Universidad de Buenos Aires

Facultad de Ingeniería

Trabajo Práctico Diseño: Tolerancia a fallos

Sistemas Distribuidos I (TA050)

2° Cuatrimestre 2025

Alumno	**Padrón**	**E-mail**
Federico Genaro	109447	fgenaro@fi.uba.ar
Santiago Sevitz	107520	ssevitz@fi.uba.ar
Máximo Utrera	109651	mutrera@fi.uba.ar

Fecha de Presentación: 16 de Septiembre de 2025

Índice

1 Introducción

Este trabajo propone el diseño de un sistema distribuido para analizar datos de ventas de una cadena de cafeterías en Malasia. Se busca obtener información clave sobre transacciones, clientes y productos, priorizando la escalabilidad y robustez del sistema mediante conceptos y buenas prácticas de sistemas distribuidos.

2 Escenarios y necesidades del cliente

Para comenzar se hace un análisis de los requisitos funcionales que el sistema debe cumplir para ser de utilidad al cliente. Para esto se escriben los posibles casos de uso del mismo, y se modelan en un simple diagrama auto-explicativo.

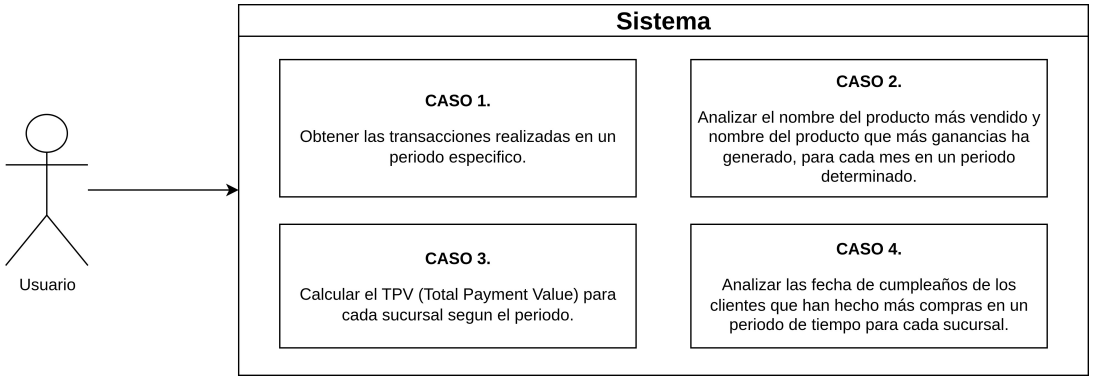


Figura 1: Actividades generales que el cliente debe ser capaz de realizar.

3 Análisis de los conjuntos de datos a procesar

Previo a detallar las capas de procesamiento per-se, se hace un análisis del conjunto de datos a procesar de manera de entender que transformaciones se necesitan aplicar.

El total de los conjuntos de datos que se tienen son:

1. Transactions *(múltiples archivos)*
2. Transaction Items *(múltiples archivos*)
3. Menu Items *(un archivo)*
4. Stores *(un archivo)*
5. Users *(múltiples archivos)*
6. Vouchers *(un archivo)*
7. Payment Methods *(un archivo)*

Dentro de los mismos, se consideran estrictamente necesarios los primeros dos conjuntos, ya que ofrecen la información principal que el cliente quiere analizar. En cuanto al resto de conjuntos, los siguiente tres (menu items, stores, users) se utilizaran para hacer uniones con la información principal obtenida, con el fin de que el resultado final sea un resumen fácil de entender al leerlo (sin números de identificación, etc).

Esto quiere decir que se deja sin uso a los últimos dos conjuntos de datos, ya que no son estrictamente necesarios para obtener el resultado esperado por el cliente.

4 Procesamiento y flujo de datos

En esta sección se tiene el objetivo de demostrar el funcionamiento del sistema a gran escala, explicando la cadena de procesos por la que pasara la información, así como también la comunicación entre el cliente y el sistema, y la comunicación interna.

4.1. Cadenas de procesado (pipelines)

A continuación, se demuestra como la información pasa por varias etapas de procesamiento en forma de cadena, hasta llegar al cliente, representado como sumidero *(sink)*.

4.1.1. Requisito (1)

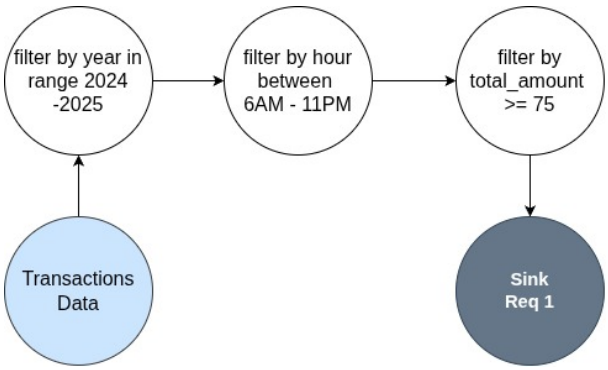


Figura 2: Procesado y confección del resultado para el primer requisito.

Como se puede observar en la figura, para este requisito se toma la información de las transacciones y se lleva a cabo una serie de filtrados con diferentes criterios (como por año, hora y cantidad total), para luego hacer el envío del resultado al cliente.

4.1.2. Requisito (2)

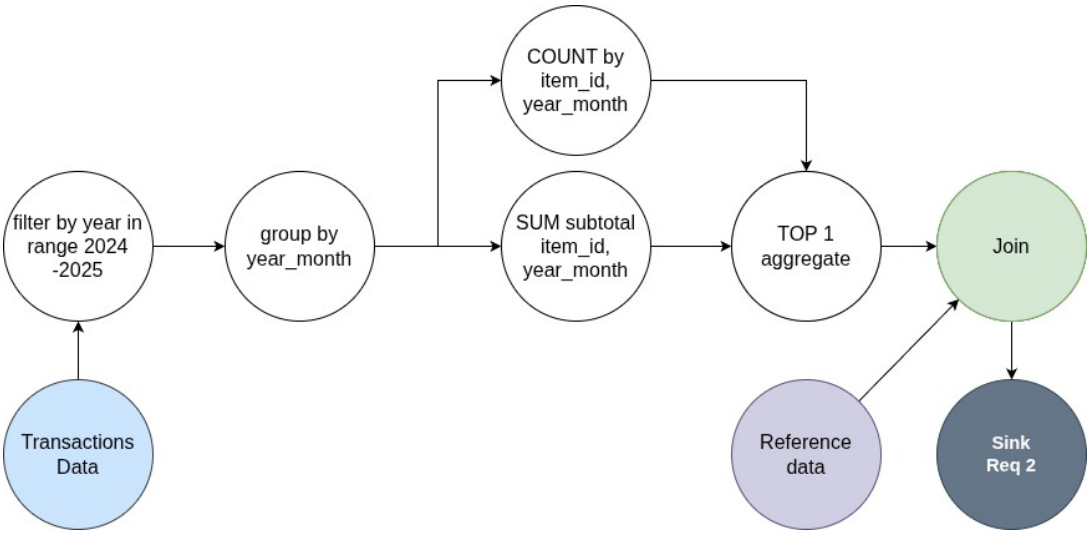


Figura 3: Procesado y confección del resultado para el segundo requisito.

Para el segundo requisito, el procesamiento ya no es tan trivial como un simple filtrado, sino que acá la cadena se divide en dos caminos luego de filtrar y agrupar los datos por año y mes. Los dos caminos hacen operaciones de reducción diferentes pero ambos convergen en el *aggregator*, que luego de desemboca en un *joiner* y este en el sumidero.

4.1.3. Requisito (3) y (4)

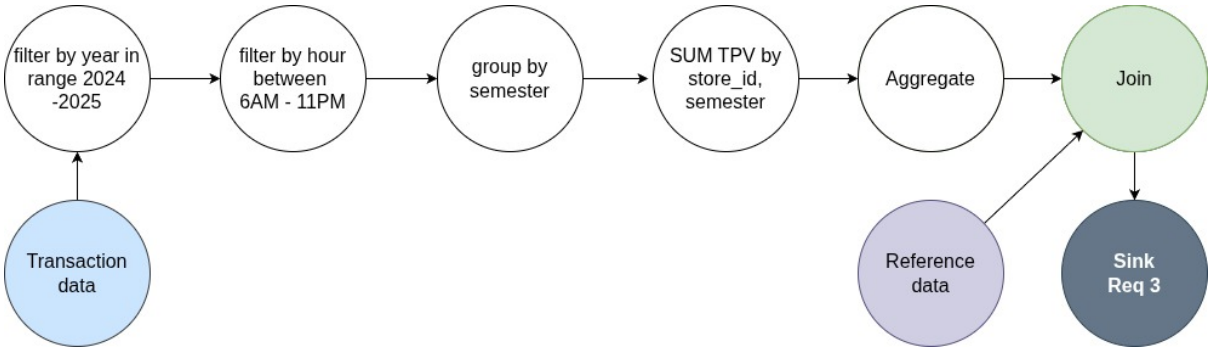


Figura 4: Procesado y confección del resultado para el tercer requisito.

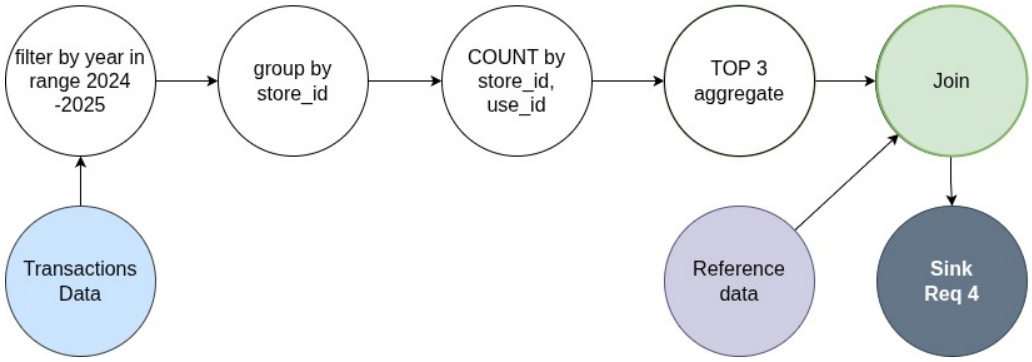


Figura 5: Procesado y confección del resultado para el cuarto requisito.

En el caso del tercer y cuarto requisito se lleva a cabo un proceso similar al del segundo, pero se simplifica un poco el flujo, ya que no hay una bifurcación para realizar la reducción (suma/conteo). Sin embargo ambos se diferencian principalmente en el comportamiento del paso de agregación.

4.2. Flujo de mensajes y procesos (punta a punta)

En esta sección se presentan flujos de punta a punta que ilustran el funcionamiento general del sistema mediante diagramas.

