Informe de Trabajo Práctico

Sistemas Distribuidos I

Integrantes:

Mateo Alvarez - 108666

Tomás Szejnfeld Sirkis - 107710

Martín González Prieto - 105738

Alcance

Requerimientos Funcionales

- Se solicita un sistema distribuido que analice la información de ventas en una cadena de negocios de Cafés en Malasia.
- Se cuenta con información transaccional por ventas (montos, items vendidos, etc), información de los clientes, de las tiendas y de los productos ofrecidos.
- Queries a hacer:
    - 1. Transacciones (Id y monto) realizadas durante 2024 y 2025 entre las 06:00 AM y las 11:00 PM con monto total mayor o igual a 75.
    - 2. Productos más vendidos (nombre y cant) y productos que más ganancias han generado (nombre y monto), para cada mes en 2024 y 2025.
    - 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.
    - 4. Fecha de cumpleaños de los 3 clientes que han hecho más compras durante 2024 y 2025, para cada sucursal.

Requerimientos No Funcionales

- El sistema debe estar optimizado para entornos multicomputadoras
- Se debe soportar el incremento de los elementos de cómputo para escalar los volúmenes de información a procesar

- Se requiere del desarrollo de un Middleware para abstraer la comunicación basada en grupos.
- Se debe soportar una única ejecución del procesamiento y proveer graceful quit frente a señales SIGTERM.

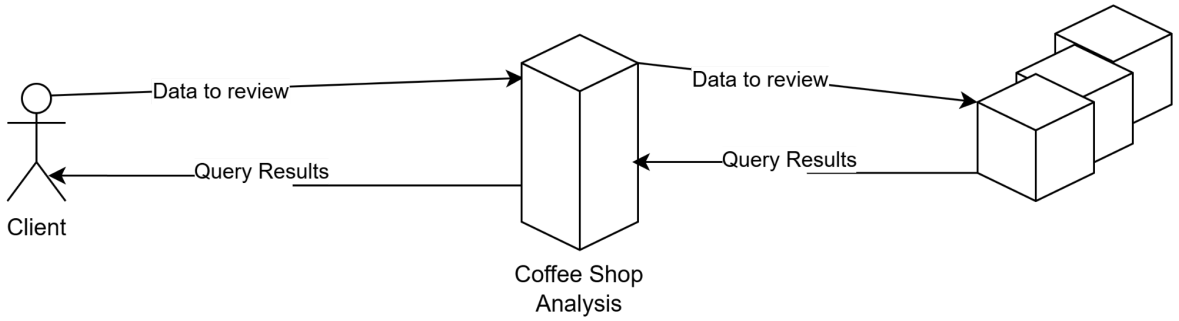
Escenarios

Casos de Uso

En el **Diagrama de Casos de Uso**, se identifican los siguientes elementos:

- **Actor Cliente**: Usuario que realiza las consultas al sistema, solicitando información para su análisis.
- **Servidor Coffee Shop Analyst**: Representa el sistema encargado de recibir y procesar los resultados de las consultas realizadas.
- Nodos de análisis: Estos serian los filtros a aplicar a los datos enviados
- **Casos de Uso (Pedido de Querys)**:
    - *R*epresenta los datos enviados y los distintos tipos de respuestas que el cliente puede recibir del sistema. Cada una de estas respuestas es enviada desde el sistema.

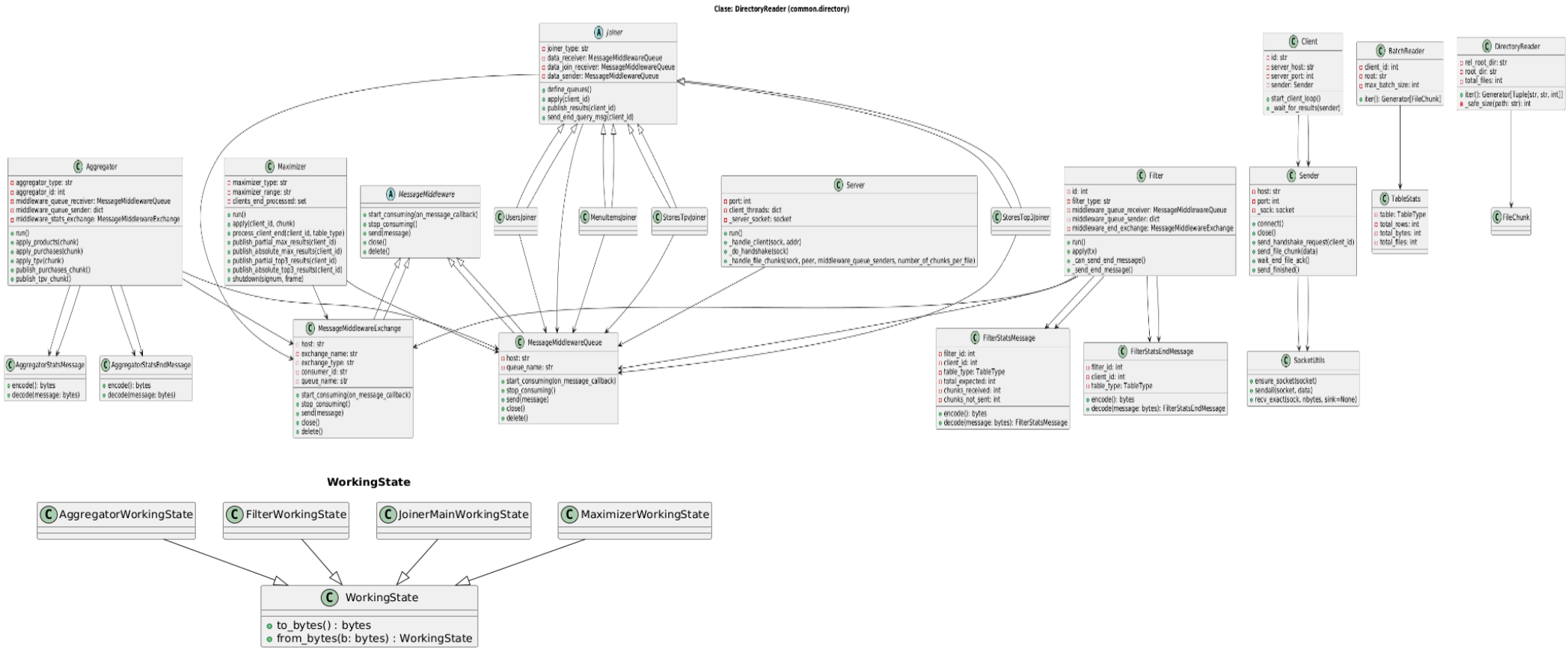
Los escenarios se centran en el flujo de información entre cliente y el sistema Coffee Shop Analysis, destacando cómo el sistema soporta múltiples tipos de consultas. Además, este modelo refuerza el rol del sistema como un **intermediario entre los datos y el análisis de negocio**, asegurando que la interacción sea clara, flexible y extensible a futuros tipos de consultas.



