FACULTAD DE INGENIERIA
Universidad de Buenos Aires

TP Diseño - Coffee Shop Analysis

[75.74] Sistemas Distribuidos I
Segundo cuatrimestre de 2025

Avecilla, Ignacio	105067	iavecilla@fi.uba.ar
Avila, Gaston	104482	gavila@fi.uba.ar
Muñoz, Juan Martín	106699	jmmunoz@fi.uba.ar

Índice

1. Alcance

El presente informe presenta la documentación de un sistema distribuido flexible, robusto y escalable, capaz de resolver las consultas otorgadas por la catedra con una cantidad de unidades de procesamiento mayor o igual a uno.

Las consultas a resolver son:

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.

2. Arquitectura

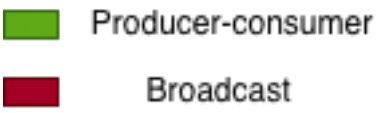
2.1. Vista Física

Se muestra la enteridad del sistema y las conexiones existentes entre las diversas entidades del sistema.

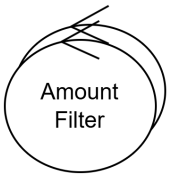
2.1.1. Diagrama de Robustez

Se visualizan todos los componentes que interactuan en nuestro diseño, desde la interacción inicial del cliente, hasta la finalización del procesado de todas las consultas.

- Los trabajadores se comunican a través de la inserción y consumo de datos en diversas colas, las cuales pueden ser accedidas por multiples trabajadores concurrentemente. Se distinguen 2 tipos de cola segun como se distribuyen los mensajes:



- Las entidades que pueden ser escaladas a multiples unidades de computo se representan de la siguiente forma:



Cada **worker** tiene asociado una cola de input y una cola de output.

Las colas marcadas como tipo "broadcast"se basan en multiples colas producer-consumer, cada una de ellas conectadas a un unico worker de salida, el worker que se encarga de producir en esta cola iterara cada una de ellas dejando el mismo dato en todas, de esa forma cada worker de salida recibira el mismo mensaje y podra procesarlo de manera independiente. Usualmente usado para joiners.

El **request handler** es el punto de entrada al sistema y es el encargado de recibir los datos del clientes y encolarlos en las colas de entrada correspondientes para que los workers puedan comenzar a procesarlos. A su vez existe un **proxy** actuando de load balancer entre los diferentes request handlers, se va a asignar un cliente a un request handler especifico y este sera el unico con el que se comunicara durante todo el envio de los datos (en el caso donde todo salga bien).

El cliente se comunicara con el request handler y recibira un *Query ID* unico para identificar su consulta, este *Query ID* sera propagado a lo largo de todo el sistema en cada mensaje para identificar a que cliente pertenece cada dato procesado. El cliente luego podra pedir los resultados de su consulta con este *Query ID*.

Todos los resultados seran insertados en diferentes colas, una para cada consulta, el **Response builder** sera el encargado de leer de ellas y mandar una unica respuesta para la consulta correspondiente

El **response builder** hara un broadcast a todos los **request handlers** con el resultado de las consultas de un cliente, de esta manera todos tiene disponible los resultados asi que no importa a quien asigne el **proxy** a la hora de pedir los resultados, siempre podra obtenerlos.

A continuación, se muestran cuatro extractos del diagrama de robustez, destacando los aspectos más relevantes de cada consulta.

Consulta 1

Es la más simple. Requiere tres filtros encadenados (**Year Filter**, **Hour Filter** y **Amount Filter**). Cada etapa puede escalarse mediante un esquema **producer–consumer**: los workers consumen mensajes de la cola, procesan la entrada y deciden si reenviarla a la cola de salida o descartarla.

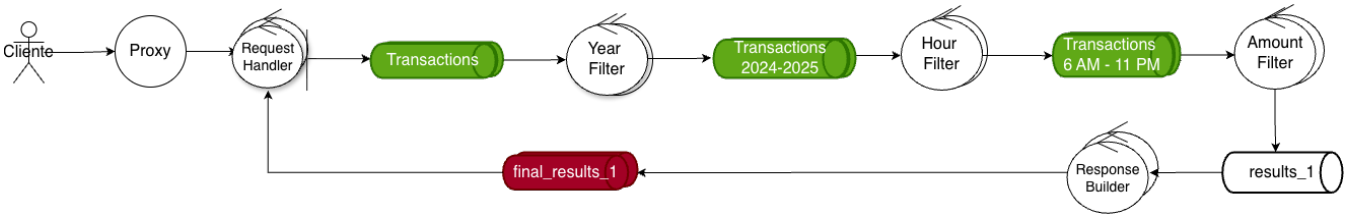


Figura 1: Diagrama de robustez - Consulta 1

Consulta 2

Ya que al final de esta consulta es necesario hacer un join de dos datasets distintos, el joiner final procesara y guardara todos los datos de `Menu Items` a la vez que procesa los datos de `Transactions profit and quantity`, para al finalizar hacer los joins correspondientes. Por este motivo, los datos de `Menu Items` se envian a todos los joiners mediante un esquema de broadcast.

1. **Menu Items**: los datos se difunden a todos los `Item Id Joiner`. Estos almacenan en memoria la lista de ítems (pocos y estáticos) y luego consumen de la cola **Items with max values** (producer–consumer) para realizar los joins. Como el total de items del menú siempre son 8 entonces podemos cargarlos en memoria sin problemas.
   Transactions Items: Estas colas obtienen los datos directo del request handler que es lo que recibe los datasets del cliente, no hay ningun pre-proceso previo asi que estos datos representan filas del dataset original.

2. **Transactions 2024-2025**: Estas colas contienen las transacciones filtradas por año, ademas cada worker elimina las columnas que no van a ser necesarias en los proximos pasos de la query.

3. **Transactions profit and quantity**: En este caso el aggregator necesita todos los datos pertenecientes a un pismo mes para poder calcular cuales son los items con mayor profit y quantity, por lo que se realiza un hash partitioning por mes, de esta forma cada worker recibe todos los datos de un mes especifico y puede calcular los resultados parciales correspondientes. Luego el aggregator se encarga de juntar todos los resultados parciales en un solo resultado final.

4. **Items with max values**: Estas colas contienen las transacciones el resultado de agrupar todas las transacciones por mes con el profit y el quantity.

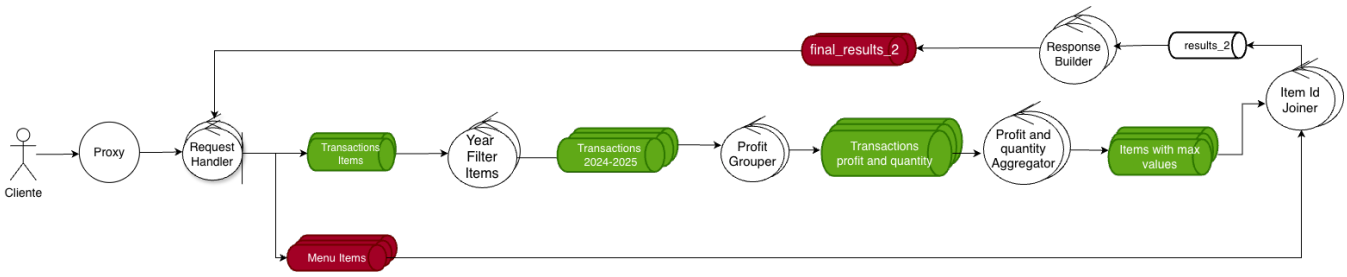


Figura 2: Diagrama de robustez - Consulta 2

Consulta 3

1. **Stores_3**: Estos datos se difunden a todos los `Store ID Joiner` y contienen todos los registros de las tiendas. Estos almacenan en memoria la lista de ítems (pocos y estáticos) y a su vez consumen de la cola **Transactions per semester** (producer–consumer) para realizar los joins. Como el total de items del menú siempre son 10 entonces podemos cargarlos en memoria sin problemas. En este caso tiene el sufijo _3 para diferenciarlo de las colas de la consulta 4.
   Transactions: Estas colas contienen las transacciones originales sin ningun pre-proceso previo asi que estos datos representan filas del dataset original.

2. **Transactions 2024-2025**: Estas colas contienen las transacciones filtradas por año, ademas cada worker elimina las columnas que no van a ser necesarias en los proximos pasos de la query.

3. **Transactions 6 AM - 11 PM**: Estas colas contienen las transacciones filtradas por hora, ademas cada worker elimina las columnas que no van a ser necesarias en los proximos pasos de la query.