A continuación, se detallan dos casos: uno mas abstracto de las complejidades inherentes, proponiendo una visión del procesamiento como un modelo de caja negra, y luego otro donde se profundiza en dicha caja negra para entender mejor la comunicación interna del sistema.

4.2.1. Resumen de comunicación del sistema

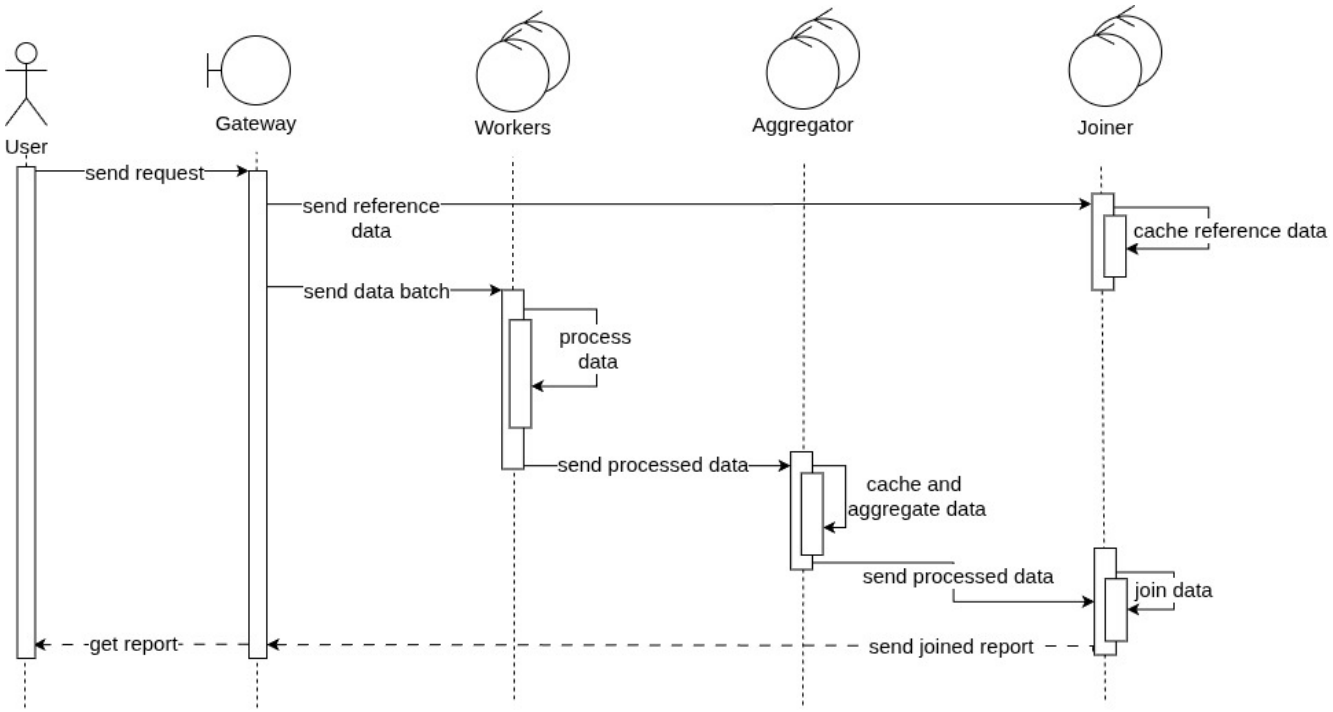


Figura 6: Funcionamiento general de sistema, con un resumen a gran escala del funcionamiento de los *workers*

El usuario interactúa únicamente a través del *Gateway*, haciendo inicialmente una solicitud indicando la tarea que quiere ejecutar. Ante esta solicitud, el *Gateway* se la reenvía al *Controller* para que le confirme si el sistema tiene los recursos necesarios para afrontar la solicitud. De ser así, se le confirma al cliente y éste empieza a transmitirle al *Gateway* la *reference data*, que posteriormente será utilizada por el *joiner* para vincular *datasets*. Una vez enviada la información de referencia, el cliente comienza a enviarle al *Gateway* (en forma de *batches*) los datos de transacciones a procesar.

En la figura se resumen las tareas ejecutadas por los *workers*: tras finalizar el procesamiento, los datos son integrados almacenados y agregados en el *aggregator*, para posteriormente ser vinculados con los reference datasets en el *joiner*. Una vez transmitidos todos los *batches*, el *joiner* genera y envía el reporte al *Gateway*, quien finalmente le envía al usuario el reporte consolidado.

4.2.2. Profundización sobre la comunicación del sistema

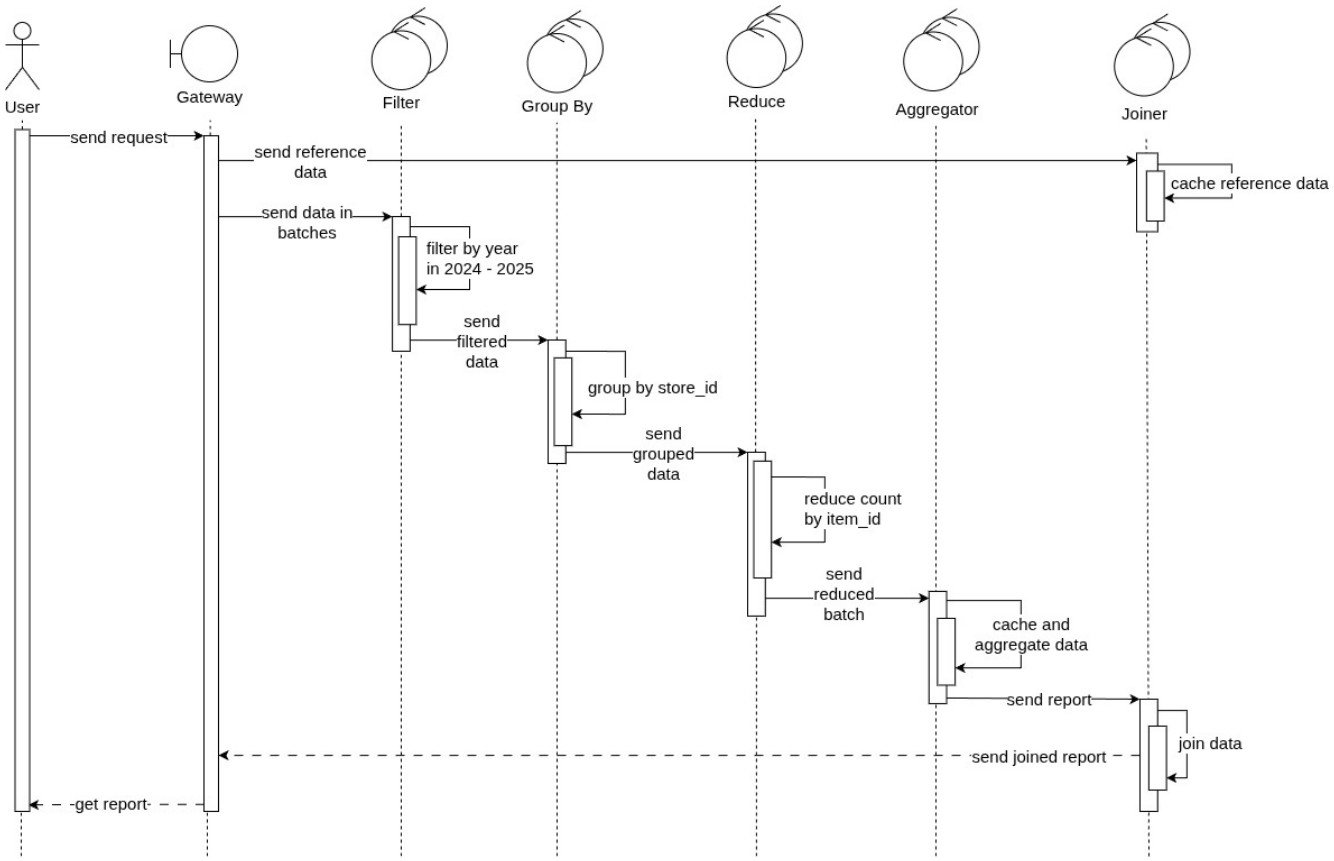


Figura 7: Requisito (3), Calcular el TPV (Total Payment Value) para cada sucursal entre 2024 y 2025

Como se observa en la figura de arriba, los *workers* procesan cada *batch* aplicando su tarea específica (*filter*, *groupby*, *reduce*) y posteriormente lo envían al siguiente *worker*, donde la información se refina mediante la operación correspondiente. Este flujo se repite hasta llegar al *aggregator*, quien se encarga de generar el reporte agregado para luego enviársela al *joiner*, que vincula los datos procesados con el *dataset* de referencia, para finalmente crear el reporte final y ser enviado al cliente.

4.3. Actividades de cada proceso del sistema

En esta sección se describen las actividades realizadas por cada actor del sistema, tomando como caso de referencia el requisito 2.

4.3.1. Punto de Entrada

En principio, se observa la interacción entre los nodos *Client* y *Gateway*, que funcionan como punto de entrada al sistema.

El *Gateway* constituye la interfaz utilizada por el usuario para enviar comandos y recibir reportes.

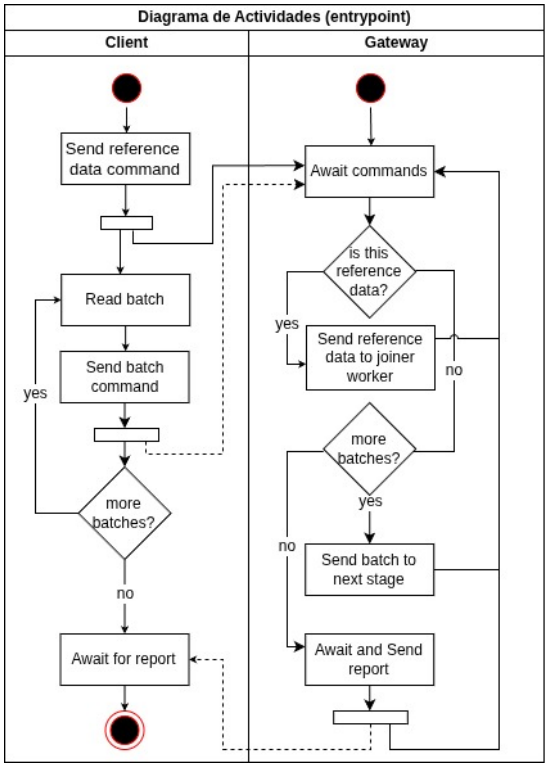


Figura 8: Actividades del punto de entrada al sistema.