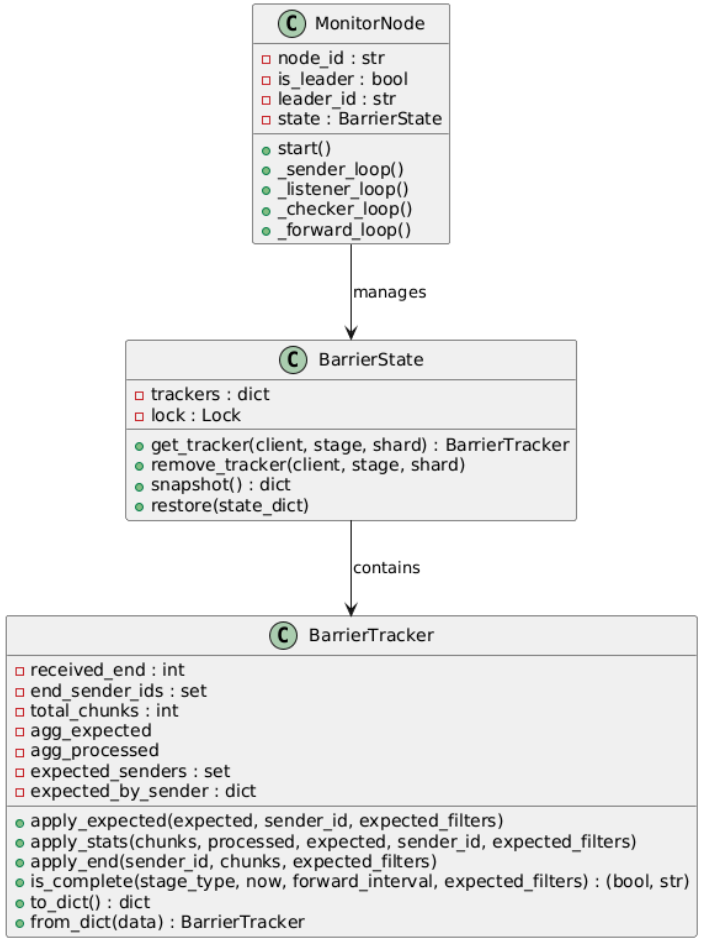
Vista Lógica

En esta vista se describen los principales módulos del sistema y sus responsabilidades dentro del procesamiento de consultas y manejo de archivos:

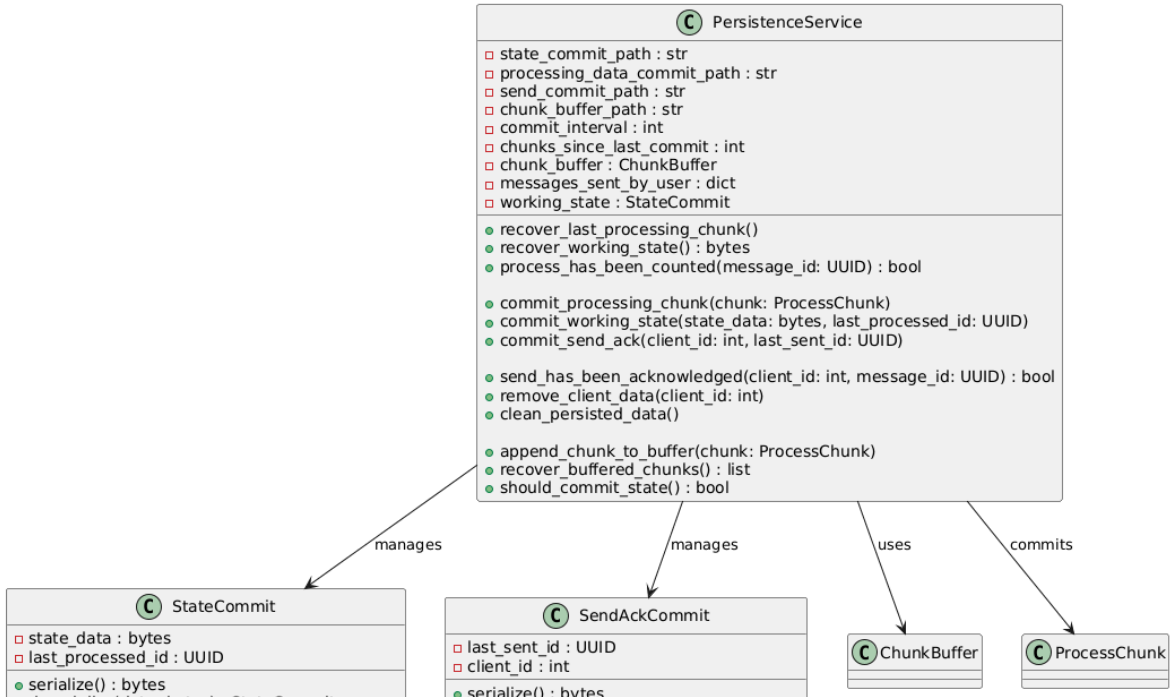
Clases



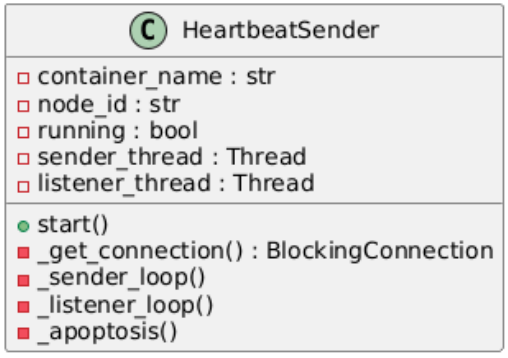
Monitor Node Classes



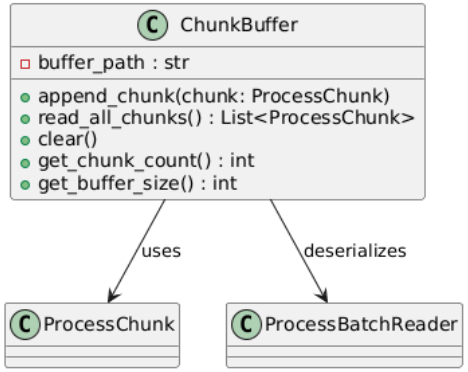
Persistence Service - Unified Classes



HeartbeatSender

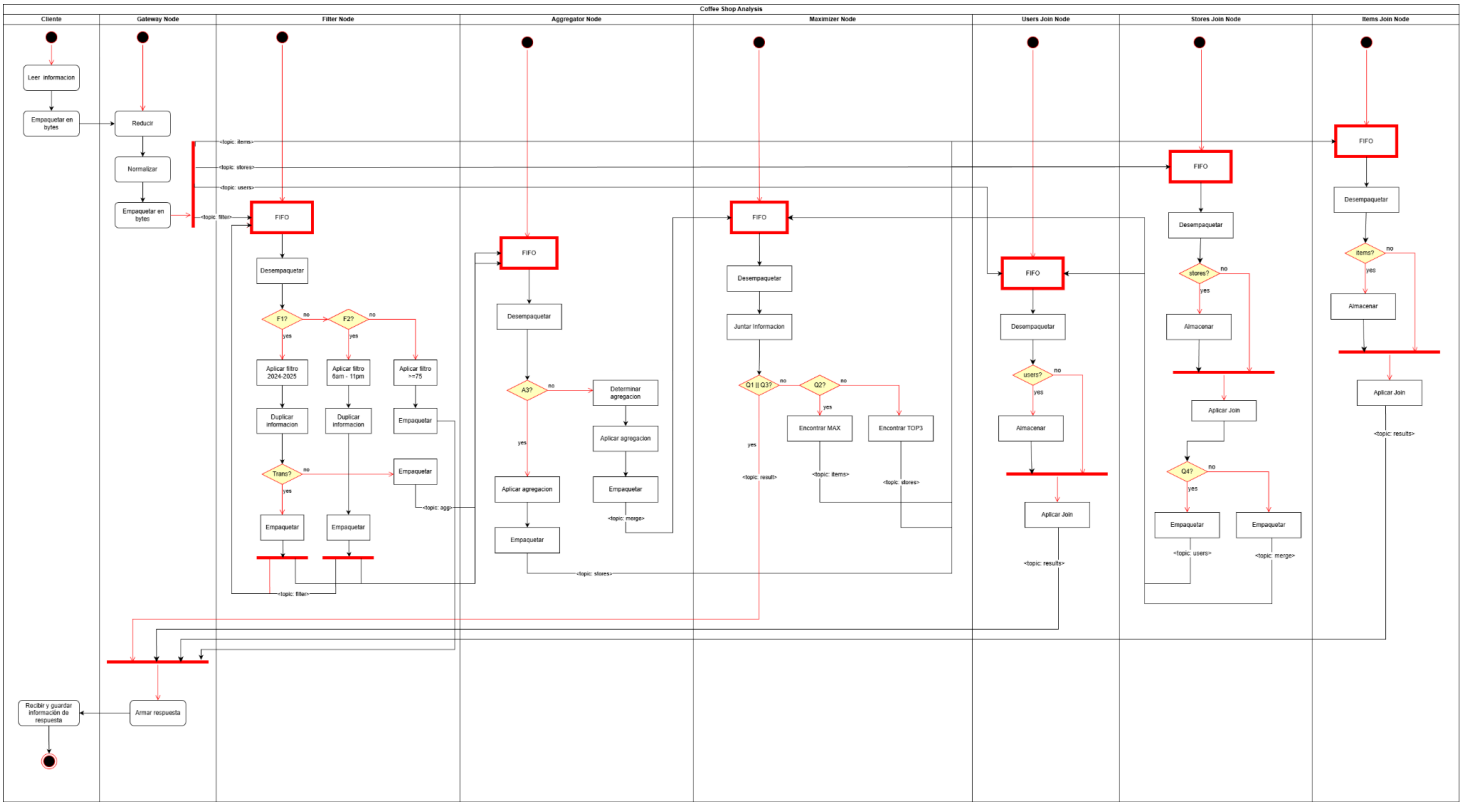


ChunkBuffer



Vista de Procesos

Actividades



El diagrama representa el **flujo completo de procesamiento distribuido** del sistema, organizado por nodos funcionales especializados. Cada carril describe las actividades que ocurren desde la recepción inicial de datos hasta la generación final de resultados.

El proceso inicia en el **Cliente**, donde la información es leída y empaquetada para ser enviada al **Server Node**. En este nodo se realiza una reducción y normalización de los datos, que luego se reempaquetan y distribuyen a los distintos tópicos del sistema.

Después tenemos los **Worker Nodes** (Filter, Joiner, Aggregator y Maximizer) que son los encargados de procesar los datos para lograr los resultados correctos.
Ejecutan las operaciones específicas dependiendo de cada query:

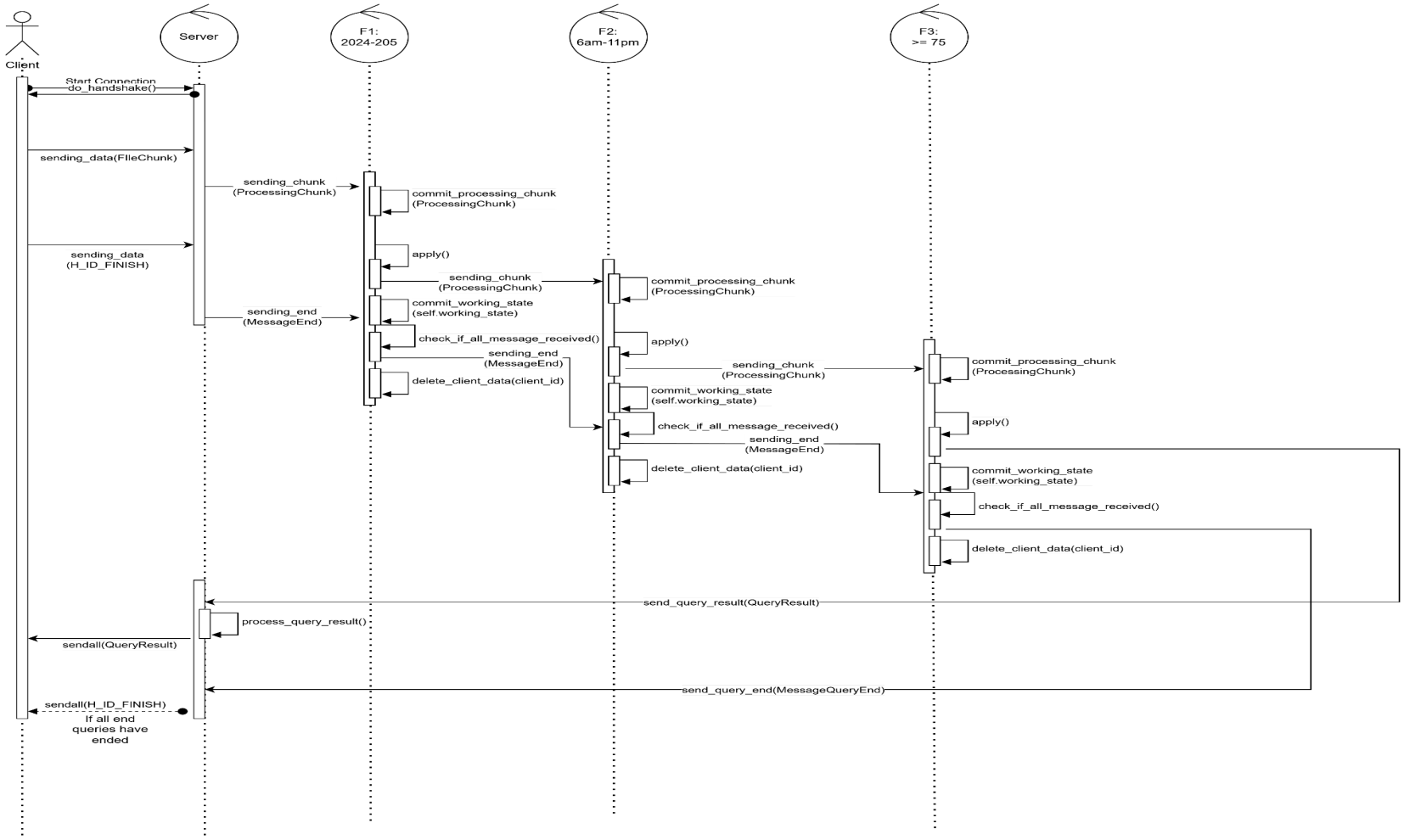
- Query 1: filtrar por fechas, horarios y montos, luego agregar montos.
- Query 2: calcular productos más vendidos y más rentables por mes.
- Query 3: calcular TPV semestral por sucursal en franjas horarias válidas.
- Query 4: contar compras por cliente, obtener top 3 y cruzar con datos de cumpleaños.

Después de procesar manda los resultados parciales al servidor.
El servidor reúne todos los resultados parciales, los consolida y los envía al cliente, quien los recibe y los presenta en pantalla, finalizando así el proceso.

