del manejo de EOF para garantizar que todos los workers terminaron de procesar los registros recibidos:

- 1. Se envían todos los registros a los filtros
- 2. Se envía el EOF a alguno de los filtros
- 3. El filtro reinserta el EOF insertándo su id
- 4. El resto de los filtros repiten lo mismo hasta que ven todos los demás ids. Si es ese el caso, se envía a la cola del siguiente worker para procesar la query.

1.3. Diagrama de Actividades

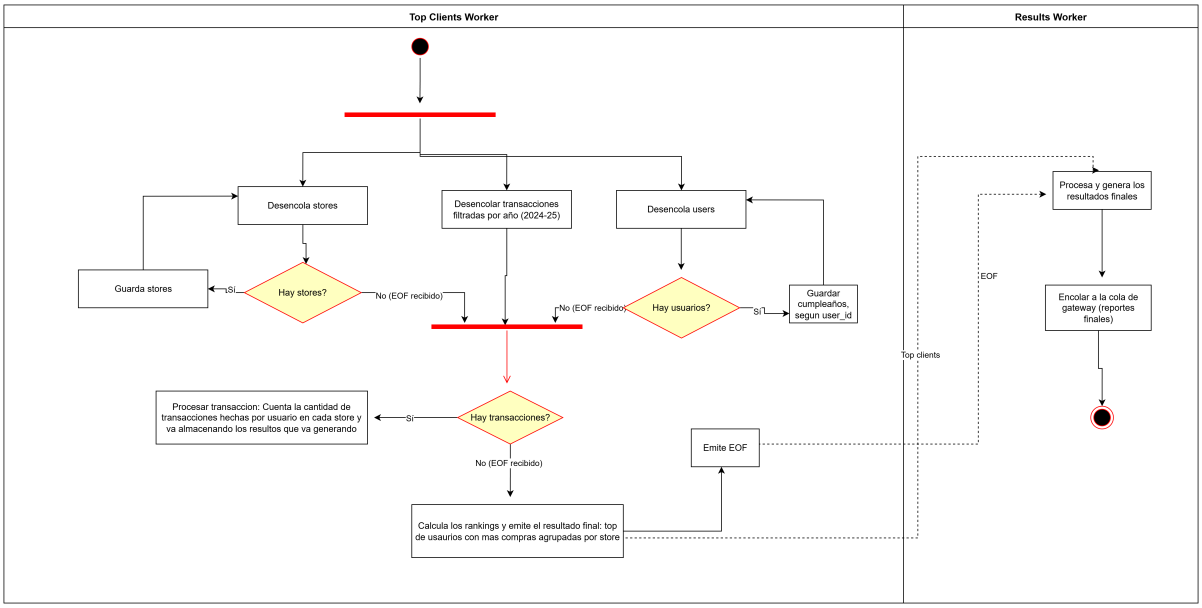

Figura 3: Diagrama de actividades

Este diagrama corresponde a las fechas de cumpleaños de los tres clientes que han hecho más compras durante 2024 y 2025, para cada sucursal. Se puede ver que el worker se encarga de guardar en memoria los stores y usuarios para después contar la cantidad de transacciones hechas por usuario en cada store y va almacenando los resultados que va generando, para luego enviarle los resultados finales al results worker.