A través del *Gateway*, primero se envía un *dataset* de referencia que será utilizado posteriormente para vincular los datos procesados. Luego, el *Gateway* transmite los *batches* de datos hasta agotar la información disponible, quedando a la espera de la generación del reporte. Durante este proceso, el *Gateway* recibe cada comando y determina si corresponde a *reference data*, *batch data* o a la finalización del envío de *batches* por parte del usuario (este último proceso se detalla más adelante).

4.3.2. Procesamiento

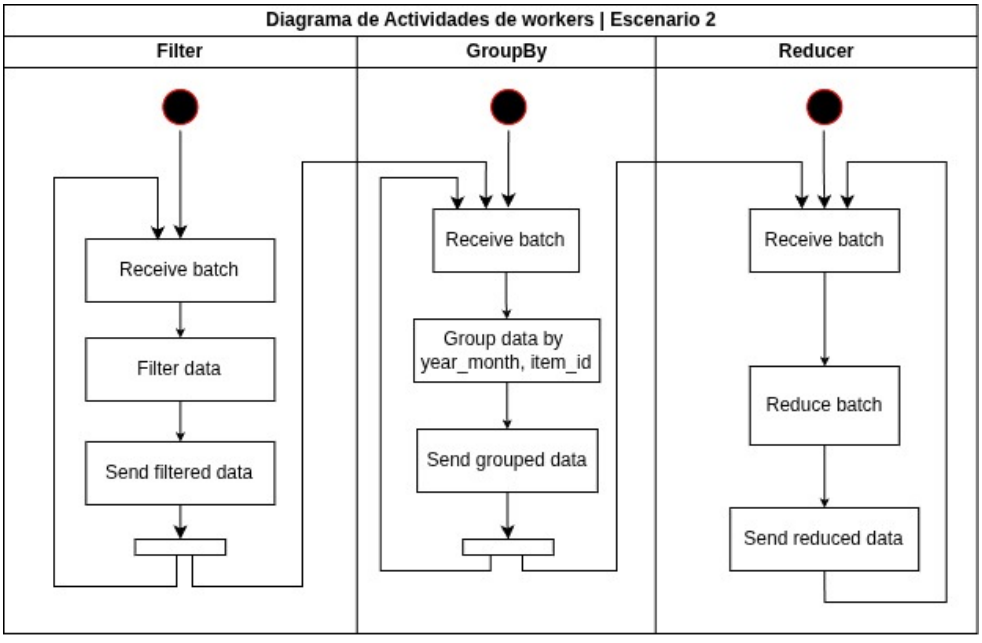


Figura 9: Actividades principales de procesamiento de información.

Cuando el *Gateway* recibe un *batch* de información para ser procesado, este es enviado inmediatamente al *worker* correspondiente. Para el caso de la data de referencia, se envía al worker *Joiner*, y para el caso de la data en sí, se envía al worker *Filter*.

Como se observa en la figura anterior, cada *worker* ejecuta tareas similares que consisten en recibir los datos, procesarlos y remitir el resultado al siguiente *worker*, encargado de continuar el refinamiento de la información.

Cada uno de los workers tienen estos tareas que hacer:

- **Filter Worker**: selecciona datos en función de criterios definidos (por ejemplo, rango temporal).
- **GroupBy Worker**: agrupa los registros según diferentes atributos, como el ID de un ítem o el mes del año.
- **Reducer Workers**: realizan operaciones de reducción, ya sea sumatoria o conteo.

4.3.3. Post-procesamiento

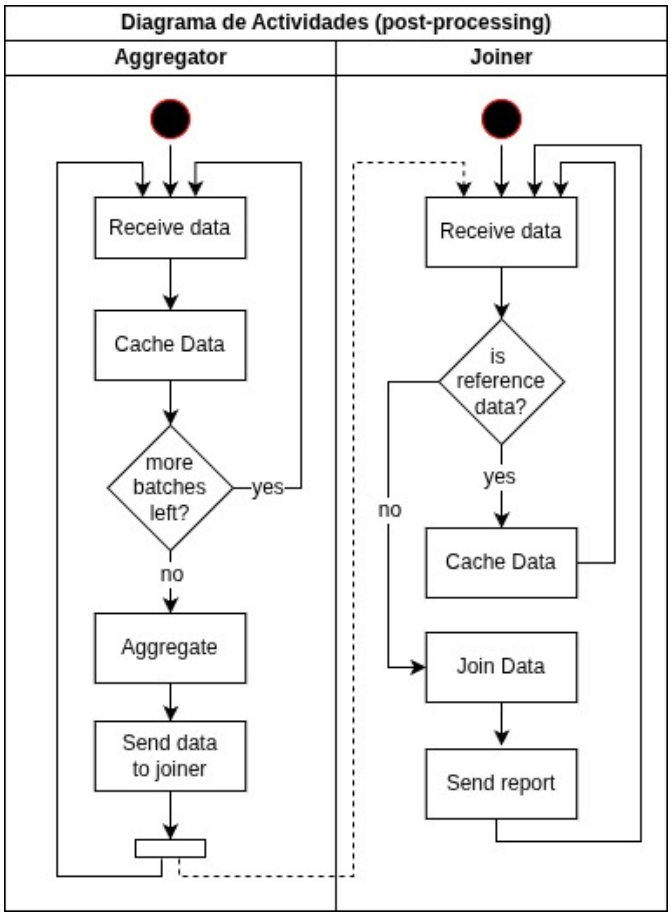


Figura 10: Actividades principales de post-procesamiento de información.

Este diagrama detalla el funcionamiento interno de los *workers* responsables del post-procesamiento, en particular el *Joiner* y el *Aggregator*. El *Joiner* recibe la *reference data* directamente del *Gateway* y la almacena en su caché. Además, recibe los datos reducidos para vincularlos con el *dataset* de referencia, y posteriormente los envía al *Gateway*.

El *Aggregator* tiene la responsabilidad de recibir y almacenar los datos hasta completar la recepción de todos los *batches* procesados, con el fin de generar el reporte con los datos agregados. Dicho reporte es enviado al *Joiner* y, posteriormente, al *Gateway*.

4.3.4. Fin de transmisión

Por completitud, en la siguiente figura se ilustra el comportamiento del sistema ante la finalización del *input* del cliente. El diagrama inicia en el Controller y engloba la actividad de todos los workers para simplificar la representación.

Como se ve en el diagrama, el Controller lleva un conteo de la cantidad de batches que envían y reciben las etapas de procesamiento, recibiendo los *MessageCounter* de cada etapa. Precisamente, solo el Controller y el Filter envían los *MessageCounter* al Controller. Inicialmente, el Gateway debe indicarle cuántos batches recibió del cliente y, acto seguido, envió al Filter. Posteriormente, los *Filter* tienen que indicar cuántos de esos batches que recibieron fueron enviados a la siguiente etapa de procesamiento (el *Group By* en todas las tasks, menos en la 1, donde el Filter manda directamente la data al *Gateway*), ya que estos workers podrían filtrar esos batches y modificar la cantidad enviada. De acá en adelante, ninguno de los workers restantes va a modificar la cantidad de batches que se envían a lo largo del pipeline de procesamiento. Por lo tanto, la cantidad de batches que sale de los *Filter* será la cantidad que recibirá el *Aggregator* (o el *Gateway* en la Task 1). Es por esto que nos alcanza con controlar la cantidad de batches que se reciben y se envían solo en el *Gateway* y en el *Filter*.

Una vez que la cantidad de batches enviados por el *Gateway* coincide con la cantidad de batches ya procesados por los workers *Filter*, el Controller sabe que, a partir de ese momento, la cantidad de batches no se modificará. Por lo tanto, el Controller envía un mensaje al *finishExchange* con la routing key *aggregator*, indicando la cantidad de batches que deben esperar los *Aggregator* antes de considerar como finalizado a ese cliente y empezar a mandarle el batch procesado al *Joiner*.

Finalmente, una vez que el *Gateway* termina de recibir toda la data procesada del *Joiner*, envía otro mensaje al *Controller* indicándole que el procesamiento para ese cliente finalizó, y que puede notificarle a todos los workers para que borren toda la data que tenían almacenada para ese cliente.

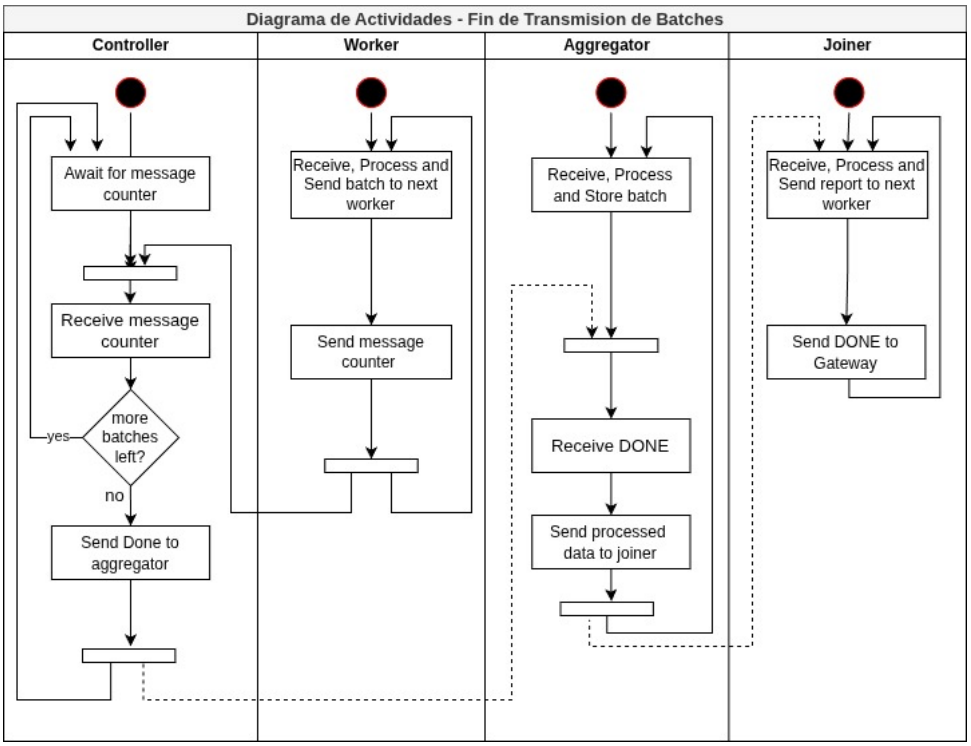


Figura 11: Actividades principales de fin de transmisión de batches.

4.3.5. Limpieza de cliente ya finalizado

Una vez que el *Gateway* recibió todos los batches del informe final y se los envió al cliente, le envía un mensaje de finalización al *Controller*. Una vez que recibe este mensaje, le envía un mensaje de finalización a todos los workers. Los workers, al recibir este mensaje de finalización, eliminan los registros de los números de secuencia de los mensajes que fueron procesando. Además, en el caso de los workers que almacenan datos (*Joiner* y *Aggregator*), se elimina toda esa información referida a ese cliente.