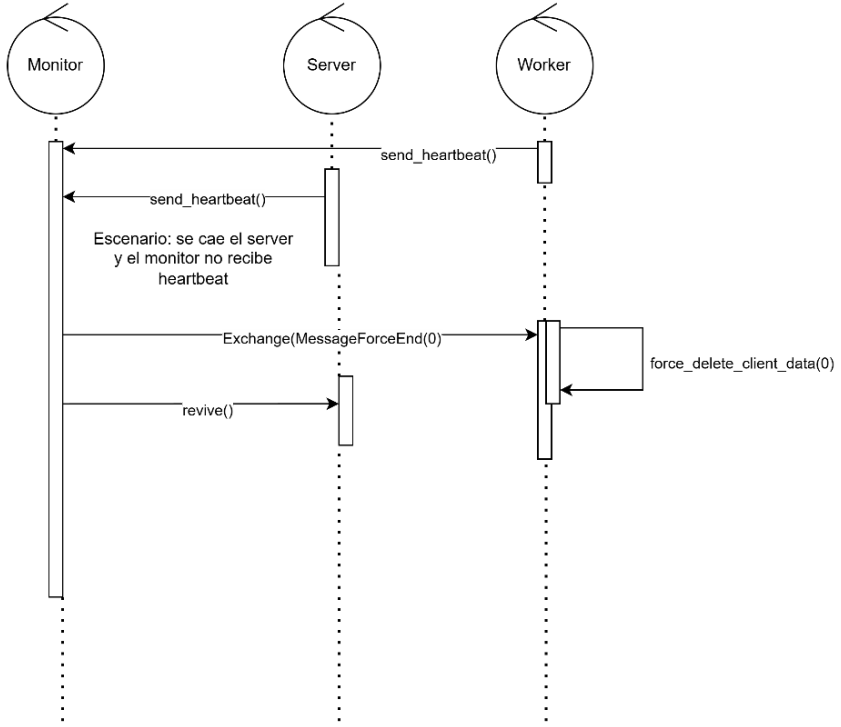
El Monitor Node, no presente en este diagrama, es verifica que esta actividad se pueda llevar a cabo con normalidad verificando que ningún nodo está caído y si lo está, poder levantarlo y que siga su ejecución como si nada hubiera pasado.

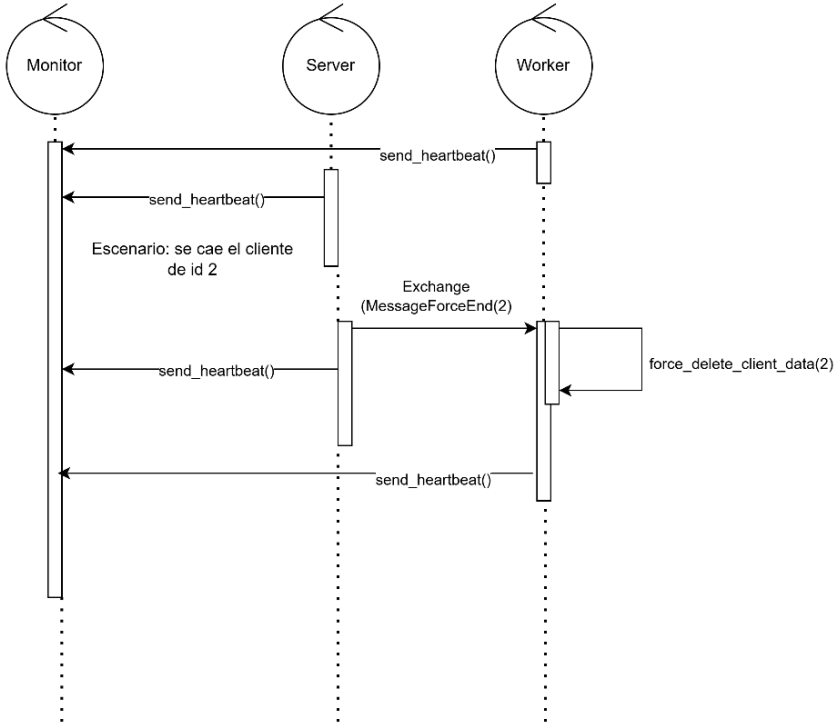
Secuencias

Query 1 – Transacciones filtradas por año, hora y monto



Escenarios de caída de servidor y cliente





Comunicación entre nodos del mensaje END

El manejo de **End** dentro del pipeline se basa en un mecanismo de conteo y sincronización que asegura que todos los chunks hayan sido procesados antes de permitir que el flujo continúe. El servidor inicialmente distribuye los chunks en distintos shards según su ID y, además de enviarlos, informa a cada shard cuántos chunks deberá procesar.

Cada **filter** mantiene un registro de cuántos chunks ha enviado a cada shard, y esa información fluye hacia abajo en el pipeline para que los siguientes nodos sepan exactamente cuánta data deben esperar posteriormente al llegado del End Message de la etapa anterior (del Servidor o de otros filtros). Paralelamente, los **filters** comunican al monitor la cantidad total de chunks que enviaron. Con esta información, el **monitor** calcula el número total de chunks que el sistema debería procesar, si cumple con ese número, envía a los filters que pueden enviar su End Message.

A medida que los **aggregators** reciben y procesan los chunks, le notifican al **monitor** cuántos han completado. El monitor compara continuamente el total procesado con el total esperado. En el momento exacto en que ambas cifras coinciden, el monitor envía un mensaje de "*barrier forward*" a los **aggregators**. Este mensaje actúa como una señal de cierre o end, indicando que se han procesado todos los chunks previstos y que ya pueden liberar la data acumulada hacia los siguientes componentes del sistema.

Finalmente, el maximizer espera recibir el end de todos los aggregators para poder completar su propio procesamiento, y el **joiner**, que es un nodo simple, continúa funcionando como antes: recibe toda la data del **maximizer** y detecta el fin del flujo una vez que recibe su señal de end. Sin embargo, el **joiner** para poder enviar su información al servidor debe contar con la tabla completa para joinear (Stores, Users, Products) y esto se determina nuevamente con el End Message que envía el servidor.

Algoritmos de Persistencia

La persistencia se implementa mediante una librería central llamada PersistenceService, utilizada por todos los workers para garantizar tolerancia a fallos y recuperación consistente. Su función principal es almacenar tres elementos fundamentales de cada worker:

1. **Su estado de ejecución** (WorkingState),
2. **El último dato en procesamiento**
3. **Los envíos ya realizados**

Cada uno de estos elementos se guarda en archivos independientes mediante operaciones atómicas, lo que asegura que una escritura nunca queda en un estado parcial: o se escribe por completo, o no se aplica. Esto se logra generando archivos temporales y usando un rename atómico, de modo que ante un crash nunca queda un estado inconsistente.

Cuando un worker procesa un mensaje, sigue siempre la misma secuencia lógica:

1. **Commit de recepción**: se persiste el chunk recibido, indicando “este mensaje llegó para ser procesado”.
2. **Procesamiento**: el worker aplica su lógica (filtrado, agregación, maximización o join).
3. **Commit de estado**: se persiste el WorkingState actualizado luego del procesamiento.
4. **Envío del resultado**.
5. **Commit de envío**: se registra que el mensaje ya fue enviado, para evitar duplicados.

Durante la recuperación, cada worker revisa estos archivos: si existe un chunk recibido pero no enviado, lo reprocesa y reenvía; si el estado estaba guardado, lo restaura; si un envío estaba registrado, evita reenviarlo. De esta forma el sistema garantiza idempotencia y consistencia incluso si falla en cualquier punto de la secuencia.

Persistencia atómica

La persistencia atómica asegura que un commit de estado o dato se refleja sólo si se completó totalmente. Si ocurre un fallo durante la escritura, el worker conserva el estado previo intacto. Esto evita estados incompletos o combinaciones imposibles, y permite que la recuperación sea determinista: el worker siempre “vuelve” al último commit válido.

La librería implementa dos primitivas internas:

```
atomic_file_upsert(path, data)
    write temp_file
    fsync(temp_file)
    rename(temp_file → path)        # operación atómica del filesystem

atomic_file_append(path, data)
    read old_data
    combined = old_data + data
    atomic_file_upsert(path, combined)
```

Algoritmos de persistencia por tipo de worker

Aunque cada worker realiza tareas distintas, todos siguen el mismo patrón de interacción con la librería. La diferencia está en qué significa su WorkingState.