4. **Transactions per semester**: Siguiendo la logica anterior tenemos un aggregator que recibe todos los datos de un semestre en particular mediante un hash partitioning por semestre, de esta forma cada worker recibe todos los datos de un semestre especifico y puede calcular el TPV correspondiente. Luego el aggregator se encarga de juntar todos los resultados parciales en un solo resultado final, el cual sera enviado a la cola de **Transactions with tpv**.

5. **Transactions with tpv**: Estas colas contienen las transacciones ya agrupadas con el TPV calculado, listas para ser unidas con los datos de las sucursales.

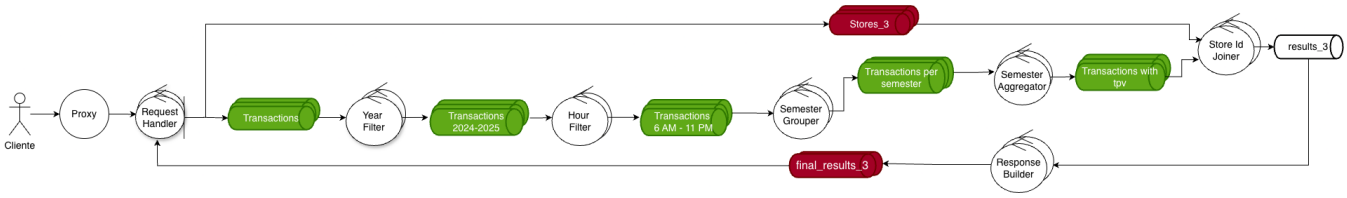


Figura 3: Diagrama de robustez - Consulta 3

Consulta 4

- **Stores_4**: Estos datos se difunden a todos los `Store ID Joiner` y contienen todos los registros de las tiendas. Estos almacenan en memoria la lista de ítems (pocos y estáticos) y luego consumen de la cola **Transactions per semester** (producer–consumer) para realizar los joins. Como el total de items del menú siempre son 10 entonces podemos cargarlos en memoria sin problemas. En este caso tiene el sufijo _4 para diferenciarlo de las colas de la consulta 3.
  Transactions: Estas colas contienen las transacciones originales sin ningun pre-proceso previo asi que estos datos representan filas del dataset original. **Users**: Estas colas contienen los datos de los usuarios de las cafeterias, los cuales son pocos y estaticos, por lo que cada worker puede cargar todos los datos en memoria y realizar los joins de manera correcta.

- **Transactions 2024-2025**: Estas colas contienen las transacciones filtradas por año, ademas cada worker elimina las columnas que no van a ser necesarias en los proximos pasos de la query.

- **Transactions 6 AM - 11 PM**: Estas colas contienen las transacciones filtradas por hora, ademas cada worker elimina las columnas que no van a ser necesarias en los proximos pasos de la query.

- **Transactions with user and store**: Al igual que en los casos anteriores, tenemos un agregator que recibe los datos de los groupers con un hash partitioning por cada store-user y los agrega en un solo resultado final, el cual sera enviado a la cola de **Top 3 transactions per store**.

- **Top 3 transactions per store**: Estas colas se comportan como si fuera un fan-out donde todos los resultados llegan a todos los workers, ya que estos resultados nunca son muy grandes (solo 3 resultados por cada sucursal) y al tenerlo en memoria, los workers pueden realizar los joins de manera correcta.

- **Top 3 transactions with birthday**: Estas colas contienen los resultados finales de la consulta, con los 3 clientes que más compras han hecho en cada sucursal, junto con su fecha de cumpleaños.

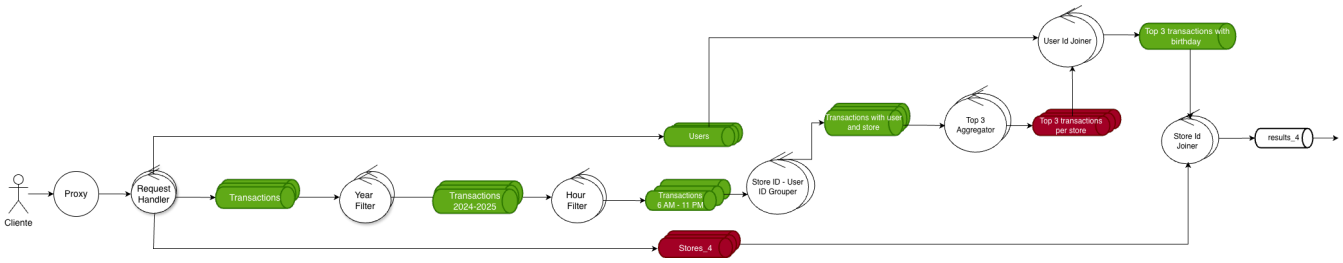


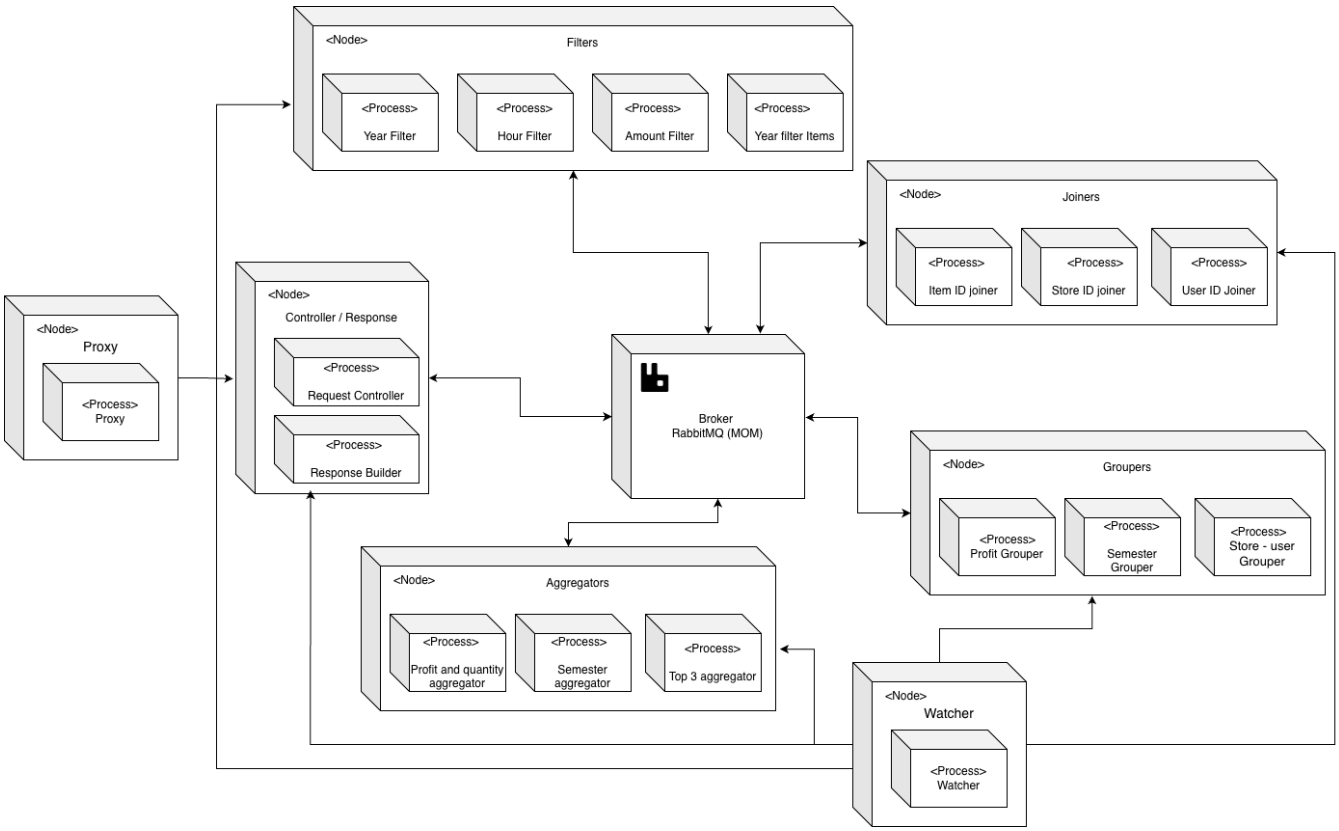
Figura 4: Diagrama de robustez - Consulta 4

2.1.2. Diagrama de Despliegue

Podemos ver como los toda comunicación interna es realizada mediante RabbitMQ (el middleware). En este caso se agrupan los workers por tipo en un mismo nodo de computo.