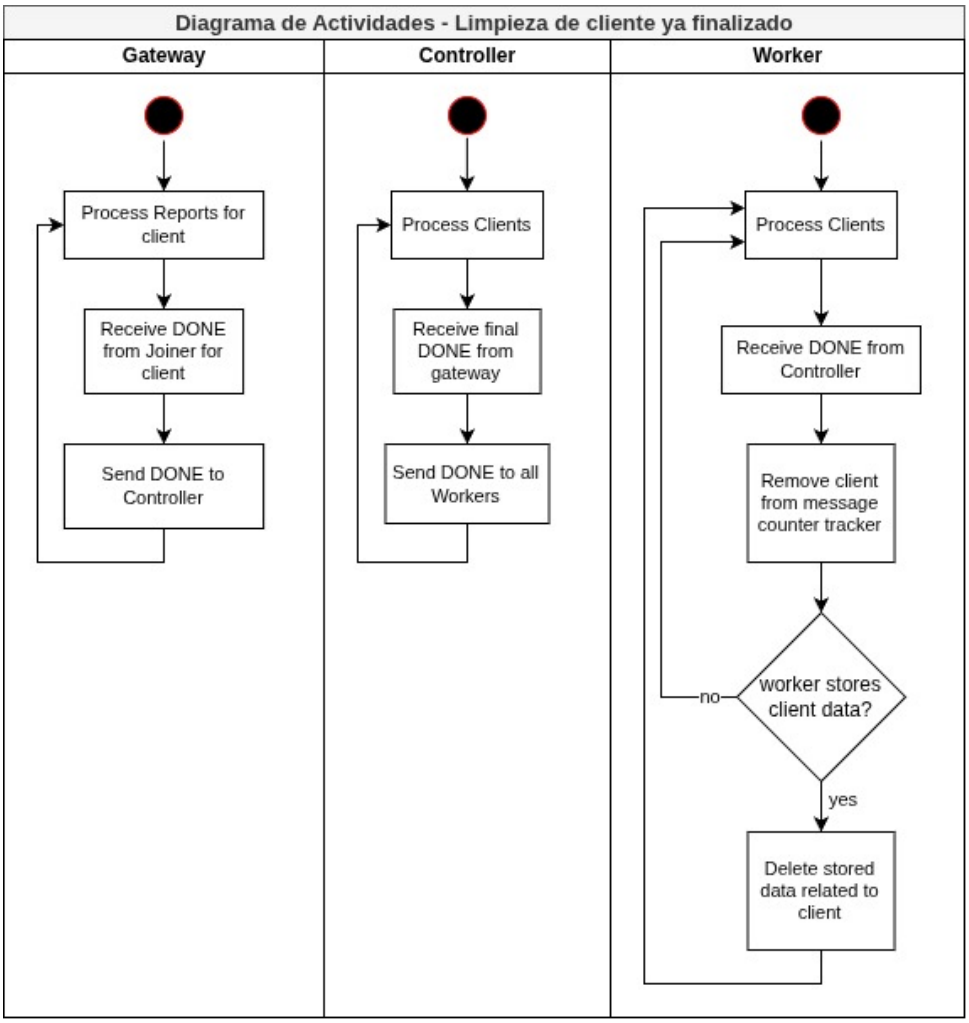


Figura 12: Actividades principales de limpieza de clientes ya finalizados

5 Arquitectura e Infraestructura

Para conformar la cadena de procesos mencionada en la sección anterior, se usara la estructura 'Worker per filter', que consta de tener una unidad de procesamiento aislada para cada etapa intermedia (filter, groupby, reduce, join y aggregate). Se opto por esta división ya que en el contexto de un sistema altamente distribuido, lleva a un mejor aprovechamiento de recursos, paralelismo y menos carga por *worker*, así como también mayor escalabilidad horizontal, a comparación de la división 'Worker per item'. Sin embargo, esta estructura también conlleva una mayor complejidad y necesidad de coordinación y comunicación entre nodos.

5.1. Arquitectura y medios de comunicación

En esta sección se presenta un diagrama de robustez, donde se ilustran las principales interacciones entre actores, fronteras y colas del sistema.

Se observa que toda comunicación entre nodos (exceptuando la comunicación cliente-gateway) se realiza a través de colas de mensajes. Cada worker toma información de una cola de entrada, la procesa y la envía a una cola de salida, de la cual leerá el próximo nodo en la linea de procesamiento. A su vez, las *replicas* de los nodos se grafican como círculos duplicados y apilados.

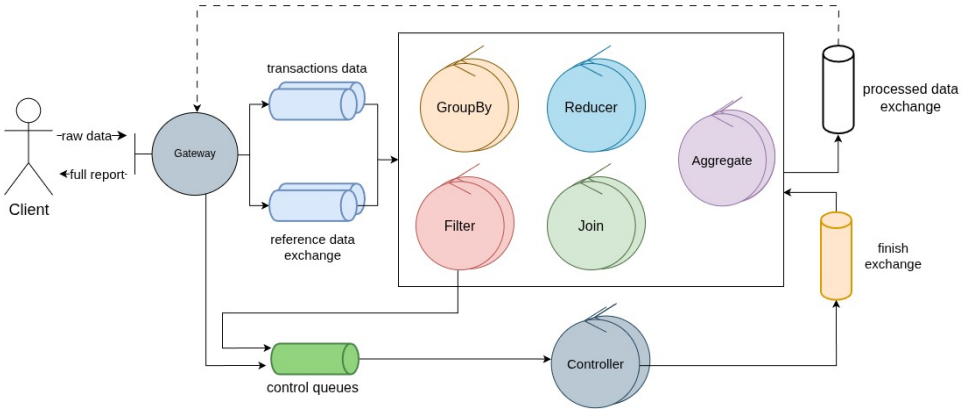


Figura 13: Visión general de la interacción entre *workers* y la transmisión de información mediante colas de mensajes.

En las siguientes figuras se amplían partes de interés del anterior diagrama de robustez, de modo de demostrar un seguimiento paso a paso de las interacciones.

5.1.1. Arquitectura de procesos para cada requisito

A continuación se profundiza la arquitectura utilizada para el procesamiento de cada requisito, tomando como punto de partida el proceso inicial común mencionado.

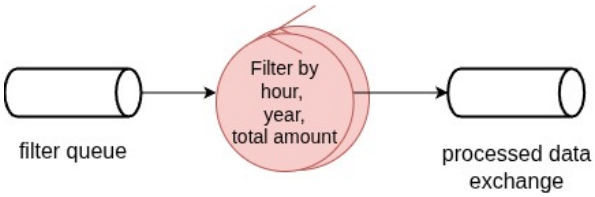


Figura 14: Camino relacionado al requisito 1.

En la figura previa se demuestra el flujo correspondiente al requisito 1. El proceso inicia con un filtrado por hora, año y por monto total de las transacciones, obteniendo finalmente los datos procesados, los cuales son enviados al *exchange* de la data procesada.

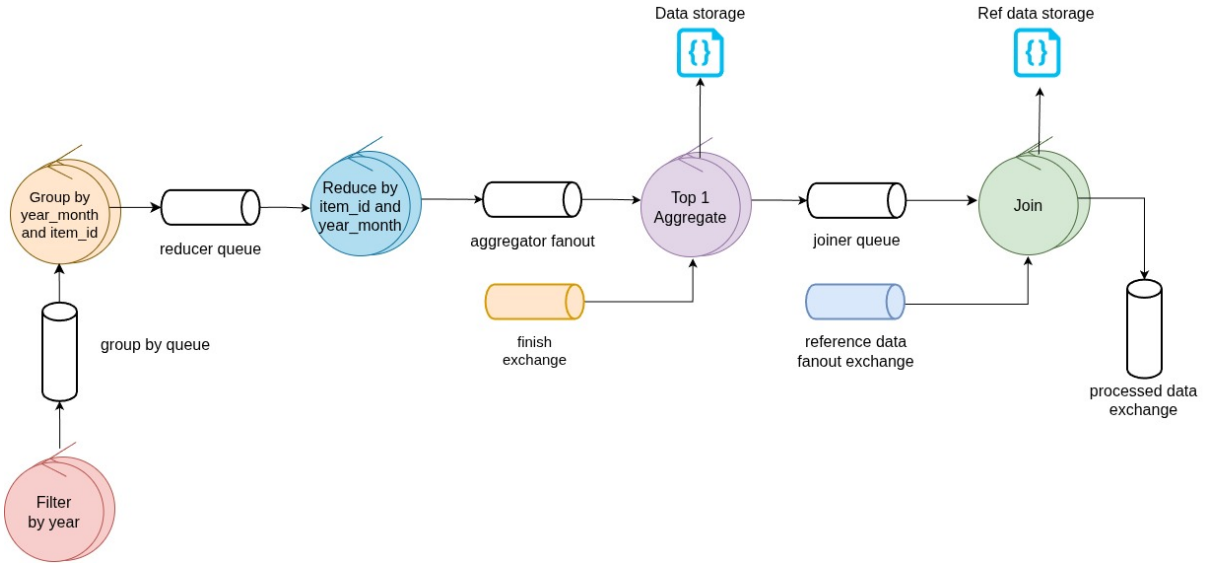


Figura 15: Camino relacionado al requisito 2.

La figura ilustra el flujo correspondiente al requisito 2. En este caso, las transacciones se agrupan por mes y posteriormente se envían al *worker reducer*, encargados de contar la cantidad de ítems y de calcular la suma del *final_value* de cada ítem. A continuación, cada conjunto de datos pasa por la etapa de *aggregator*. Este último no solo almacena los *batches* procesados, sino que además ordena los resultados para seleccionar el *Top 1* ítem, ya sea el más vendido (mayor *count*) o el que generó mayores ganancias (mayor *final_value*). Finalmente, el reporte agregado es enviado al *joiner*, donde se vinculan con el *dataset* de referencia, y se envían al *Gateway* mediante el *exchange* de la data procesada.