1. Filter Worker

Su estado es simple: únicamente registra qué mensajes ya procesó por cada cliente.

Procesamiento

```
on_message(x):
    commit_processing(x)
    #ACK Cola Rabbit
    r = filtrar(x)
    processed_ids.add(x.id)
    commit_state(processed_ids)
    enviar(r)
    commit_send(x)
```

Recuperación

```
recover():
    state = load_state()
    x = load_last_processing()
    if x and x.id not in state.processed_ids:
        r = filtrar(x)
        commit_state(state + x.id)
        enviar(r)
        commit_send(x)
```

2. Aggregator Worker

Además del set de procesados, su estado mantiene acumuladores y banderas de END. Usa un buffer de chunks porque trata de evitar la constante escritura a disco de un estado que puede crecer de acuerdo a la información. Su WorkingState contiene:

- acumuladores por cliente/tabla,
- contadores de chunks recibidos,
- flags de END,
- set de IDs ya procesados.

Procesamiento

```
on_message(x):
    commit_processing(x)
    #ACK Cola Rabbit

    if x.is_data:
        buffer.append(x)
        apply_aggregation(x)
    else:   # END
        marcar_END(x)

    if interval_reached:
        commit_state(acumuladores, contadores, ends, processed_ids)
        buffer.clear()

    enviar_si_corresponde(...)
    commit_send(x)
```

Recuperación

```
recover():
    state = load_state()

    # chunk interrumpido
    x = load_last_processing()
    if x and x.id not in state.processed_ids:
        apply_aggregation(x)

    # chunks del buffer
    for b in load_buffer():
        if b.id not in state.processed_ids:
            apply_aggregation(b)

    commit_state(state_actualizado)
    reenviar_ENDs_o_resultados_pendientes()
```

En la recuperación procesa primero los chunks del buffer, luego reenvía END o resultados pendientes.

3. Maximizer Worker

Su WorkingState contiene los máximos globales y los senders ya finalizados. No usa buffer: cada chunk actualiza inmediatamente el estado. Su WorkingState almacena:

- máximos globales de ventas/ganancias,
- sets de IDs procesados,
- senders terminados,
- clientes finalizados.

Procesamiento

```
on_message(x):
    commit_processing(x)
    #ACK Cola Rabbit
    actualizar_maximos(x)
    processed_ids.add(x.id)
    commit_state(maximos, processed_ids, senders)
    clear_processing_commit()
    enviar_parcial_si_corresponde(...)
    commit_send(x)
```

Recuperación

```
recover():
    state = load_state()
    x = load_last_processing()

    if x and x.id not in state.processed_ids:
        actualizar_maximos(x)
        commit_state()

    if todos_senders_terminaron(state) and not state.cliente_finalizado:
        finalizar_cliente(state)
```

En la recuperación, si había un chunk sin terminar, lo reprocesa; si existen END pendientes, completa la finalización.

4. Joiner Worker

Es el más complejo porque combina dos flujos independientes. Por eso tiene dos WorkingStates: uno para los datos principales (chunks y ENDs provenientes del maximizer, set de procesados, senders finalizados, resultados de join) y otro para la tabla de join (tabla auxiliar de join (productos, usuarios, etc.), set de procesados). Cada estado persiste por separado.

Procesamiento

```
on_main(x):
    commit_processing_main(x)
    #ACK Cola Rabbit
    guardar_chunk(x)
    marcar_procesado_main(x)
    commit_state_main()
    if datos_join_listos: ejecutar_join()

on_join(x):
    commit_processing_join(x)
    #ACK Cola Rabbit
    actualizar_tabla_aux(x)
    marcar_procesado_join(x)
    commit_state_join()
    if hay_datos_main_pendientes: ejecutar_join()
```

Recuperación

```
recover():
    main = load_state_main()
    join = load_state_join()

    x_main = load_last_processing_main()
    if x_main and x_main.id not in main.processed:
        guardar_chunk(x_main)

    x_join = load_last_processing_join()
    if x_join and x_join.id not in join.processed:
        actualizar_tabla_aux(x_join)

    # intentar completar joins pendientes
    if datos_aux_completos and hay_chunks_main:
        ejecutar_join()
        commit_state_main()
    reenviar_ENDs_pendientes()
```

En la recuperación retoma ambos estados, reprocesa los últimos chunks de cada flujo, y completa joins o envíos de END que hayan quedado pendientes.

Algoritmo de Elección de Líder

Para los monitores, se decidió implementar un algoritmo de elección de líder. El líder es el que hace los trabajos de revivir nodos caídos. Pero si se cae un monitor, tenemos que garantizar que se vuelva a levantar. Para eso, distribuimos el nodo y hacemos que si otro de los monitores detecta la caída del líder, hace un llamado a elección y utilizando el algoritmo de bully, decide el nuevo líder mientras el anterior líder se levanta.

Protocolo Force End

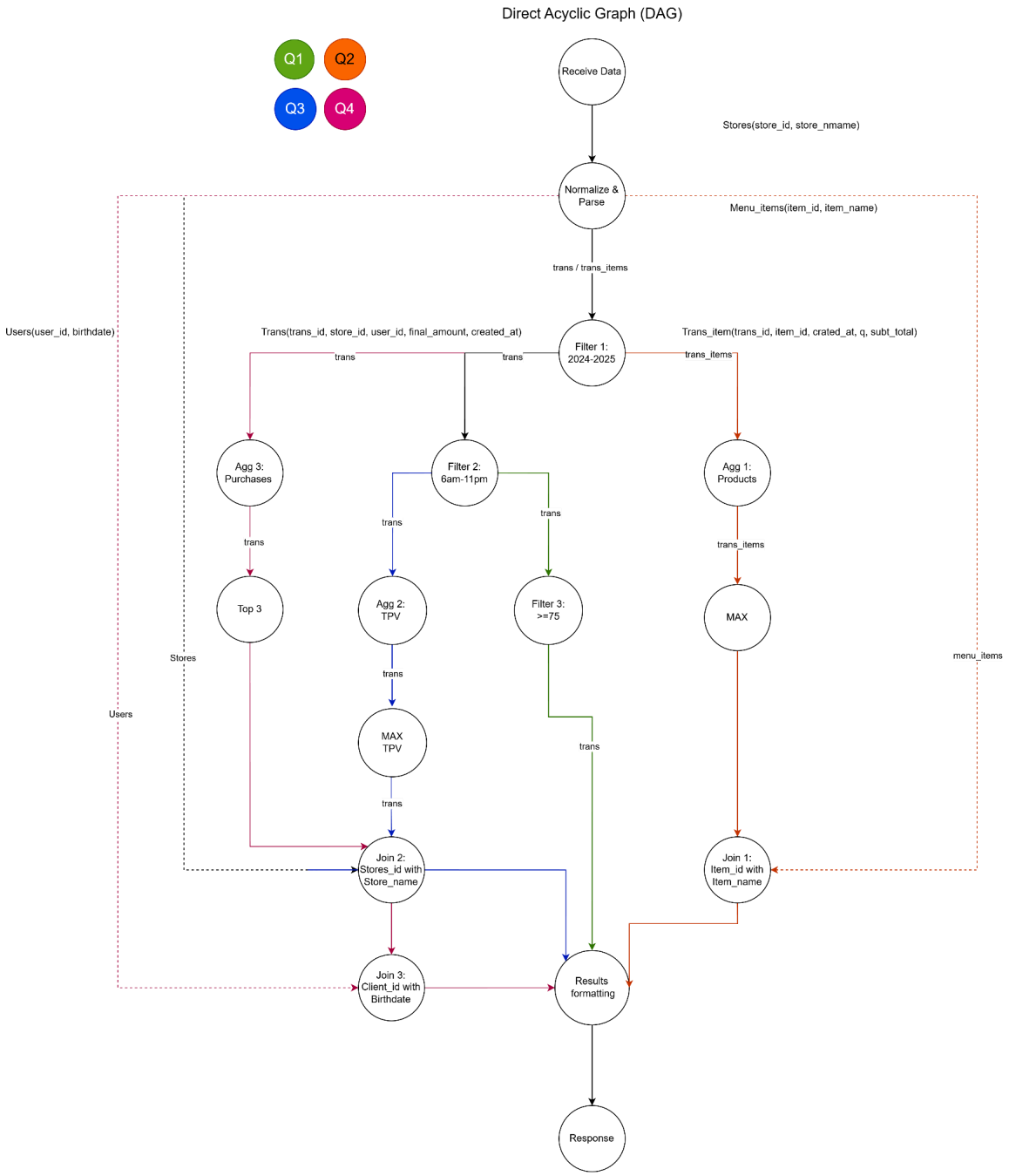
El protocolo del Force End es un protocolo sencillo para que, si se cae un **Cliente** o el **Server**, borrar mensajes de clientes que ya no van a usar.