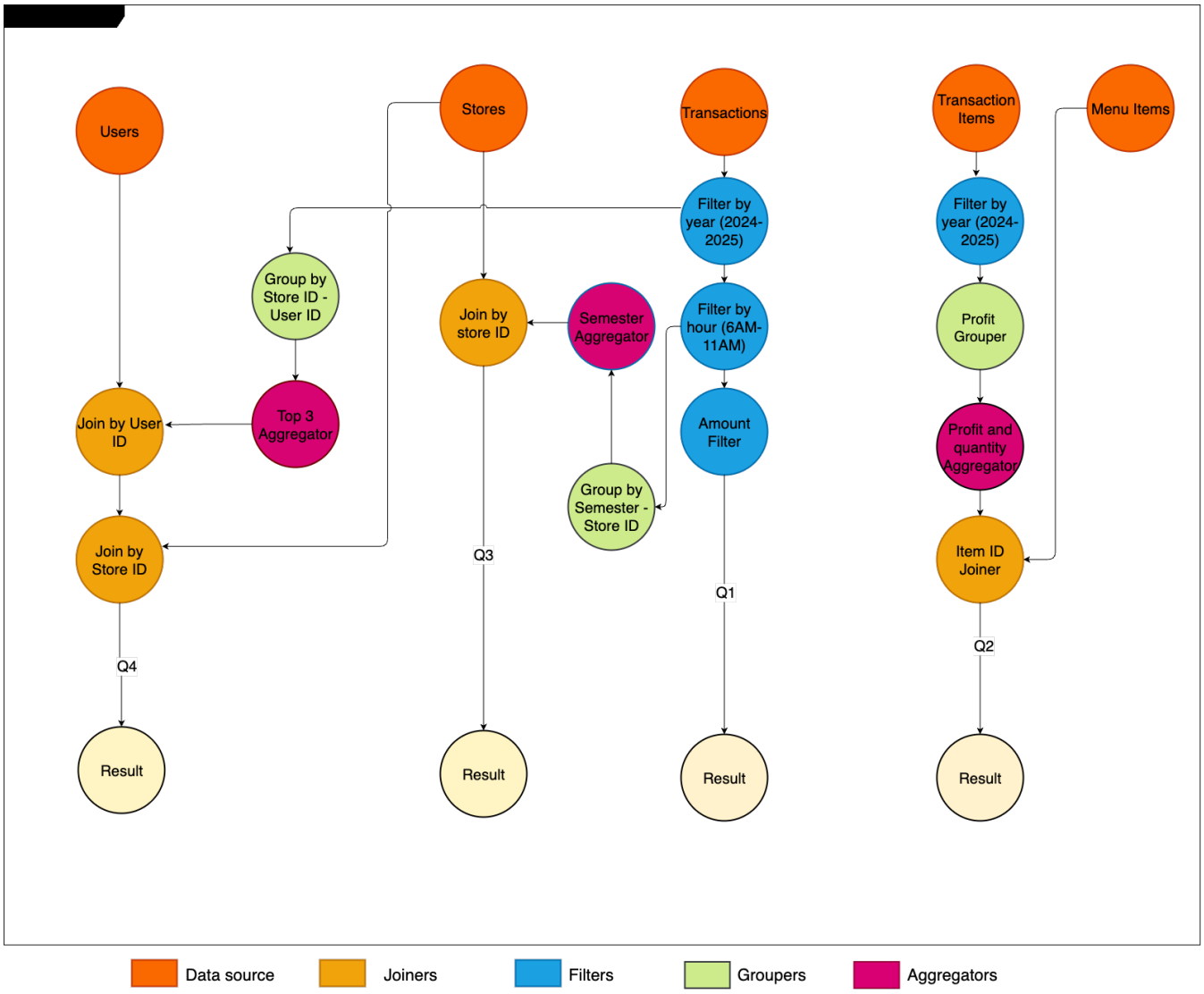
Tambien podemos ver el proxy mencionado anteriormente que se encarga de distribuir la carga entre los distintos request handlers.

Tambien un nuevo componente llamado **Watcher** que se encarga de monitorear el estado de los distintos nodos del sistema y en caso de detectar una falla reiniciar el nodo caido para asegurar la disponibilidad del sistema.



2.2. Vista Lógica

2.2.1. DAG



Se visualiza un directed acyclic graph que muestra el flujo de datos siendo atravezado por los distintos componentes, comenzando de arriba hacia abajo. Los trabajadores (workers) se dividen en agrupadores (groupers), acumuladores (joiners), agregadores (aggregators) y filtradores (filters). Las fuentes de información inicial, y donde se almacena finalmente lo procesado son a nivel de implementación colas. Los aggregators son los nodos encargados de obtener data de los groupers y acumularla en un solo resultado final segun sea necesario, de esta manera los groupers pueden recibir cualquier tipo de data y agrupar en base a los datos que le lleguen.

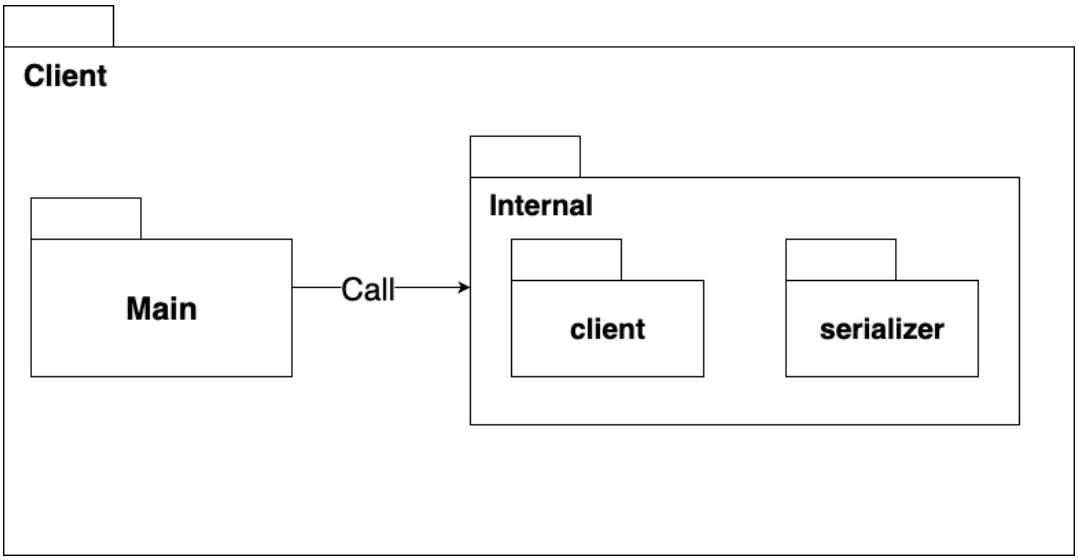
2.3. Vista de Desarrollo

Aquí podemos visualizar como esta planeada la arquitectura del sistema desde la perspectiva del código. Se divide el sistema en distintos modulos para Client, Worker, Request Handler, Response builder, Middleware, Healthcheck, Protocol, proxy y watcher.