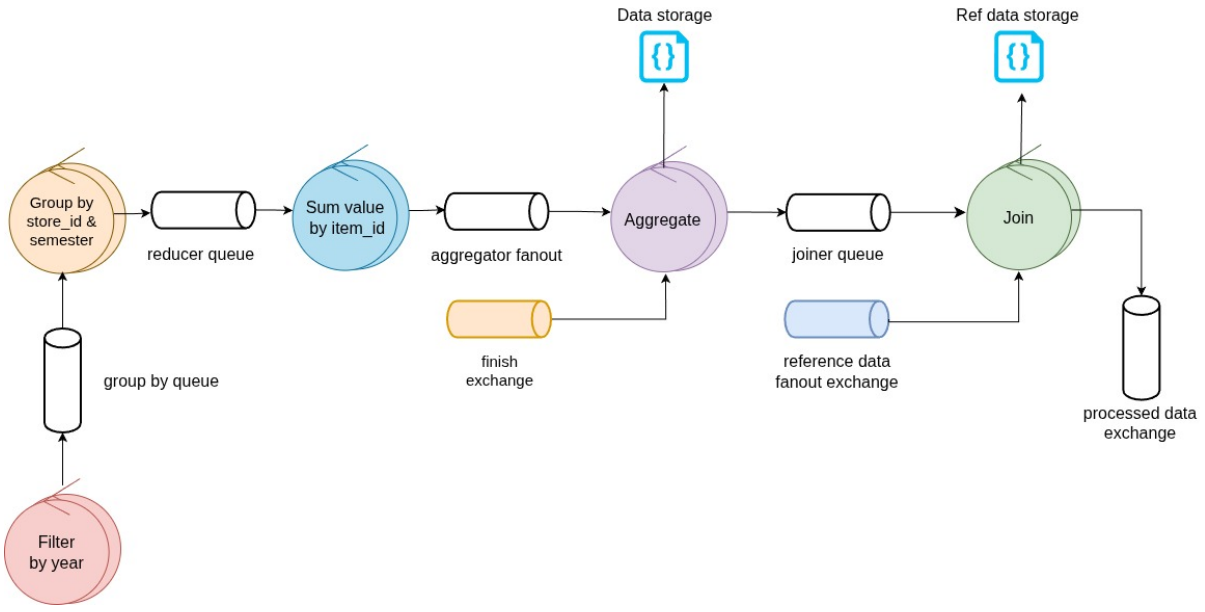


Figura 16: Camino relacionado al requisito 3.

La figura muestra el flujo asociado al requisito 3. En este escenario, las transacciones se agrupan por *store_id* y semestre, para luego calcular la suma de valores por *item_id*. Posteriormente, los resultados se consolidan en el *aggregator*, obteniendo el reporte procesado, el cual es enviado a la etapa de *join* en donde es vinculado con el *dataset* de tiendas (*stores*).

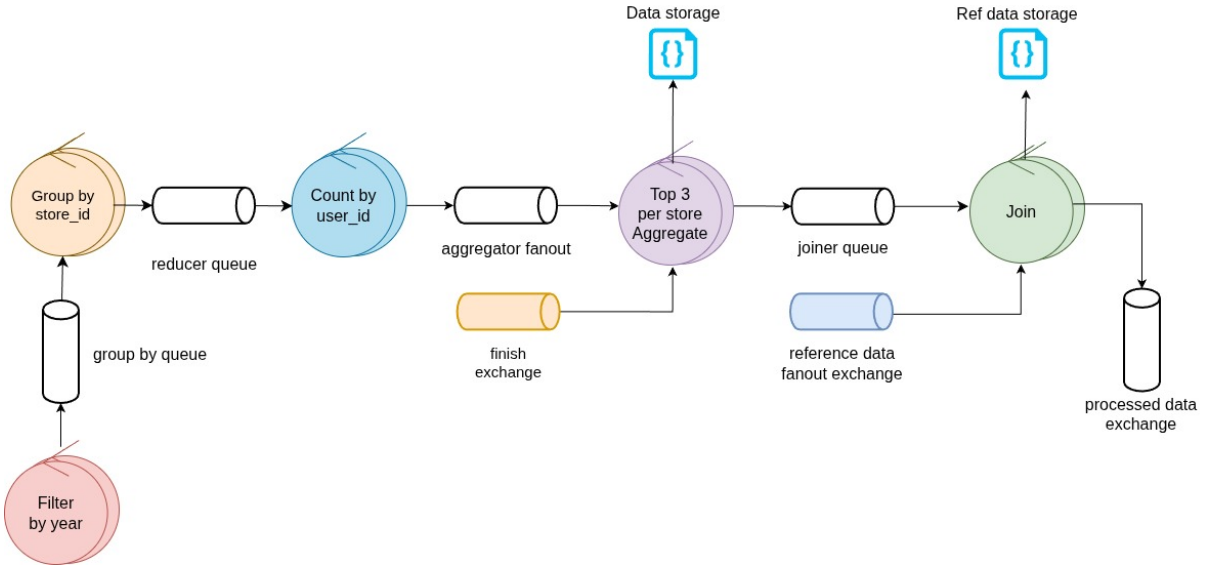


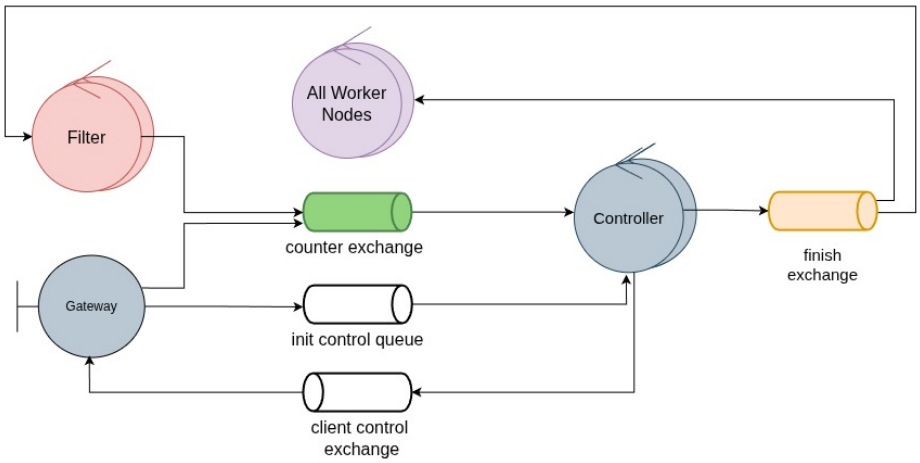
Figura 17: Camino relacionado al requisito 4.

La figura ilustra el flujo correspondiente al requisito 4. Las transacciones se agrupan por *user_id* y luego se cuentan en un *worker reducer*. Luego, el *aggregator* consolida la información y selecciona el *Top 3* de usuarios en función a la cantidad de transacciones hechas por usuario.

Observación: Cabe mencionar que el cierre ordenado del sistema en su completitud sera manejado mediante el uso de un exchange definido especialmente para este propósito (finish_exchange).

5.1.2. Arquitectura de secuencia de control

A continuación, se profundiza la arquitectura utilizada para el control del fin de transmisión y de la inicialización de los clientes.

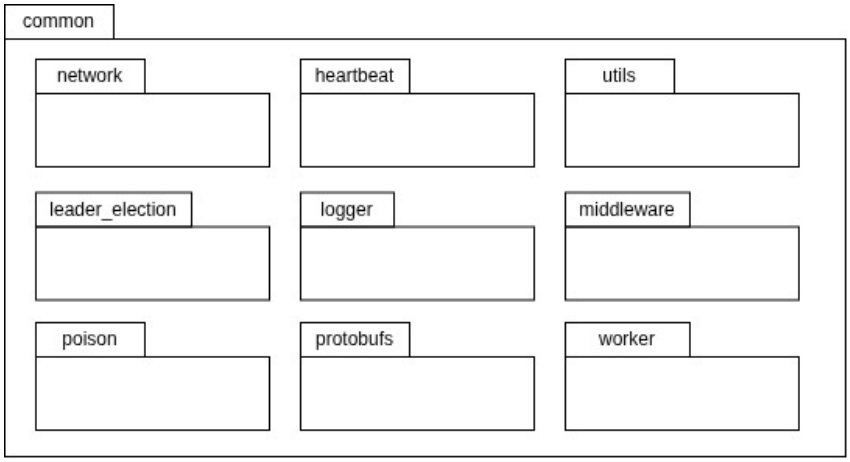


Como se explicó en las secciones anteriores, el Controller lleva un conteo de la cantidad de batches que envían y reciben las etapas de procesamiento, recibiendo los *MessageCounter* de cada etapa. Para recibir estos mensajes es que se utiliza la *counter exchange*.

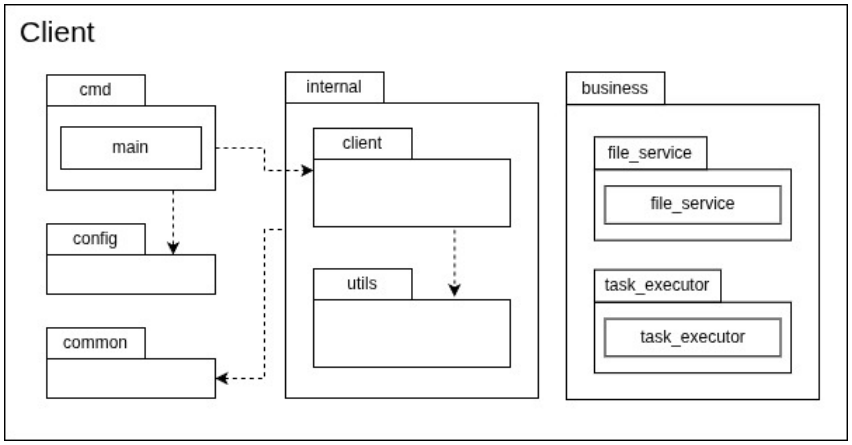
Por otro lado, dijimos que el usuario hace inicialmente una solicitud al *Gateway* indicando la tarea que quiere ejecutar, y ésta es reenviada al *Controller*. Este reenvío se hace mediante la *init control queue*, de donde el *Controller* consume las solicitudes y envía las respuestas mediante la *client control exchange*.