La idea base es que si el servidor detecta que se cayó la conexión TCP con el cliente, el servidor envía un mensaje de Force End con el id del cliente vía exchange fanout a todos los trabajadores para decirles que borren la data de ese cliente específico y que si llegan chunks de data relacionada a ese cliente, la descarte.

Pero si se cae el servidor, el monitor al momento de detectar que el servidor no responde, envía un mensaje de Force End via el mismo exchange con id -1 para decirles que descarten todos los mensajes de los clientes actuales y descarten todo mensaje que les lleguen relacionado a ellos.

DAG

Los **DAGs (Direct Acyclic Graphs)** representan la ejecución de consultas como un flujo de pasos secuenciales y ramificados. Cada consulta se modela como una cadena de transformaciones de datos: primero se recibe y normaliza la información, luego se aplican filtros, agregaciones o uniones, y finalmente se formatea la respuesta.



Vista de Desarrollo

Paquetes

Diagrama de Paquetes

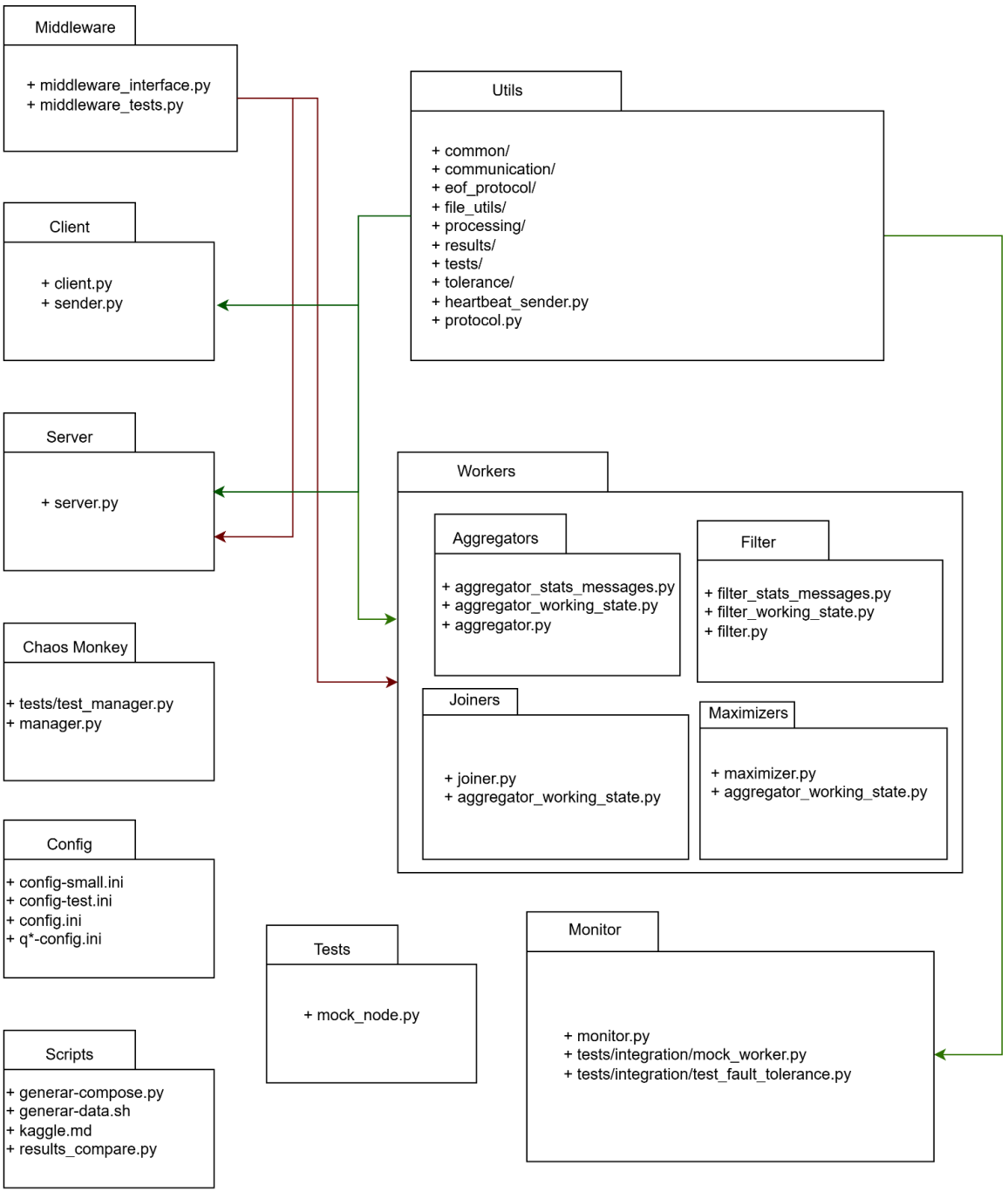
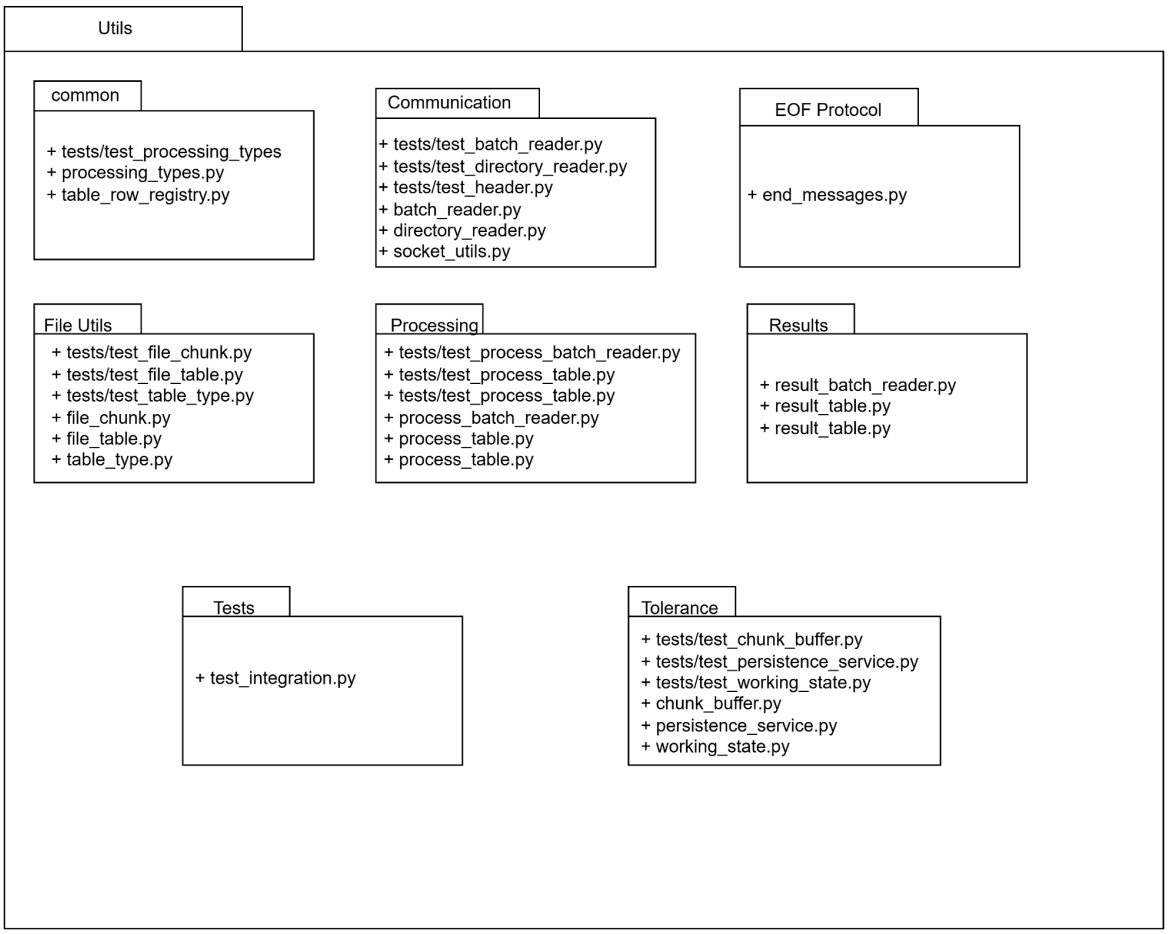