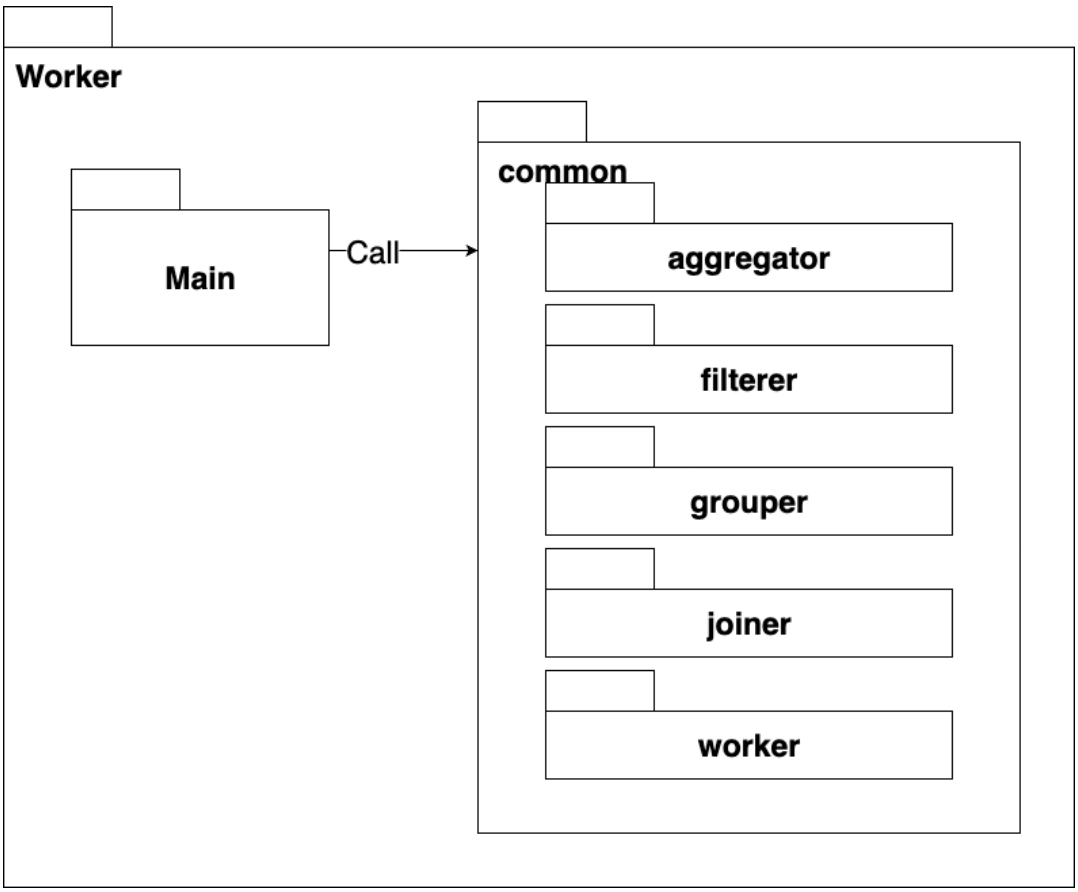
2.3.1. Diagrama de Paquetes

Se muestran los distintos modulos implementados:

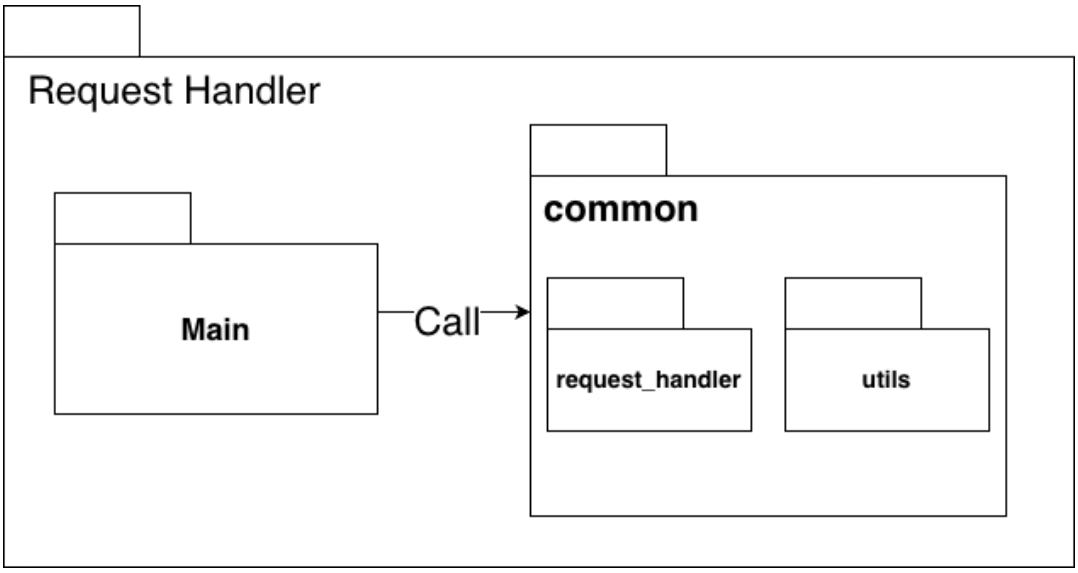
El cliente tiene la responsabilidad de comunicarse con el Request handler para enviarle toda la data necesaria para procesar las 4 queries. Tiene como entrypoint un modulo main que se encarga de parsear la configuracion y comenzar la comunicacion llamando al modulo client. El serializer es un modulo con algunas funciones auxiliares que sirven para serializar y deserializar las filas de los datasets en mensajes que puedan ser enviados a traves de TCP.



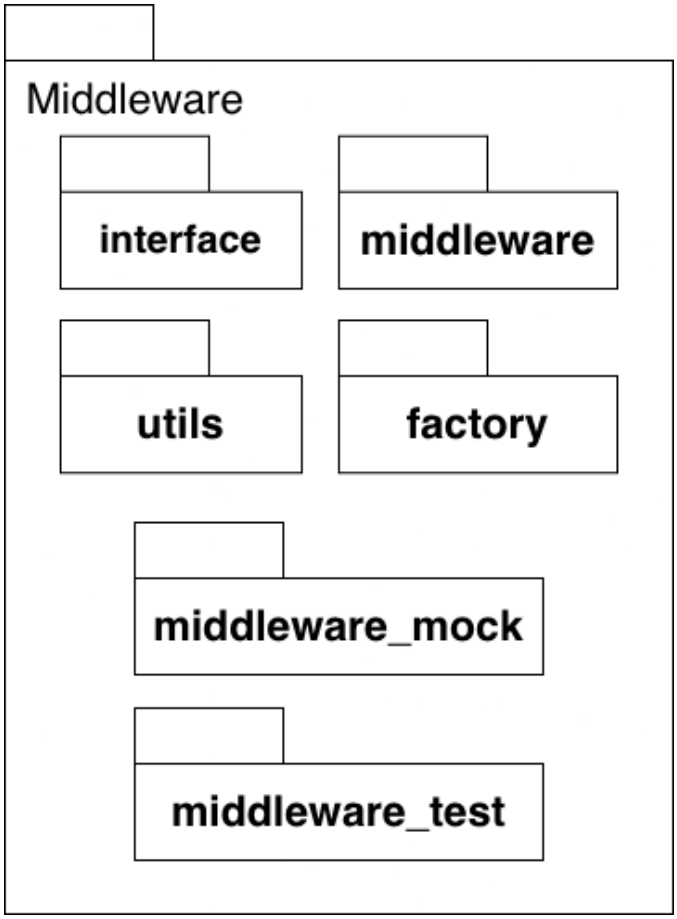
Los workers tienen la tarea de filtrar, acumular o agrupar los diversos tipos de datos que lean de las colas a las que estes subscriptos. En este caso todos los tipos de workers comparten el mismo modulo main donde se parsea la configuracion y se inicializa el worker llamando al modulo worker, dicho modulo tiene la logica comun a todos los tipos de workers, como la conexion a RabbitMQ y el manejo de colas segun esten configuradas. Este modulo luego llamara a los modulos correspondientes segun el tipo de worker que sea, leyendo una variable de ambiente, donde ya se implemenara la lógica especifica para cada uno de ellos. Para simplificacion del gráfico no se muestran todos los tipos de workers pero existe un modulo distinto para cada uno de ellos dentro de su respectivo paquete.