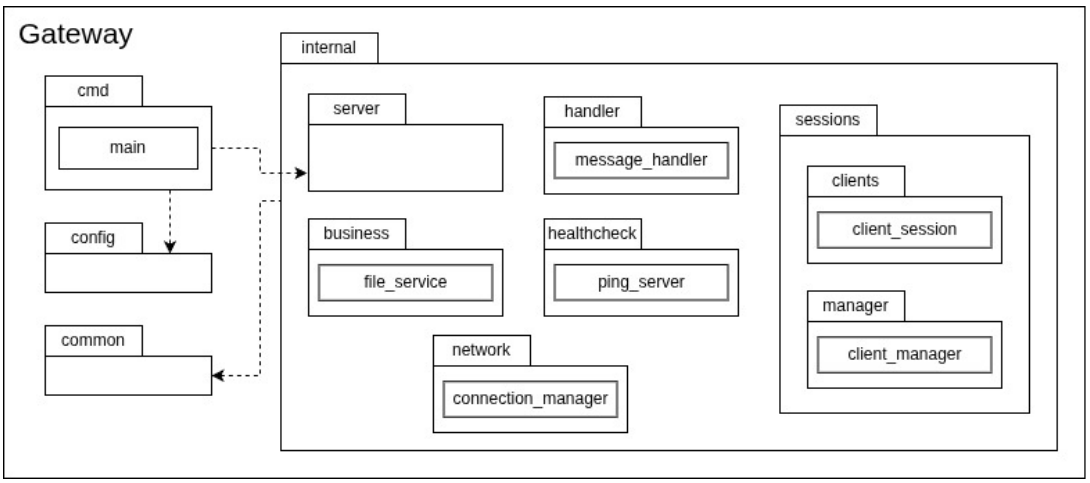
5.2. Estructura de paquetes y módulos de cada actor

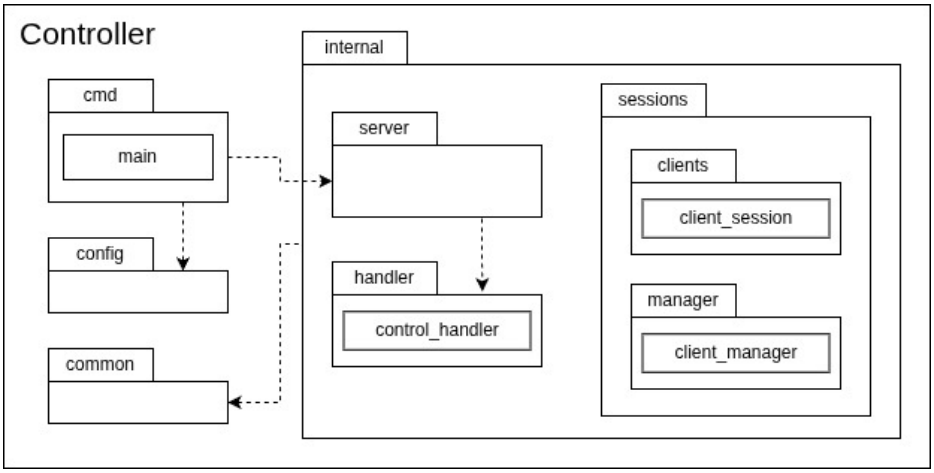
Esta sección muestra los diagramas de la estructura de módulos y paquetes de código definida durante la implementación del sistema. El módulo *Common* es el que contiene comportamiento relacionado a la red, comunicaciones y protocolos (e.g. protobufs).



El paquete de *Client* contiene lógica de envío de información al servidor y recepción de datos completamente procesados, para lo que se comunica a través de la red con el servicio *Gateway* cuyo paquete contiene comportamiento de recepción de clientes y despachado de tanto los datos crudos enviados por el cliente como los datos procesados saliendo del sistema. El paquete de *Controller* contiene lógica de recepción de clientes y control de los datos que están dentro del sistema.







Los paquetes de 'Filter', 'GroupBy' y 'Reducer', contienen la lógica de negocio principal del sistema, con la que se realizara el procesado completo de datos de manera distribuida.

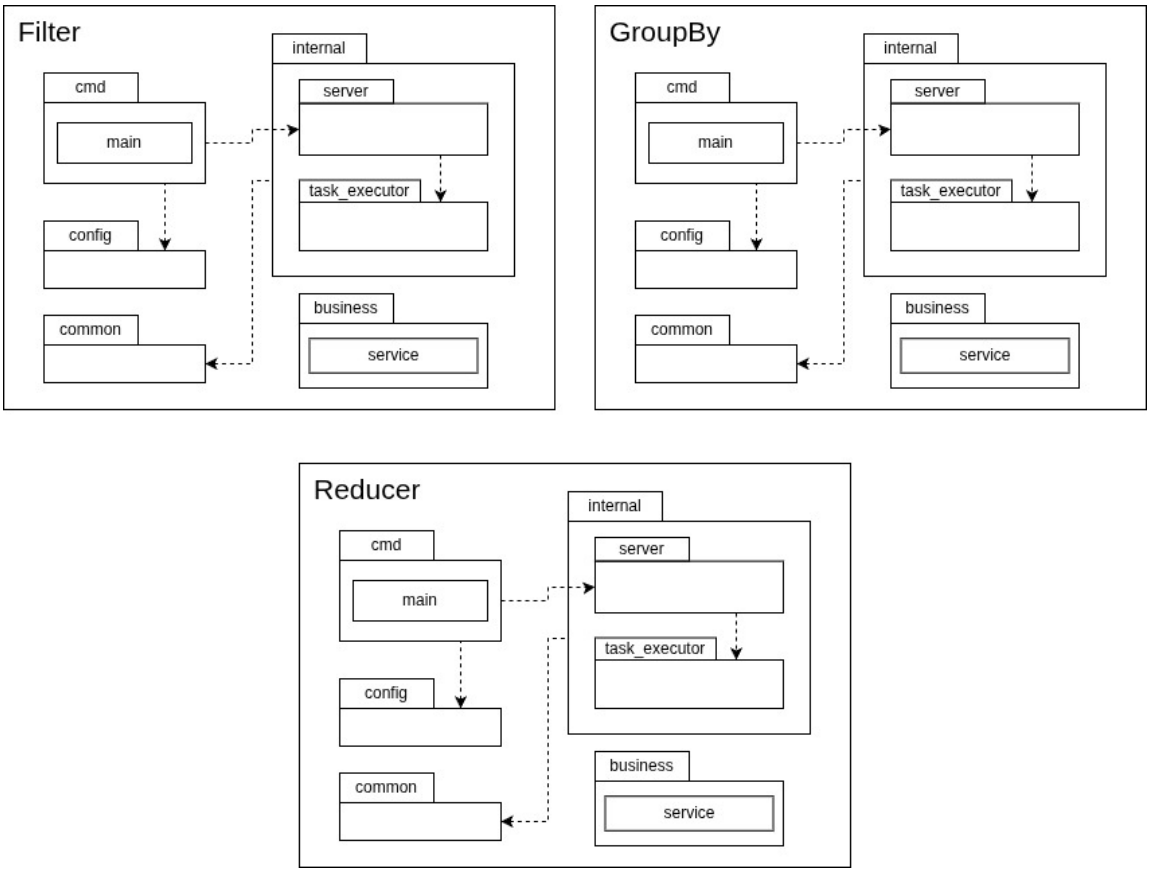


Figura 18: Paquetes de *workers* de filtrado/reducción de datos.

Los dos últimos paquetes diagramados constan del comportamiento relacionado al refinado de datos (unir con los conjuntos de datos correspondientes, e.g. Transactions + Menu Items), y unificación de la información distribuida para poder ser enviada al cliente. Estos cuentan con un **cache** en disco de los datos procesados para evitar perdida de información durante re-inicios del *worker* por problemas inesperados.

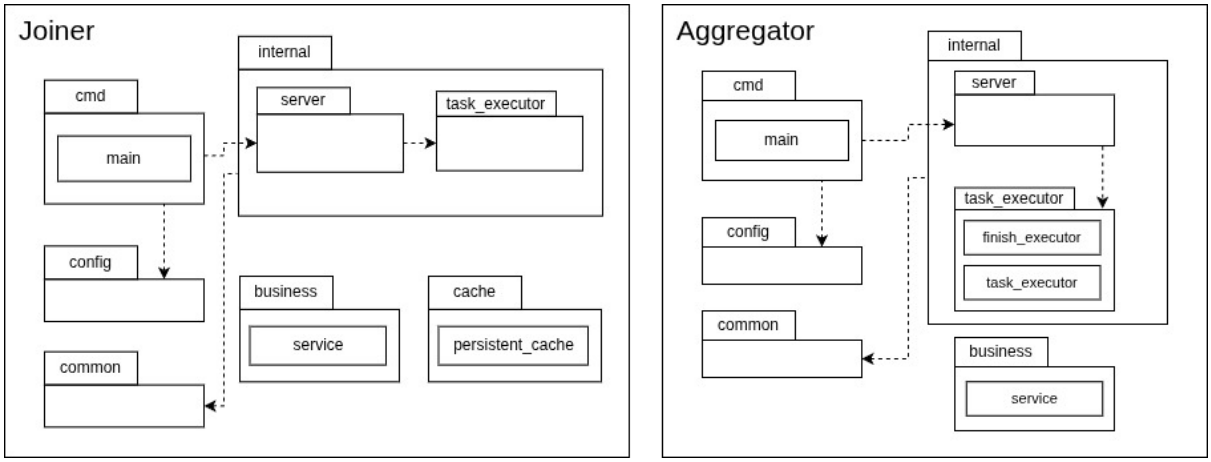


Figura 19: Paquetes de *workers* de post-procesamiento y unificación de la información.

5.3. Infraestructura del sistema

A continuación se detalla la infraestructura ideada para el funcionamiento distribuido del sistema, haciendo foco en la confiabilidad ante fallos y escalabilidad horizontal (agregando más nodos/computadores).

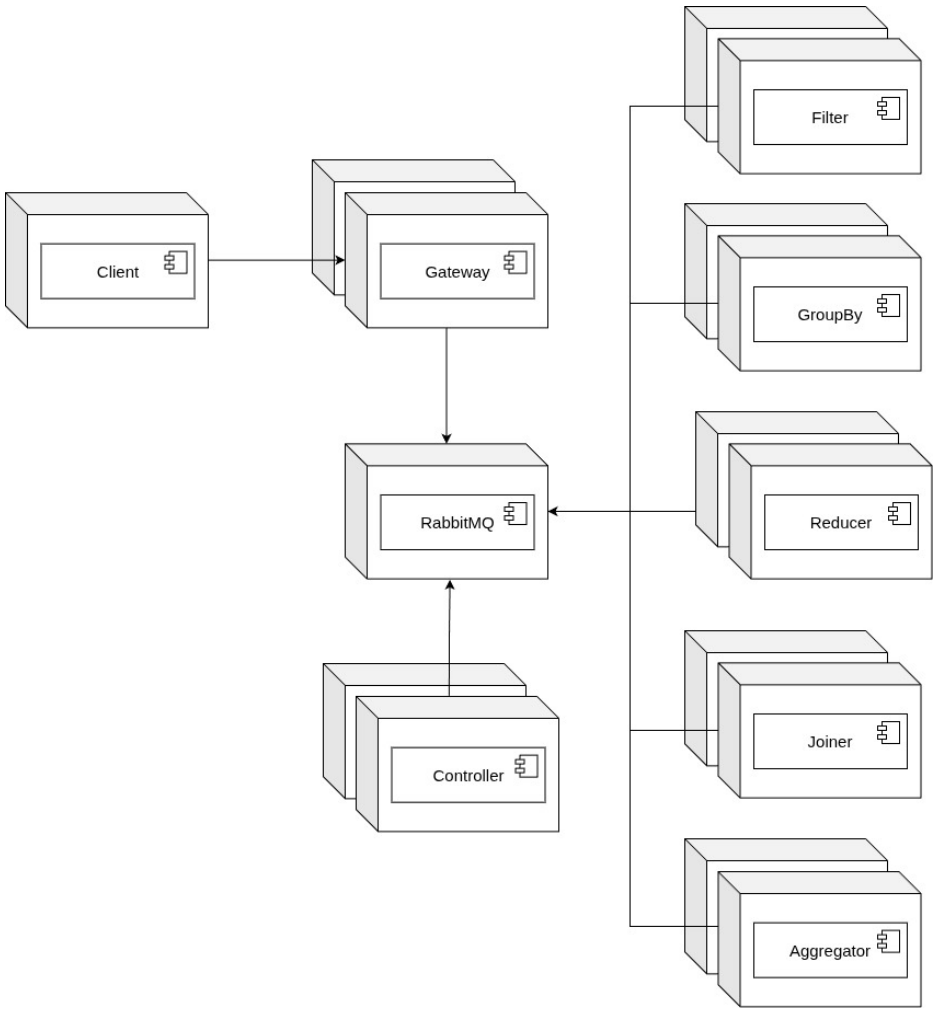


Figura 20: Arquitectura necesaria para desplegar el sistema.