Diagrama de Paquetes - Utils



En la parte superior se encuentra el paquete **Utils**, que contiene todos los submódulos de uso general en todo el proyecto como por ejemplo todo lo que tiene que ver con la tolerancia o la comunicación TCP.

El paquete **Middleware** incluye la lógica intermedia de comunicación, con los archivos middleware_interface.py y middleware_tests.py.

Por otro lado, el paquete **Client** representa la capa que envía los datos para hacer la consulta, mientras que el paquete Server actúa como receptor o procesador principal.

También, tenemos el paquete **Monitor** que representa la capa que monitorea que los diferentes nodos están activos y funcionando.

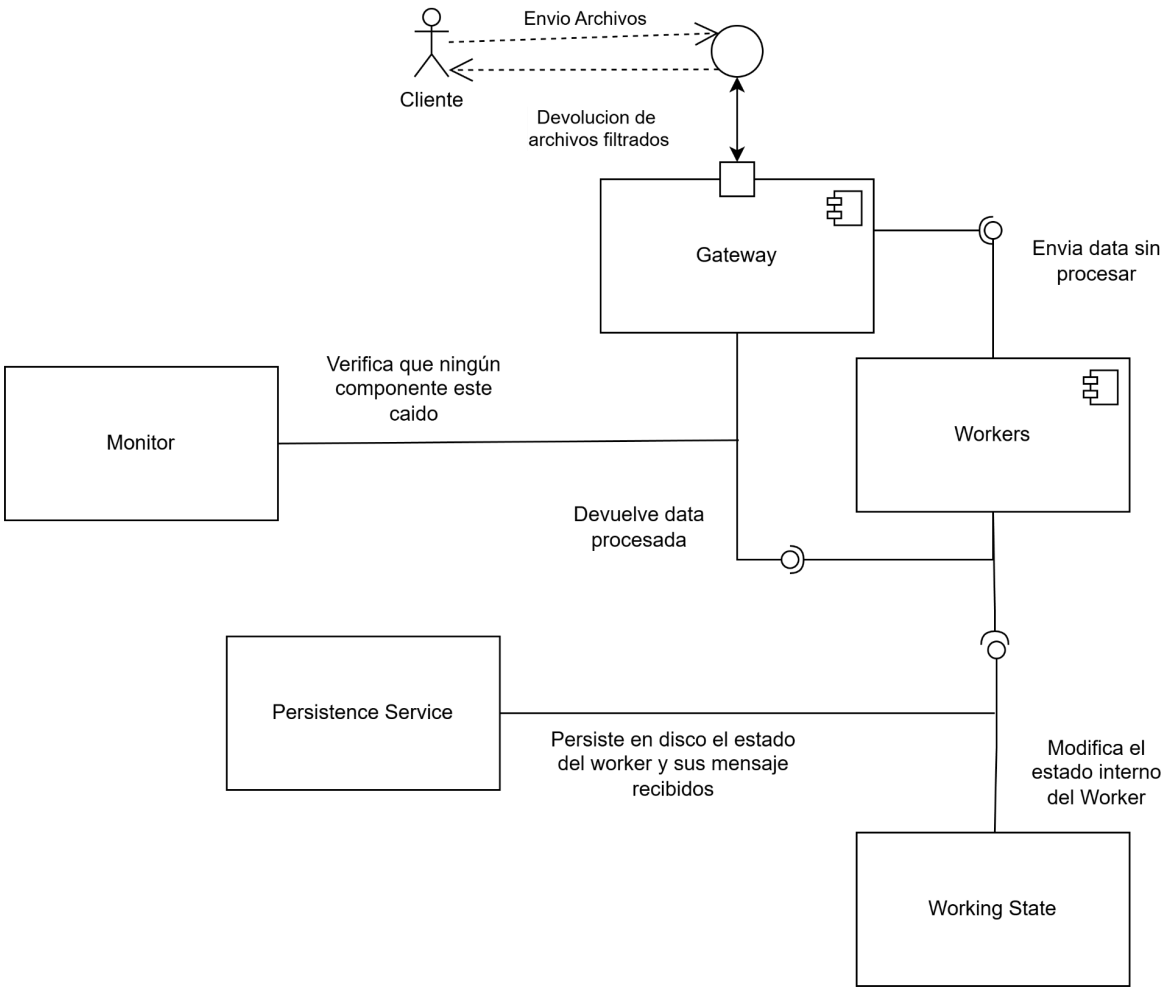
Además, tenemos paquetes más pequeños para testing, y para levantar más rápidamente el proyecto.

Finalmente, el paquete Workers agrupa diferentes tipos de tareas: Aggregators (para consolidar estadísticas), Filter (para filtrar datos innecesarios), Joiners (para unir tablas) y Maximizers (para obtener valores máximos como Top3 o Top1).