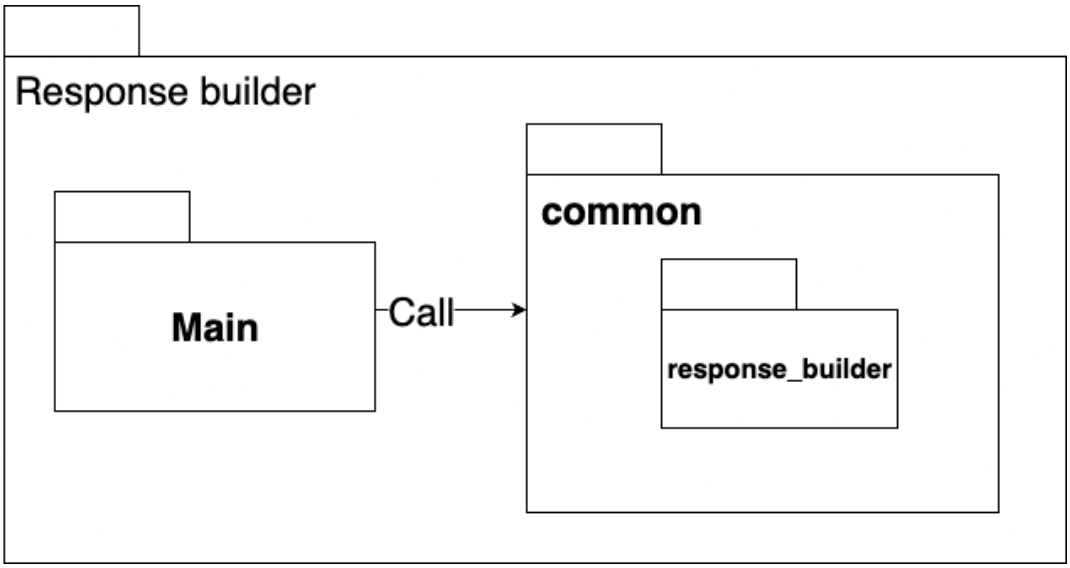
La tarea del request handler es interpretar los mensajes del cliente, para poder derivarlos a las diferentes colas de RabbitMQ. Tiene un modulo main que se encarga de parsear la configuracion y comenzar la comunicacion llamando al modulo request handler, el cual tiene la logica para interpretar los mensajes del cliente y enviarlos a las colas correspondientes para empezar el trabajo. Tambien cuenta con un modulo de utils donde tiene algunas funcionalidades extra relacionadas a la tolerancia a fallos del sistema, seran explicadas en su respectiva seccion.



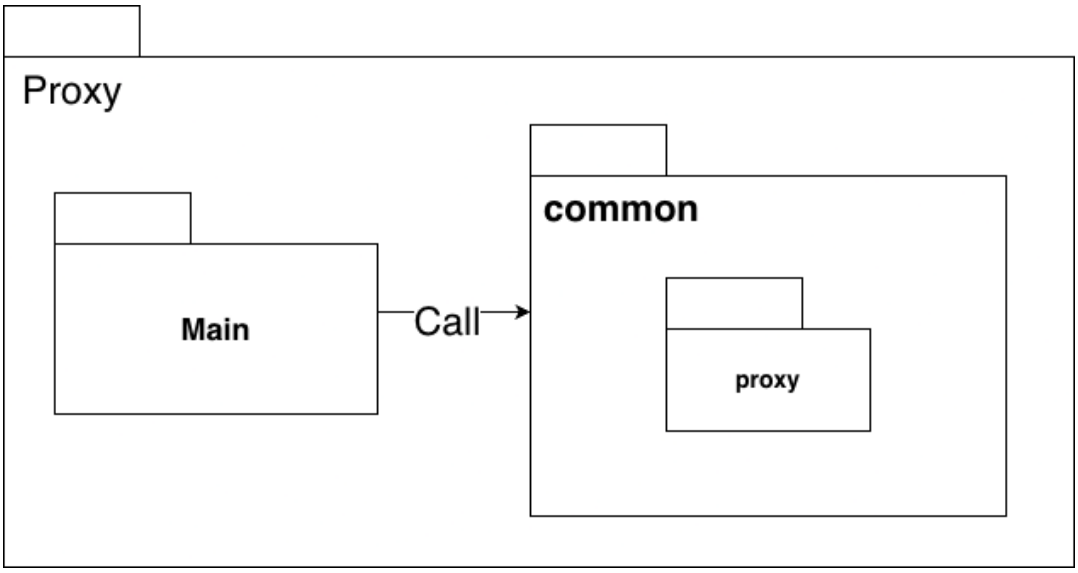
El Middleware es el que se encarga principalmente de la interaccion con la libreria de RabbitMQ. Contiene un modulo con la interfaz (provista a traves del enunciado). La implementacion de dicha interfaz se encuentra en el modulo middleware, el cual tiene la logica para conectarse a RabbitMQ, encolar y desencolar mensajes. El modulo de utils contiene funciones auxiliares utilizadas por el modulo middleware. Los tres modules restantes se orientan a la utilizacion del middleware en distintos tests de integracion. El modulo factory contiene una interfaz para poder crear middlewares de test y middlewares reales, los middleware de tests no interaccionan con RabbitMQ y abstraer esa parte para hacer mas facil la escritura de tests, dicha implementacion esta en middleware_mock. El modulo de middleware_test contiene algunas funciones extra que acompañan al middleware usado en los tests.



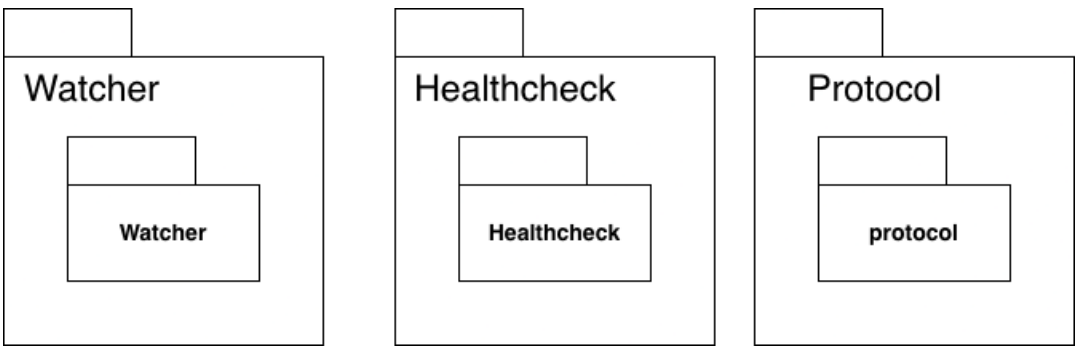
El response builder tiene la responsabilidad de procesar los resultados parciales de cada query que van llegando de los ultimos workers y armar la respuesta final para enviarsela al request handler, quien luego se encargara de enviarla al cliente. Tiene un modulo main que se encarga de parsear la configuracion y comenzar la comunicacion llamando al modulo response builder, el cual tiene la logica para leer los resultados parciales de las colas correspondientes y armar la respuesta final.



El proxy tiene la responsabilidad de balancear la carga entre los distintos request handlers disponibles. Tiene un modulo main que se encarga de parsear la configuracion y comenzar la comunicacion llamando al modulo proxy, el cual tiene la logica para recibir las conexiones de los clientes y derivarlas a los request handlers disponibles.



El watcher tiene la responsabilidad de monitorear el estado de los distintos nodos del sistema y en caso de detectar una falla reiniciar el nodo caido para asegurar la disponibilidad del sistema. Tiene un unico module con un script que mediante un Healthcheck periodico verifica el estado de los nodos y en caso de detectar una falla ejecuta un script de reinicio del nodo caido. El healthcheck es un modulo aparte que contiene la logica para crear un servicio de healthcheck que pueda ser utilizado por los distintos nodos del sistema para reportar su estado hacia el watcher. El Protocol es un modulo que contiene la definicion de los mensajes que se envian entre el cliente y el request handler, tanto los tipos de mensaje como su estructura, formas de parsearlos y serializarlos.