Características:

- Se demuestra al cliente como un nodo aparte, que interactúa directamente con un *Gateway* para subir sus conjuntos de datos a ser procesados. Este ultimo como se menciono anteriormente, cumple el rol de punto de entrada al sistema.
- Entre nodos del sistema se comunican a través de un Middleware orientado a mensajes (MOM), utilizando colas.
- Reafirmando lo dicho en la sección de 'Vista de Desarrollo', los *worker nodes*, están separados por filtro, es decir hay un solo tipo de *worker* por computador. Alternativamente se puede modelar el sistema con procesos tomando el rol de nodos, en caso de no contar con un entorno multi-computadoras.
- Se toma como **único punto de falla** el servicio de colas de mensajería, proporcionado por RabbitMQ, se evalúa la posibilidad de tener replicas para mayor disponibilidad.
- Los nodos de *Gateway*, *Controller* y *Aggregator* se identifican como componentes críticos del sistema, por lo que incorporamos réplicas que aumenten la tolerancia a fallos, lo cual se explica más en detalle en las próximas secciones.

6 Tolerancia a fallos

6.1. Health Check

Uno de los componentes más importantes que incorporamos en el sistema para manejar la tolerancia a fallos es el *Egg of Life*, cuya responsabilidad es monitorear el estado de los demás nodos y, en caso de detectar una falla, reiniciar dicho nodo para que se reincorpore al funcionamiento del sistema.

El monitoreo se lleva a cabo mediante conexiones UDP entre los nodos y el *Egg of Life*. Utilizando un intervalo de tiempo configurable (*timeout_interval*), un nodo se considera inactivo si la cantidad de tiempo que pasó desde que se recibió el último heartbeat de él es mayor a ese intervalo en cuestión. Lógicamente, el *Egg of Life* mantiene actualizado el momento en el que le llegó el último heartbeat para cada nodo, y la acción de chequear si algún nodo superó el intervalo configurado se hace cada cierta cantidad de tiempo también configurable (*check_interval*).

6.2. Manejo de duplicados

Para el manejo de duplicados, decidimos que todos los mensajes (para un cliente en específico) tengan un número de secuencia único y secuencial. Esto lo hacemos tanto para la data de referencia que se envía inicialmente como para los batches de la data a procesar.

Este número de secuencia nos permite llevar un registro de los mensajes que ya fueron enviados y procesados para cada cliente en cada una de las capas. Así, si por algún motivo alguno de los nodos recibe un nodo duplicado, va a poder detectarlo a partir del número de secuencia y descartarlo. Este mapa de números de secuencia recibidos por cliente es algo que tienen todos los nodos del sistema, cada uno con el objetivo de descartar un determinado tipo de mensajes.

Además, todos los nodos workers llevan un registro de los IDs de los clientes que ya fueron finalizados (es decir, para los cuales se recibió el mensaje de fin de transmisión desde la *finish_exchange*). Esto nos sirve porque, al recibir el fin de transmisión para un cliente, cada uno de los workers va a eliminar toda la información que almacenaba para ese cliente en particular (números de secuencia, reference data en el caso del *Joiner*, etc). Entonces, esto nos sirve para evitar procesar mensajes duplicados que nos puedan llegar fuera de término cuando el cliente ya fue finalizado (dado que, al borrar la información del cliente, los workers ya no tienen conocimiento de los números de secuencia que ya procesaron para ese cliente en el pasado, y no considerarían a ese mensaje como duplicado aunque sí lo es).

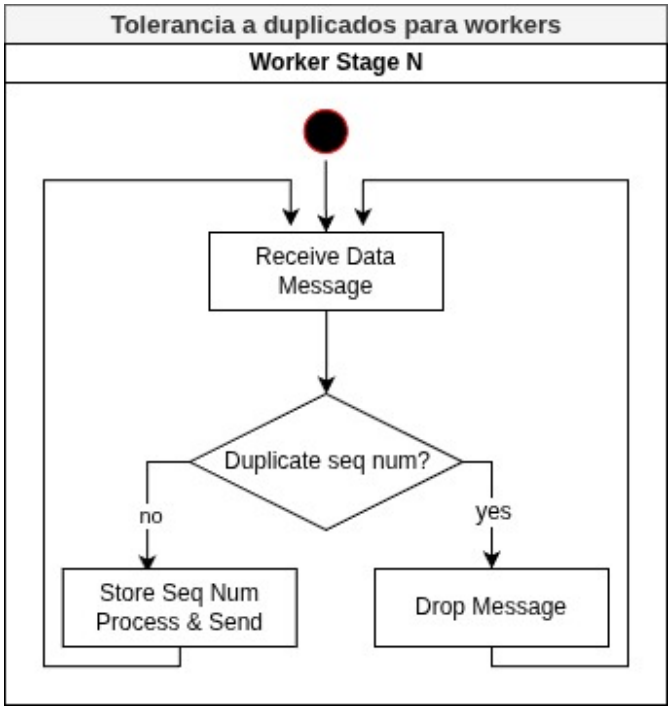


Figura 21: Tolerancia a duplicados en los workers

Uno de los grandes beneficios de esto, más allá de no considerar los mensajes duplicados que nos puedan llegar en cualquier momento y re-procesarlos, es que el mensaje de fin de transmisión es **idempotente**. Es decir, al enviarse repetidamente dicho mensaje, no cambia el resultado final obtenido. Esto es simplemente porque dicho mensaje final (y la data que podría llegar a enviarse según a qué worker le llegue ese mensaje de fin duplicado) va a ser filtrado en alguna de las capas, teniendo también un manejo de duplicados en el *Gateway* como última instancia antes de mandarle el reporte final al cliente.

6.3. Tolerancia a fallos en workers

Para manejar la tolerancia a fallos en los workers y evitar que se pierdan los batches de data que se estaban procesando cuando se produjo la caída (y que no llegaron a entregar al siguiente worker en el pipeline de procesamiento), hicimos que los workers realicen el ACK de dicho batch de data una vez que éste fue procesado y enviado a la siguiente capa. Así, si se cae mientras el worker estaba procesando el mensaje, éste último quedará sin "ACKear", y ese batch de data será re-encolado en la cola de RabbitMQ, permitiendo que otro worker de la misma capa pueda tomar y procesar ese mensaje.

Para el caso especial de los *Filter* workers (que son los únicos workers que envían los *MessageCounter* al *Controller*), el ACK del batch de data que están procesando se manda posteriormente a enviar dicho mensaje de control al *Controller*. Esto nos permite evitar el caso en donde se envía un batch de data desde el *Filter* a la siguiente capa, pero sin que el *Controller* fuera notificado, haciendo que la cantidad de mensajes que espera recibir éste último del *Filter* nunca sea alcanzada (ya que se perdió un *MessageCounter*). De la forma en la que lo hacemos, nos aseguramos de que el *Controller* siempre sea notificado antes de "ACKear" un batch de data.

6.3.1. Joiner worker

Estos workers son un caso especial porque, como se aprecia en los diagramas de robustez, tienen persistencia de los datos que reciben. Precisamente, el *Joiner* almacena la data de referencia que le llega desde el *Gateway*.

Para garantizar la tolerancia a fallos en estos tipos de nodos, además de lo que hacemos en el resto de los workers, el sistema está diseñado para que, ante una caída, los nodos sean recuperados con su estado previo a la caída. Persistir los datos en disco dentro de cada nodo nos asegura (mediante el manejo adecuado) que dichos nodos sean levantados con todos los archivos que tenían antes de la caída.

Así, cuando un nodo *Joiner* se reinicia después de una caída, tendrá toda la data de referencia que ya había almacenado y podrá terminar de recibir todos los batches de este tipo que fueron enviados

mientras estaba caído (si es que no había llegado a terminar de almacenarlos). Esto último es posible porque la *reference_data_exchange* (usada por el *Gateway* para mandarle la data de referencia al *Joiner*) utiliza colas persistentes, permitiendo que todos los batches de referencia que sean enviados mientras el nodo está caído se mantengan hasta que éste último se reinicie y vuelva a consumirlos.

Durante una ejecución normal, cada mensaje de data de referencia que le llega al *Joiner* es escrito en un archivo en disco. Debido a esto, podría darse el caso de que el *Joiner* se caiga mientras se está escribiendo dicho archivo.

Por eso es que no se manda el ACK para dicho mensaje en ese momento, sino que nos "reservamos" el envío de ese ACK para realizarlo posteriormente. Precisamente, una vez que alcanzamos una cierta cantidad de mensajes recibidos, se realiza un *flush* de los datos a disco, y una vez que termina esa operación, realizamos el envío del ACK para todos los mensajes que habían quedado sin "ACKear".

De esta forma, evitamos mandar el ACK para un batch de datos que no llegó a ser persistido en disco, haciendo que dicho batch sea re-encolado en caso de que se produzca una caída del nodo y no haya llegado a enviar dicho ACK.

Cabe destacar que, al hacerlo de esta forma, podría darse el caso de que tengamos registros duplicados dentro del mismo archivo, porque el *Joiner* podría caerse en medio de un *flush*, haciendo que algunos de los mensajes sin "ACKear" lleguen a ser escritos, pero otros no. Entonces, cuando vuelva a escribir todos los mensajes que quedaron sin "ACKear", algunos serán escritos nuevamente, quedando duplicados en el archivo. Esto no es un problema porque, al tratarse de datos de referencia, lo único que hacemos con esa información es utilizarla para vincular los registros de datos que nos llegan del *Aggregator*. Entonces, si tenemos dos registros que referencian a la misma información, tomar cualquiera de ellos produce el mismo resultado final.

6.3.2. Aggregator worker

Al igual que los worker *Joiner*, estos workers también tienen persistencia de los datos que reciben. Precisamente, el *Aggregator* almacena los batches que le van llegando desde el *Reducer* para luego agregarlos y mandarlos al *Joiner*.