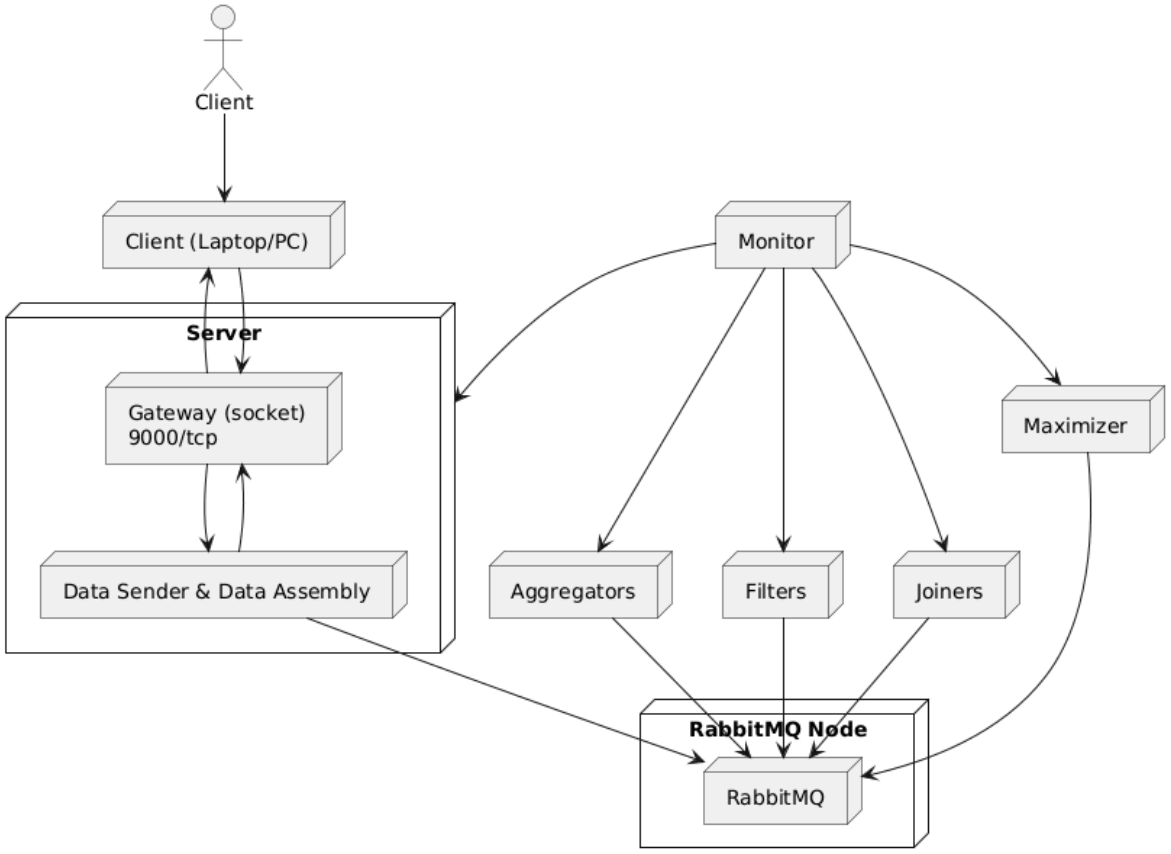
Las flechas entre los paquetes indican dependencias: cómo cada módulo utiliza funciones o clases de otros.

Componentes

Diagrama de Componentes



Vista Física



Arquitectura de Nodos

En primer lugar, el cliente se comunica con el servidor. Este recibe los datos y se encarga de formatearlos y enviarlos hacia el sistema de mensajería.
Los datos procesados se publican luego en un nodo RabbitMQ, que actúa como intermediario y distribuye los mensajes hacia distintos módulos especializados.

Un componente central denominado Monitor verifica y coordina el funcionamiento de los módulos de procesamiento —Aggregators, Filters, Joiners y Maximizers — y el servidor, que consumen los mensajes desde RabbitMQ y generan nuevas salidas. Esta estructura modular y orientada a eventos permite una alta escalabilidad y separación de responsabilidades, facilitando la extensión o modificación de cada etapa del flujo sin afectar el resto del sistema.

Workers

Filter Workers

Los Filter Workers realizan la primera etapa del procesamiento. Su función es recibir transacciones crudas y descartar aquellas que no cumplen un criterio específico, de manera que el resto del pipeline procese exclusivamente los datos relevantes. No mantienen acumuladores ni estado complejo; se limitan a filtrar y reenviar.

Existen tres tipos:

- **FilterYear**, que retiene solo transacciones entre dos años (especificados en configuración).
- **FilterHour**, que retiene transacciones en un rango horario (especificados en configuración).
- **FilterAmount**, que retiene transacciones iguales o superiores a un valor (especificado en configuración)

Cada uno actúa como un nodo “pasamanos”: cuando una fila pasa su filtro, es reenviado a las exchanges correspondientes. **FilterYear** distribuye registros hacia agregadores (para productos, compras y TPV) y hacia FilterHour; **FilterHour** reenvía hacia **FilterAmount** y también hacia el agregador de TPV; **FilterAmount** entrega directamente al servidor.

Aggregator Workers

Los Aggregators son la segunda etapa del pipeline. Su tarea es acumular valores parciales por shard, agrupando datos por claves específicas según el tipo de agregación requerida por la query.

Se implementan tres variantes:

- **AggregatorProducts**, que agrupa por (item_id, month_year) y acumula ventas y ganancias.
- **AggregatorPurchases**, que agrupa por (user_id, store_id) y acumula compras totales.
- **AggregatorTPV**, que agrupa por (store_id, year_half) y acumula el TPV parcial.

Cada uno mantiene en memoria los acumuladores que corresponden a su shard, y al finalizar la recepción de datos provenientes de todos los filtros, publica sus resultados parciales. Dichos resultados son luego combinados globalmente por los Maximizers.

Maximizer Workers

Los Maximizers recolectan los resultados parciales de todos los Aggregators y calculan los resultados globales de la query. Son los únicos nodos que ven todos los shards al mismo tiempo, por lo que pueden determinar máximos totales, rankings o sumatorias finales.

Los tipos definidos son:

- **MAX**, que obtiene las máximas ventas y ganancias por (item_id, month_year).
- **TOP3**, que calcula el top 3 de clientes por tienda.
- **TPV**, que suma el TPV total por (store_id, year_half) combinando los shards.

Cada Maximizer mantiene su estructura global: diccionarios de máximos, heaps de top 3 o acumuladores totales. Una vez que todos los shards de Aggregator enviaron su END, publica los resultados definitivos hacia su respectivo Joiner.

Joiner Workers

Los Joiners forman la etapa final del pipeline. Su propósito es agregar información a los resultados generados por los Maximizers utilizando tablas auxiliares (productos, tiendas o usuarios). Para ello reciben simultáneamente los datos provenientes del Maximizer, y la tabla auxiliar completa enviada desde el servidor.

Con ambos insumos almacenados, aplican el join: por ejemplo, agregan el nombre de un producto a un registro de ventas, o el nombre de la tienda a un registro de TPV, o la información del usuario en un resultado de top3.

Existen tres Joiners:

- **ItemsJoiner**, que utiliza MENU_ITEMS para agregar nombres de ítems;
- **StoresJoiner**, que utiliza STORES para agregar información de tiendas;
- **UsersJoiner**, que utiliza USERS para completar datos de usuarios.

Cada uno espera recibir tanto la tabla auxiliar como los resultados completos del Maximizer. Una vez que ambas fuentes están completas (incluyendo ENDs), realiza el join y publica el resultado final hacia el cliente.

Middleware de Mensajería

El sistema emplea **RabbitMQ** como bus de comunicación asíncrona. Este middleware desacopla la interacción entre los clusters, permitiendo escalabilidad y tolerancia a fallos. Los mensajes enviados incluyen instrucciones de procesamiento y resultados intermedios, lo que asegura una ejecución distribuida eficiente.

Monitor

El Monitor utiliza Exchanges de RabbitMQ para comunicarse con todos los nodos del sistema distribuido ya sea para chequeo de heartbeats, para manejo de los End Messages y comunicar el Force End. Esto se decidió por la simplicidad de la interfaz y asegurarnos que los mensajes lleguen correctamente sin miedo a que haya algún problema con la saturación de conexiones, como podría pasar si cargamos mucho las conexiones TCP, y garantizarnos el envío de mensajes.

Diagrama de Robustez