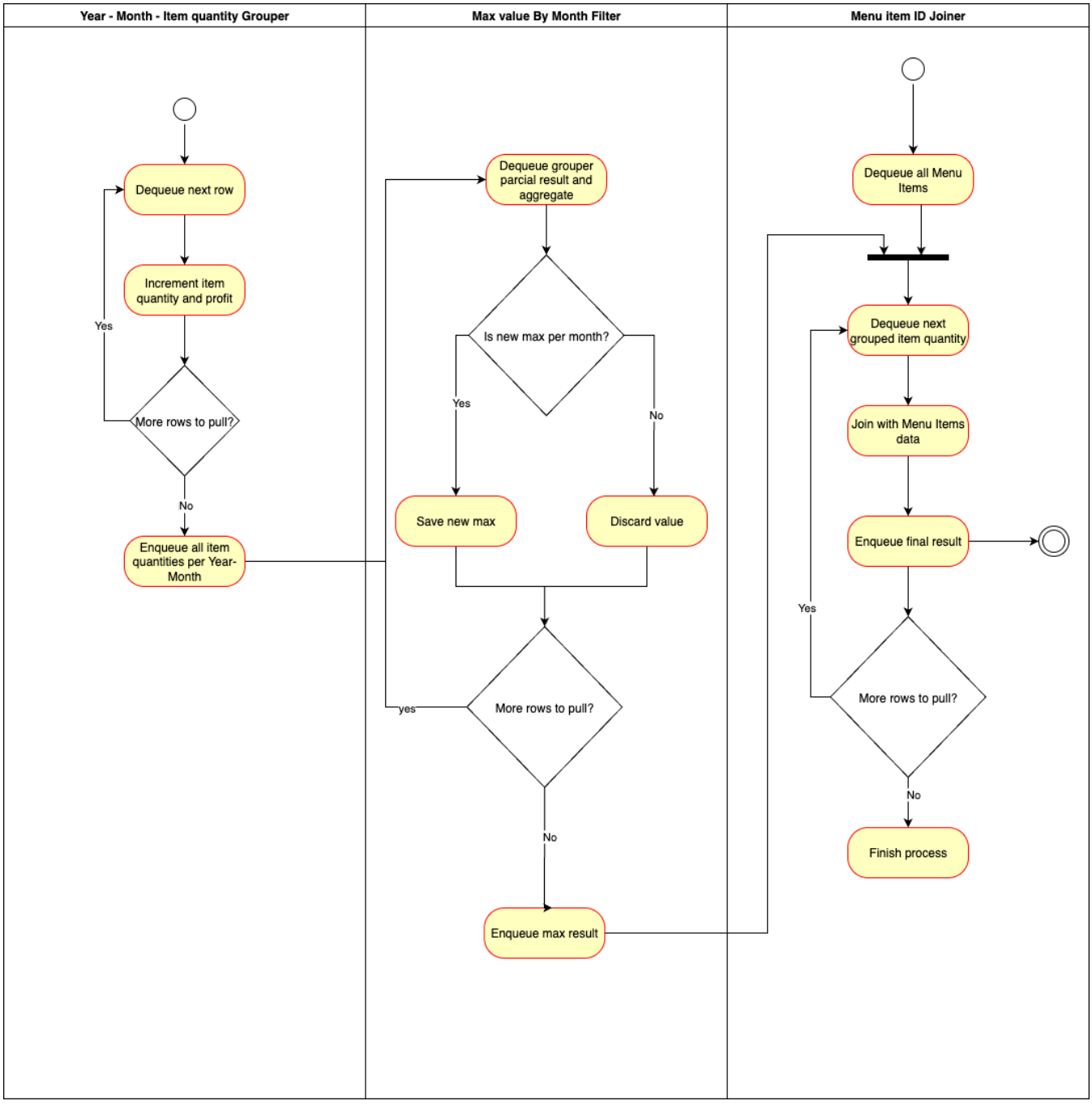
2.4. Vista de Procesos

Se representa la interacción entre los componentes del sistema, su forma de comunicación de principio a fin. Es posible visualizar la concurrencia del sistema, asi como también la escalabilidad y distribuición de tareas.

2.4.1. Diagrama de Actividad

En el siguiente diagrama se muestra parcialmente el proceso de obtener los productos más vendidos y que más ganancias han generado, para cada mes, en este caso se representa solamente la parte de mayor complejidad de obtención, la de mayor valor y no solo mayor cantidad de ventas.

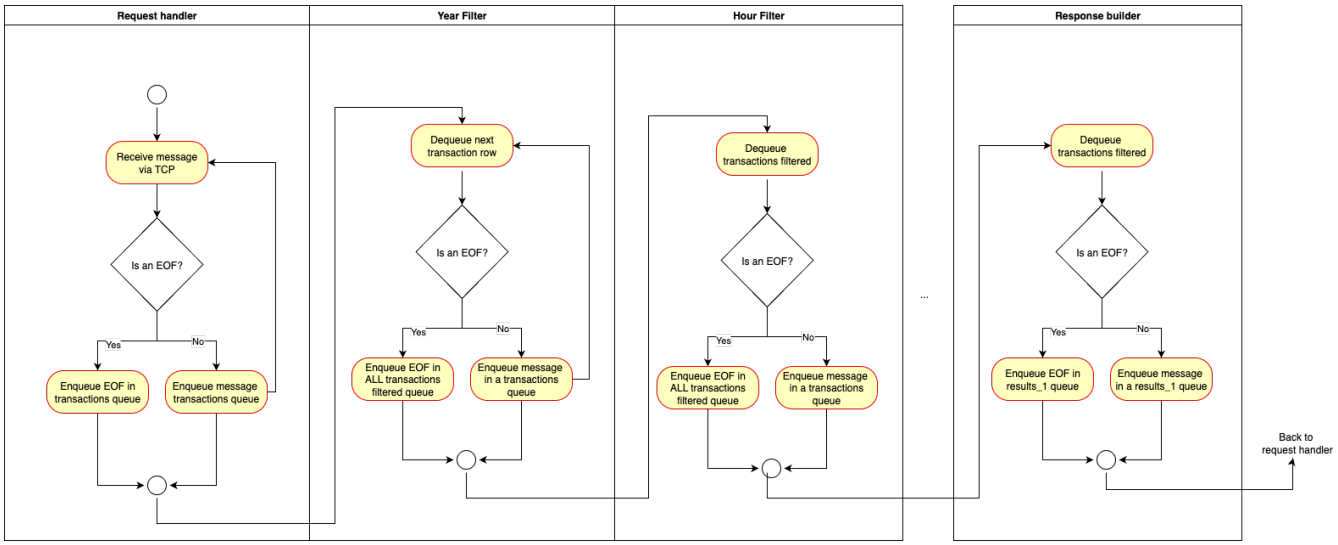
*Se considera más complejo debido a que los pasos necesarios para obtener este dato es superior a la de calcular el producto más vendido en cantidad.



El proceso que observamos en el diagrama comienza con los groupers, en este caso estos tomaran la data de diferentes colas de entrada donde tienen transacciones e iran sumando los valores correspondiente a un item del menu en un mismo año y mes, para luego enviar sus resultados a la cola de salida para que el aggregator los pueda procesar El aggregator se encargara de recibir todos los resultados parciales de los groupers y sumarlos en un solo resultado final para cada año-mes. Este sera enviado a la cola de salida para que el joiner del menu de items pueda agregar el nombre del item a los resultados El joiner del menu de items en un principio espera a obtener los datos del dataset de menu items ya que sin estos no podra realizar el join, una vez los tiene se queda esperando por el resultado

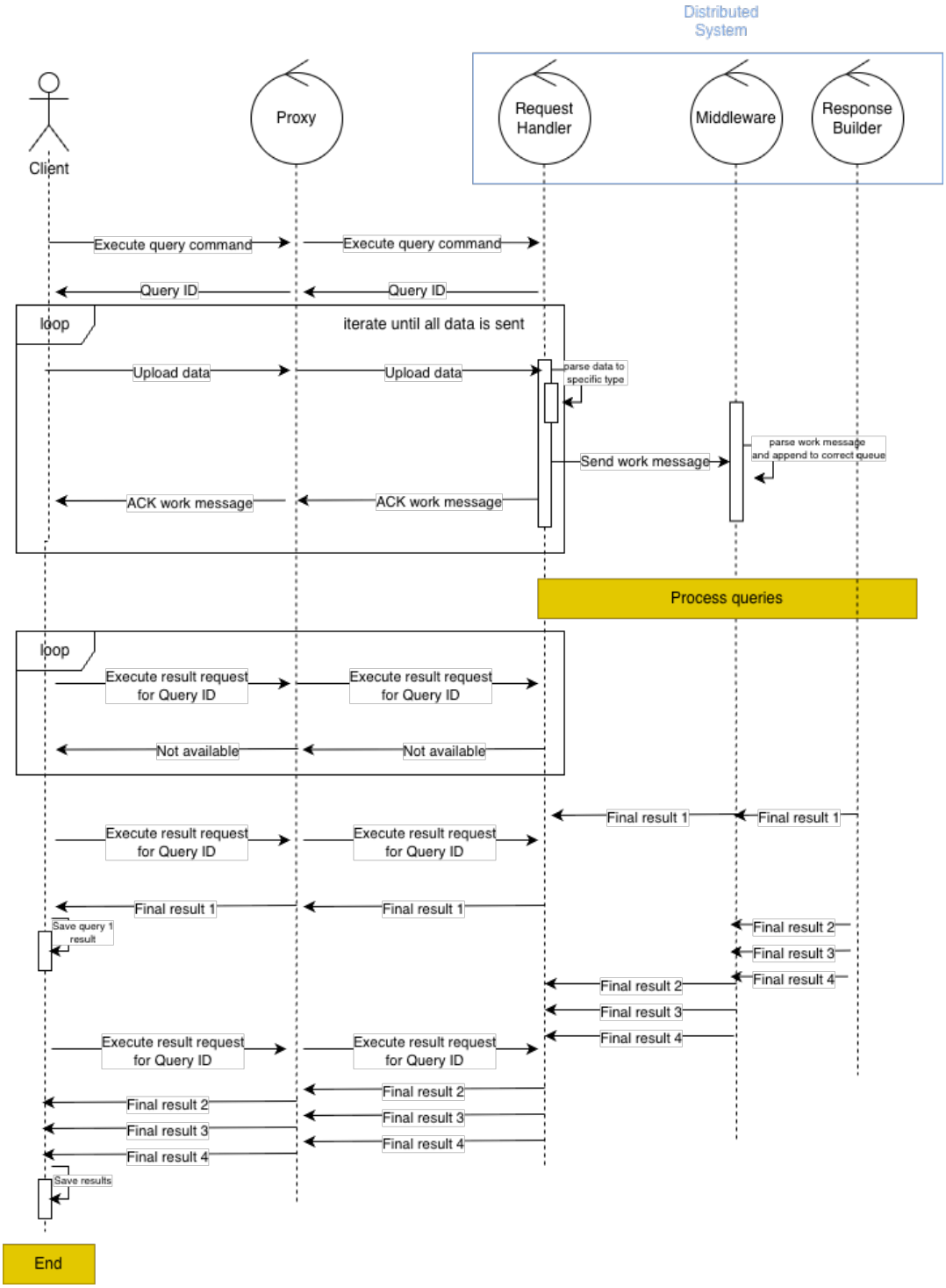
del aggregator. Une vez recibidos los resultados finales del aggregator y realizar el join con los datos correctos envia dicha fila de resultado a la cola de resultados que va a ser leida por el response builder