1.4. Diagrama de Robustez

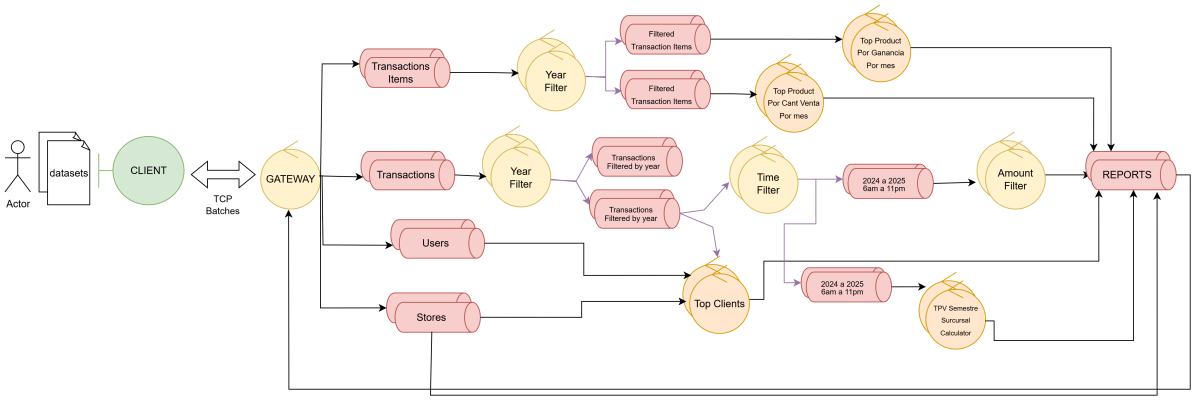

Figura 4: Diagrama de Robustez

En este diagrama se explica el sistema interno, mostrando las colas, los controladores y como interactúan entre ellos. En primer lugar, el gateway recibe el dataset y, adelantándonos al final de todo, recibe el reporte de todos los queries.

Nuestra idea es un sistema de forma "caudal de agua" donde algunos workers (en particular el filtro por año 2024-2025) está repetido, pero esto es porque toma datos de una cola distinta y encola el resultado del procesamiento en otras colas. Las flechas violetas indican que el resultado es duplicado en dos colas que

reciben información idéntica pero son utilizada por controladores que se encargan de queries distintas, desencolando ítems a su necesidad.

Finalmente, hay algunos controladores que tienen un estado interno, ya que encuentran el top productos/clientes/ventas por sucursal, mes, y/o semestre. Se encarga de hacer el computo de la agregación de una agrupación por las columnas necesarias para resolver cada query en particular.

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.

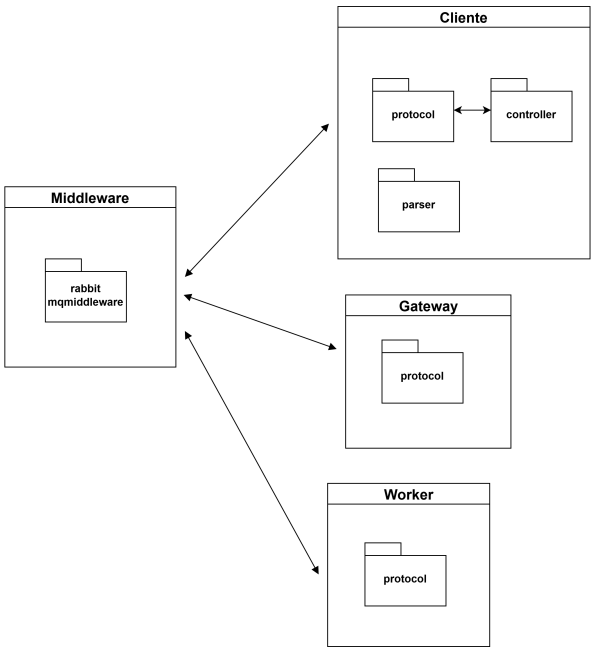

Figura 5: Diagrama de paquetes

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.