Manejo de EOF: En el siguiente diagrama se muestra como se maneja el EOF desde que comienza una query hasta que termina para determinar cuando los workers dejan de esperar por datos y dejan de leer una cola. Para simplificar el diagrama se muestran solo algunos filtros del total ya que el proceso es el mismo para todos.



Este proceso se da como en una especie de cascada. En primer lugar el cliente enviara datos indicando que tipo de dataset esta enviando (transactions, menu_items, etc.) mandando un EOF cuando ya envio todos los datos relacionados a un tipo especifico. El request handler detectara este mensaje de EOF y encolara un nuevo mensaje EOF en las colas de salida correspondientes. Asi se dara el proceso donde cada vez que un worker lee un mensaje de EOF debe propagarlo hacia la cola de salida sea cual sea, asi se asegura de que el proximo worker tambien lo recibe. Una vez llega al ultimo worker de la query este encola el resultado final indicando tambien que ha terminado de procesar todos los datos. El response builder espera a recibir todos los EOFs de todos los workers de salida para saber que ya tiene la totalidad de la respuesta para la query entonces puede procesarla y enviarla al request handler.

2.4.2. Diagrama de Secuencia



En este caso vemos el diagrama de secuencia completo desde que el cliente comienza a enviar los datos hasta que recibe la respuesta final. Podemos observar como el cliente se comunica con el proxy (y luego al request handler) para enviarle los datos, luego el request handler encola los datos en las colas correspondientes, mediante el middleware, para que los workers puedan comenzar a procesarlos. Los workers leen de las colas, procesan los datos y encolan los resultados en las colas de salida correspondientes. Finalmente, el response builder lee los resultados de las colas de salida y arma la respuesta final para enviarsela al request handler, quien luego se la envia al cliente. El cliente al inicio d ela conexion recibe lo que se denomina un Query ID unico para su consulta, este sera propagado a lo largo de todo el sistema en cada mensaje para identificar a que cliente pertenece cada dato procesado. El cliente luego podra pedir los resultados de su consulta con este Query ID. A medida que esten los resultados de las distintas queries disponibles, el request handler ira enviandolos al cliente cuando este los solicite.

3. Comunicacion

3.1. Vista general

Se describe el protocolo de comunicación utilizado entre el client y request_handler para la transferencia de archivos CSV y la obtención de resultados. El protocolo utiliza framing de tipo TLV (type-lenght-value) para prevenir short reads/writes.

3.2. Sistema de Tipos de Archivo

El protocolo utiliza un sistema de enumeración (`enum`) para los tipos de archivo, con el fin de asegurar que tanto el cliente como el servidor estén de acuerdo en la categorización de los datos:

```
type FileType int

const (
FileTypeTransactions        FileType = 0  // Datos de transacciones
FileTypeTransactionItems    FileType = 1  // Detalles de los items de transaccion
FileTypeStores              FileType = 2  // Informacion de tiendas
FileTypeMenuItems           FileType = 3  // Catalogo de items del menu
FileTypeUsers               FileType = 4  // Informacion de usuarios
)
```

Estructura de Directorios

El cliente espera la siguiente estructura de directorios bajo el directorio `/data`:

```
/data/
|-- transactions/         -> FileType 0 (FileTypeTransactions)
|   |-- file1.csv
|   +-- file2.csv
|-- transaction_items/    -> FileType 1 (FileTypeTransactionItems)
|   |-- file1.csv
|   +-- file2.csv
|-- stores/               -> FileType 2 (FileTypeStores)
|   |-- file1.csv
|   +-- file2.csv
|-- menu_items/           -> FileType 3 (FileTypeMenuItems)
|   |-- file1.csv
|   +-- file2.csv
+-- users/                -> FileType 4 (FileTypeUsers)
    |-- file1.csv
    +-- file2.csv
```

3.3. Tipos de Mensaje

Todos los mensajes comienzan con un byte identificador del tipo de mensaje:

Tipo de Mensaje	Valor	Dirección	Descripción
MessageTypeBatch	0x01	Cliente → Servidor	Lote de datos CSV
MessageTypeEOF	0x02	Cliente → Servidor	Fin de un tipo de archivo
MessageTypeFinalEOF	0x03	Cliente → Servidor	Todos los datos transferidos
MessageTypeACK	0x04	Servidor → Cliente	Confirmación de recepción (ACK)
MessageTypeResultChunk	0x05	Servidor → Cliente	Fragmento de datos de resultado
MessageTypeResultEOF	0x06	Servidor → Cliente	Fin del resultado
MessageTypeQueryId	0x07	Servidor → Cliente	Identificador de consulta
MessageTypeQueryRequest	0x08	Cliente → Servidor	Solicitud de inicio de query
MessageTypeResultsRequest	0x09	Cliente → Servidor	Solicitud de resultados
MessageTypeResultsPending	0x0A	Servidor → Cliente	Resultados en proceso
MessageTypeResultsReady	0x0B	Servidor → Cliente	Resultados listos
MessageTypeResumeRequest	0x0C	Cliente → Servidor	Solicitud de reanudación de envio de data
MessageTypeHealthCheck	0x0D	Watcher → Nodo	Mensaje de verificación de estado health check

Muchos de estos mensajes fueron ya mencionados en las secciones previas, pero se agregan algunos nuevos para la implementación de múltiples clientes y tolerancia a fallos.

3.4. Formatos de Mensaje

3.4.1. Batch Message (0x01)

Transfiere un lote de datos CSV con metadatos.
Formato:

```
+----------+----------+------------+-------------+-------------+-------------+----------+-------------+
| Tipo     | MsgID    | ClientID   | TipoArchivo | ChunkActual | TotalChunks | NumFilas |   Payload   |
| (1 byte) | (4 bytes)| (8 bytes)  | (1 byte)    | (4 bytes)   | (4 bytes)   | (4 bytes)|  Variable.  |
+----------+----------+------------+-------------+-------------+-------------+----------+-------------+
```

Campos:

- **Tipo**: 0x01
- **MsgID**: Identificador único del mensaje para seguimiento y deteccion de duplicados. — *uint32 Big Endian*
- **ClientID**: Identificador del cliente que envía el mensaje — 8 bytes
- **Tamaño de Marco (Frame Size)**: Tamaño total del marco (excluyendo el byte de tipo) — *uint32 Big Endian*
- **TipoArchivo**: Valor del tipo de archivo (0-4) — 1 byte
- **ChunkActual**: Número del fragmento actual (indexado desde 1) — *uint32 Big Endian*
- **TotalChunks**: Número total de fragmentos — *uint32 Big Endian*
- **NumFilas**: Número de filas CSV en este fragmento — *uint32 Big Endian*
- **Payload**: Filas CSV separadas por saltos de línea — longitud variable

3.4.2. Mensaje EOF (0x02)

Indica la finalización de todos los archivos para un tipo de archivo específico.
Campos:

- **Tipo**: 0x02
- **TipoArchivo**: Valor del tipo de archivo (0-4)

3.4.3. Mensaje Final EOF (0x03)

Indica que la transferencia de todos los datos ha finalizado para todos los tipos de archivo.

3.4.4. Mensaje ACK (0x04)

Confirma la recepción exitosa de un mensaje.

3.4.5. Mensaje de resultado parcial (0x05)

Transfiere un fragmento de datos de resultado desde el servidor hacia el cliente.
Campos:

- **Tipo**: 0x05
- **QueueID**: Identificador de cola de resultados (1–4) — *uint32 Big Endian*
- **ChunkActual**: Número de fragmento actual — *uint32 Big Endian*
- **TotalChunks**: Número total de fragmentos para este resultado — *uint32 Big Endian*
- **DataLen**: Longitud de los datos — *uint32 Big Endian*
- **Data**: Datos binarios del resultado — longitud variable

3.4.6. Mensaje Result EOF (0x06)

Indica el final de los resultados provenientes de una cola específica.
Campos:

- **Tipo**: 0x06
- **QueueID**: Identificador de cola de resultados — *uint32 Big Endian*

3.4.7. Mensaje Query ID (0x07)

Proporciona al cliente un identificador único para su consulta. **Campos:**

- **Tipo**: 0x07
- **QueryID**: Identificador único de la consulta — *uint32 Big Endian*

3.4.8. Mensaje Query Request (0x08)

Solicita el inicio del procesamiento de una consulta. **Campos:**

- **Tipo**: 0x08

3.4.9. Mensaje Results Request (0x09)

Solicita los resultados de una consulta específica. **Campos:**

- **Tipo**: 0x09
- **QueryID**: Identificador único de la consulta — *uint32 Big Endian*

3.4.10. Mensaje Results Pending (0x0A)

Informa al cliente que los resultados de su consulta aún están en proceso. **Campos:**

- **Tipo**: 0x0A

3.4.11. Mensaje Results Ready (0x0B)

Informa al cliente que los resultados de su consulta están listos. **Campos:**

- **Tipo**: 0x0B

3.4.12. Mensaje Resume Request (0x0C)

Solicita la reanudación del envío de datos desde el último mensaje ACK recibido. **Campos:**

- **Tipo**: 0x0C
- **QueryID**: Identificador único de la consulta — *uint32 Big Endian*

3.4.13. Mensaje Health Check (0x0D)

Utilizado por el Watcher para verificar el estado de los nodos del sistema. **Campos:**

- **Tipo**: 0x0D

3.5. Implementación multicliente

Para la implementación de soporte de multiples clientes en simultaneo, se hicieron modificaciones a los mensajes que envia el cliente al Request Handler. Todo mensaje de datos enviado (es decir, contenido de un archivo csv), contiene un header delimitado por un caracter newline. Este header de una sola columna informa el ID de cliente que envia el mensaje generado por el request handler de manera aleatoria al inicio de la conexión.

Luego, la comunicación interna del sistema propaga este header de principio a fin, de esta forma, todo tipo de worker al recibir un mensaje analiza el cliente de origen para el mensaje, y lo almacena o procesa de forma acorde, separando el procesamiento de los distintos clientes.

Finalmente, el response_builder interpreta también los resultados específicos de cada cliente y los deriva a la cola de resultados correspondiente, el request_handler, los consume, y mediante este identificador unico que contiene el mensaje del resultado, sabe que resultados tiene disponible. En cada consulta, el cliente debe solicitar los resultados mediante su Query ID unico, y el request handler le enviara los resultados correspondientes a ese cliente.

4. Tolerancia a Fallos

4.1. Caidas de diferentes nodos

Existen dos nuevas funcionalidades principalmente que ayudan a que el sistema siga en funcionamiento incluso si los nodos fallan aleatoriamente.

4.1.1. Persistencia de datos

Todos los workers se encargan de procesar los datos y enviarlos a las colas correspondientes, una vez que el dato es proceso se realiza el ACK hacia Rabbit dando a entender que el mensaje fue correctamente procesado y puede ser eliminado de la cola. En caso de que un worker falle antes de enviar el ACK, RabbitMQ se encargara de reenviar el mensaje a otro worker disponible para que este pueda procesarlo, asegurando que ningun dato se pierda en caso de fallos.

A su vez, todos los workers del sistema y el response builder guardaran en un archivo, la cantidad de EOF que recibieron por cada cliente y por cada tipo de archivo. De esta forma, en caso de que un worker falle y deba reiniciarse, este podra leer este archivo para saber cuantos EOFs ya habia procesado y asi evitar procesar datos que ya habia procesado anteriormente o trabar la ejecucion esperando por EOFs que ya habia recibido.

En el caso especifico de los joiners, groupers y aggregators que requieren la totalidad de los datos para dar el resultado final, estos guardaran en un par de archivos adicionales los mensajes que lleguen del cliente. Una vez que reciben la totalidad de EOFs necesarios para avanzar con el procesamiento, estos podran leer los archivos del disco para obtener toda la data junta y poder procesarla correctamente. Una vez que es procesada esa data es eliminada. En caso de que cualquiera de estos nodos sufra una caida, al reiniciarse podra leer los archivos del disco para obtener toda la data que habia recibido previamente y asi continuar con el procesamiento leyendo los datos restantes directamente de las colas.

4.1.2. Watcher

El watcher es un componente adicional que se encarga de monitorear el estado de los distintos nodos del sistema y en caso de detectar una falla reiniciar el nodo caido para asegurar la disponibilidad de estos.

Todos los nodos del sistema (request handlers, response builders y workers) cuentan con un servicio de healthcheck implementado mediante un servidor HTTP que responde a las peticiones del watcher. El watcher se encarga de enviar periodicamente peticiones de healthcheck a todos los nodos del sistema y esperar su respuesta. En caso de que un nodo no responda en un tiempo determinado o directamente responda un error, el watcher asumira que el nodo ha fallado y ejecutara un script de reinicio del nodo caido para intentar recuperarlo. El tiempo es configurable.

4.1.3. Reanudación de envíos desde el cliente

El cliente se va a conectar a un request handler que va a ser proporcionado por el proxy. En caso de que el request handler falle en el medio del envio de los datos, el cliente va a detectar que la conexión se ha caído y va a intentar reconectarse al proxy para obtener un nuevo request handler disponible. Una vez conectado al nuevo request handler, el cliente enviará un mensaje de reanudación de envío (Resume Request) con el Query ID correspondiente. El request handler responderá con un ACK y el cliente comenzará a enviar los datos desde el último mensaje ACK recibido por el request handler anterior, asegurando que no se pierdan datos en caso de fallos y a la vez evitando el reenvío de datos ya enviados previamente.

El request handler no mantiene un estado en disco ya que lo unico que necesita saber es de que cliente esta recibiendo la data para poder mandar el mensaje a las correspondientes colas.

En caso de que el proxy falle, el cliente reintentara conectarse a la red durante un determinado tiempo, en caso de que se pueda conectar nuevamente, este continuara con el envio de datos normalmente. En caso de que no pueda conectarse en el tiempo determinado, el cliente abortara la conexion y finalizara la ejecucion.

En caso de que el cliente se caiga durante el envio de datos, este simplemente debera reiniciarse y volver a enviar los datos desde el principio, ya que el request handler y los workers se encargaran de manejar los duplicados y no procesar datos que ya hayan sido procesados previamente. En caso de que haya enviado la data completa y se haya caido, podra simplemente volver a conectarse y solicitar los resultados de su consulta mediante el Query ID unico que le fue asignado al inicio de la conexion.

El request handler guardara todos los resultados recibidos desde el response builder en disco, de esta forma en caso de que este falle, al reiniciarse podra leer los resultados ya procesados y evitar tener que pedirle al response builder que le envie nuevamente todos los resultados desde el principio. Luego de un tiempo determinado, el request handler eliminar los resultados que tiene en el disco, se hayan o no enviado al correspondiente cliente. Una vez que lo haga, enviara un mensaje especial de ÇLEANUP"que se propagara por todos los workers, esto evita que queden datos huérfanos en el sistema que nunca seran procesados y que se acumulen resultados viejos en disco.

4.2. Mensajes duplicados

Todos los mensajes seran procesados con el par (queryId, MsgID) este par nunca debera repetirse en caso exitoso. A la hora de procesar un mensaje se chequea que este par no haya sido procesado y en caso contrario se descarta completamente evitando el proceso de mensajes duplicados.