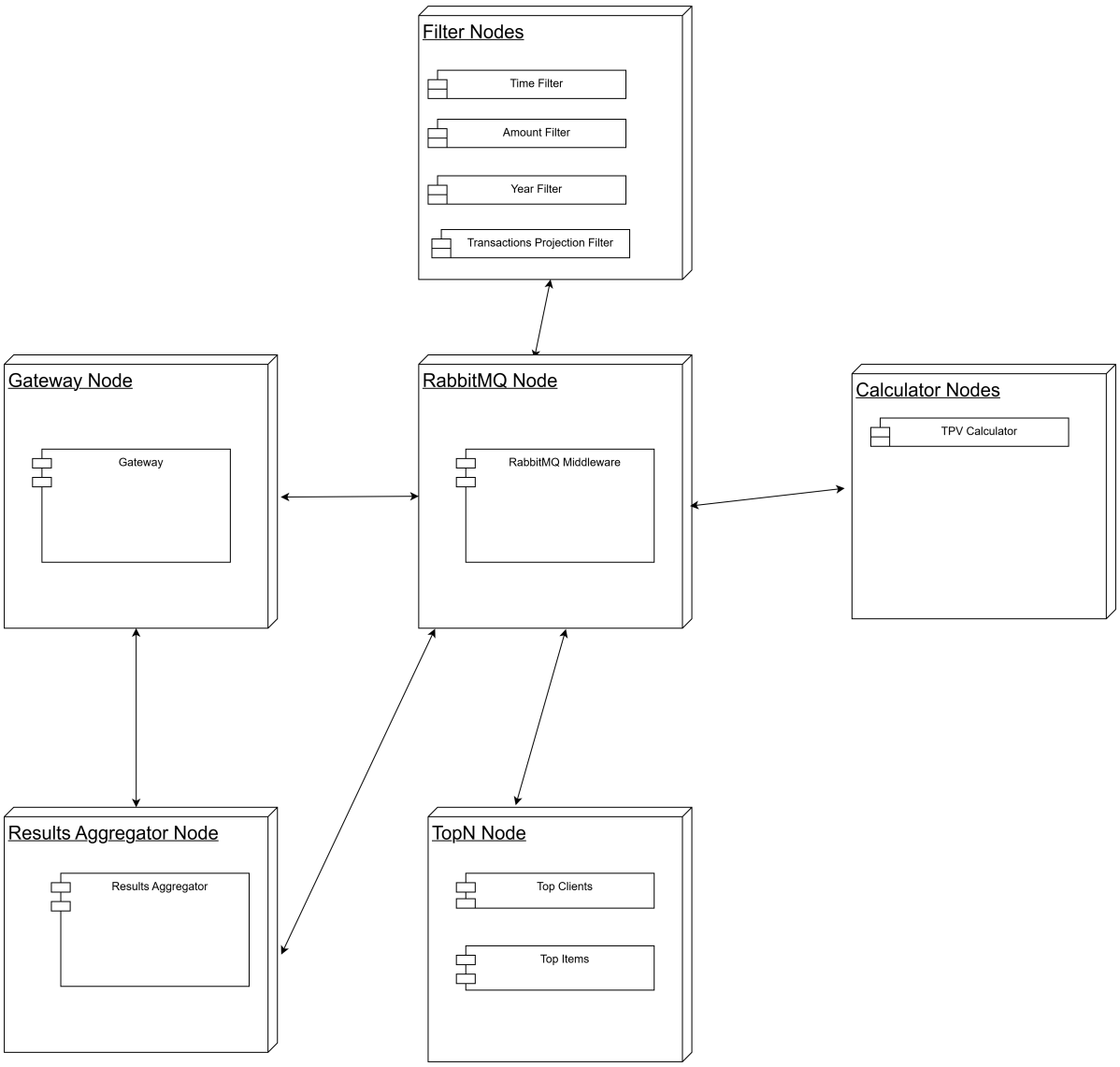

Figura 6: Diagrama de despliegue

2. Desarrollo

2.1. 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.

2.2. Cómo manejamos los EOF's

2.2.1. 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.

2.2.2. 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.

2.2.3. 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.

2.2.4. 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.

2.3. 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.

2.4. Testing del broker

Los tests del broker se encuentran en el directorio `/scripts/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 scripts/tests/test_middleware.py`.

2.4.1. 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.

2.5. Anexo: Mejoras pendientes

- En el worker de **Top de usuarios con más compras**, el join entre usuarios y tiendas aún se hace dentro del mismo proceso.
- Falta agregar **sharding del top global**, aunque ya se logró escalar el top por agregación previa entre réplicas.
- No se guarda en disco data complementaria para una query ya que no es elemental para esta entrega; demora mucho la ejecución

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**.