Para garantizar la tolerancia a fallos en estos tipos de nodos, hacemos una operatoria muy parecida a la del *Joiner*. Es decir, persistimos los datos en disco dentro de cada nodo para que dichos nodos sean levantados con todos los archivos que tenían antes de la caída.

Luego, al igual que el *Joiner*, cuando un nodo *Aggregator* se reinicia después de una caída, tendrá toda la información ya había almacenado, y podrá continuar recibiendo todos los batches de datos mandados por el *Reducer* mientras estaba caído (si es que no había llegado a terminar de almacenarlos). Esto último es posible porque el *exchange fanout* (en el cual se encolan los batches para el *Aggregator*) también utiliza colas persistentes, permitiendo que todos los batches que sean enviados mientras el nodo está caído se mantengan hasta que éste último se reinicie y vuelva a consumirlos.

6.4. Tolerancia a fallos en Gateway

El *Gateway* es el punto de entrada a nuestro sistema, y es el único nodo que mantiene conexiones TCP con los clientes (entre el resto de los nodos, la comunicación se realiza vía RabbitMQ y no tienen comunicación con el cliente).

Como explicamos anteriormente, el cliente va a establecer una conexión con el *Gateway* al inicio de la sesión, para mandarle la solicitud para la tarea que quiere realizar. La forma en la que se selecciona ese *Gateway* es completamente aleatoria, favoreciendo el *load balancing*.

Luego, si en algún momento el *Gateway* sufre una caída, el cliente va a intentar reconectarse con otro *Gateway* durante un cierto tiempo configurable. Cuando se logra la reconexión, comenzará a enviarle nuevamente los datos de referencia y los datos a procesar, al igual que en la primera conexión.

6.5. Tolerancia a fallos en Controller

Como se explicó durante el informe, la responsabilidad principal del *Controller* es el conteo de mensajes para manejar correctamente el fin de transmisión de cada cliente. Para lograr mantener esta información entre caídas de los nodos, decidimos persistirla en un archivo de la manera que será explicada a continuación.

Cada vez que un cliente se conecta, el *Controller* creará un archivo donde se va a persistir esta información. Luego, cada vez que le llegue un *MessageCounter* para ese cliente, la información necesaria de ese mensaje será persistida en dicho archivo, y posteriormente a esa escritura se enviará el ACK del

MessageCounter. Esto nos permite mantener persistida la información del conteo de mensajes para cada cliente y que los *MessageCounter* que no lleguen a ser persistidos sean re-encolados en la *counter_exchange* (la cual también usa colas persistentes) para que sean procesados cuando el *Controller* se reinicie.

Al levantarse nuevamente el *Controller*, lo primero que hará es leer estos archivos (si es que tiene alguno). De esta forma, podrá continuar el procesamiento desde el lugar en donde estaba previo a la caída. En el caso de que alguno de los clientes haya terminado y el *Controller* se haya caído justo antes de enviar el mensaje de fin de transmisión a los workers *Aggregator*, se detectará esto al terminar de procesar el archivo para ese cliente, y el mensaje será enviado como lo haría normalmente. Finalmente, una vez enviado dicho mensaje, se elimina el archivo para ese cliente.

En el caso de que el *Controller* se caiga luego de mandar el mensaje pero antes de eliminar el archivo, cuando sea reiniciado volverá a procesar ese archivo y detectará que debe enviarle el mensaje de fin de transmisión a los workers *Aggregator*. Esto no es problema porque, como explicamos antes, el mensaje de fin de transmisión es idempotente y además todos los workers descartan los mensajes duplicados.

6.6. Algoritmo de elección de líder

En un principio, nuestra intención era utilizar un mecanismo de elección de líder para el manejo de las réplicas en los nodos críticos de nuestro sistema (*Gateway*, *Controller* y *Aggregator*). Sin embargo, posteriormente a llevar a cabo esa implementación, notamos que nuestro sistema no lo requería y funcionaba correctamente aún sin tenerlo (tomando en cuenta todas las consideraciones que explicamos a lo largo de la sección).

De todas formas, este módulo es parte de nuestro sistema y consideramos relevante dedicarle una sección en el informe.

El algoritmo implementado es el de Bully, para el cual utilizamos colas de RabbitMQ para realizar el envío de mensajes entre los nodos involucrados en la elección de líder.

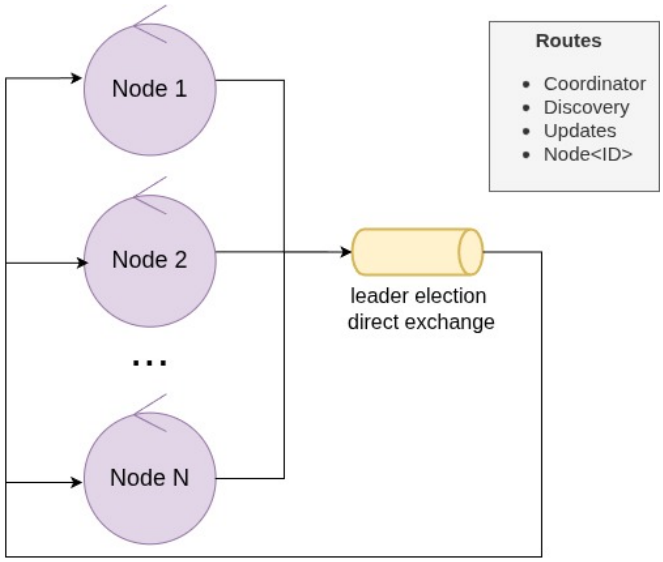


Figura 22: Visión general de la interacción entre nodos y la transmisión de información mediante colas de mensajes.

Como vemos en el diagrama anterior, todos los nodos se van a comunicar utilizando un *direct exchange*, cuyas posibles routing keys son *Coordinator* (para enviar el mensaje de Coordinator a todos los nodos), *Discovery* (utilizado en la fase de descubrimiento del líder), *Updates* (para el envío de actualizaciones a los nuevos nodos) y *NodeId* (utilizado para comunicarse exclusivamente con el nodo con esa ID).

6.6.1. Fase de descubrimiento del líder

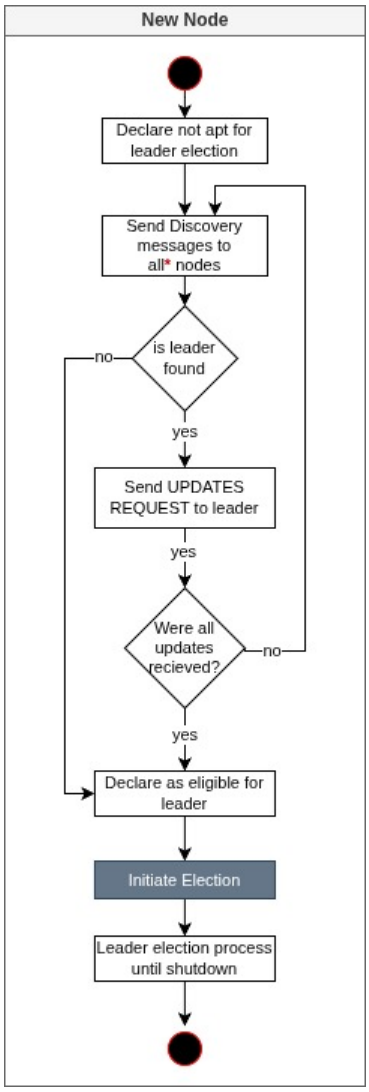


Figura 23: Actividades principales en la conexión de un nuevo nodo.

Cuando un nuevo nodo se conecta al sistema (ya sea la primera vez o una reconexión), lo primero que hace es declararse no apto para ser elegido como líder en una posible elección. Esto es porque dicho nodo acaba de conectarse y no tiene la información necesaria para empezar a actuar como líder en caso de ser elegido.

Luego, usando la *lider_election_exchange* con la routing key *Discovery*, envía el mensaje de *DISCOVER* a todos los nodos del sistema. Cabe mencionar que, en este sistema, cada nodo conoce cuántos nodos hay en el sistema como máximo (estén conectados o no), y esa cantidad no será modificada.

Luego, el resto de nodos responderán a este mensaje de *DISCOVER* indicando su propia ID y el ID del líder (si es que tienen esa información). Si al nuevo nodo le llega alguna respuesta indicando el ID del líder actual, entonces le envía una solicitud de actualización usando la *lider_election_exchange* con el ID del nodo líder como routing key.

De esta forma, el líder comenzará a enviarle toda la información actualizada que necesita para estar al día con el resto de nodos. Una vez que obtiene todos estos datos, puede declararse como elegible para una próxima elección de líder. En este punto, si el ID del nuevo nodo es mayor al ID del líder que descubrió, inicia una nueva elección de líder. Caso contrario, se quedará realizando el procesamiento que veremos en la siguiente sección.

Si luego de un tiempo de espera determinado nunca se terminan de obtener todas las actualizaciones, volvemos a iniciar la etapa de descubrimiento y se repite el procedimiento anterior.

Por otro lado, si no se encuentra el líder (luego de un tiempo de espera determinado), el nuevo nodo asume que no hay ningún líder conectado, se declara como elegible para una próxima elección de líder y la inicia.

6.6.2. Fase de elección del líder

Los nodos involucrados en el sistema (que no sean el nodo líder) están continuamente haciendo dos cosas: registrando los heartbeats que le llegan del líder y esperando un posible mensaje de elección de líder.

Tanto en el caso de que se detecte que el líder se cayó (porque el nodo dejó de recibir heartbeats) o porque se recibió un mensaje de *ELECTION* desde la *lider_election_exchange*, se inicia el procedimiento de elección de líder.

En este procedimiento, lo primero que se hace es mandarle el mensaje de *ELECTION* por la *lider_election_exchange* a todos los nodos cuya ID sea mayor a la del nodo que inició la elección.

Si este nodo nunca recibe un *ACK*, entonces significa que ninguno de los nodos con ID mayor están conectado. Por lo cual, el nodo que inició la elección se proclama líder y envía el mensaje *COORDINATOR* por la *lider_election_exchange* con la routing key *Coordinator*.

En el caso de recibir el *ACK*, entonces significa que alguno de los nodos con ID mayor está conectado. Por lo cual, el nodo se queda esperando el mensaje de *COORDINATOR*. En el caso de que no le llegue ese mensaje, entonces vuelve a iniciar el procedimiento de elección de líder, dado que no puede proclamarse líder.

En el caso de recibir tanto el *ACK* como el *COORDINATOR*, entonces se define como nuevo líder al nodo que mandó este último mensaje, y el nodo actual vuelve al estado inicial (registrando los heartbeats que le llegan del líder y esperando un posible mensaje de elección de líder).

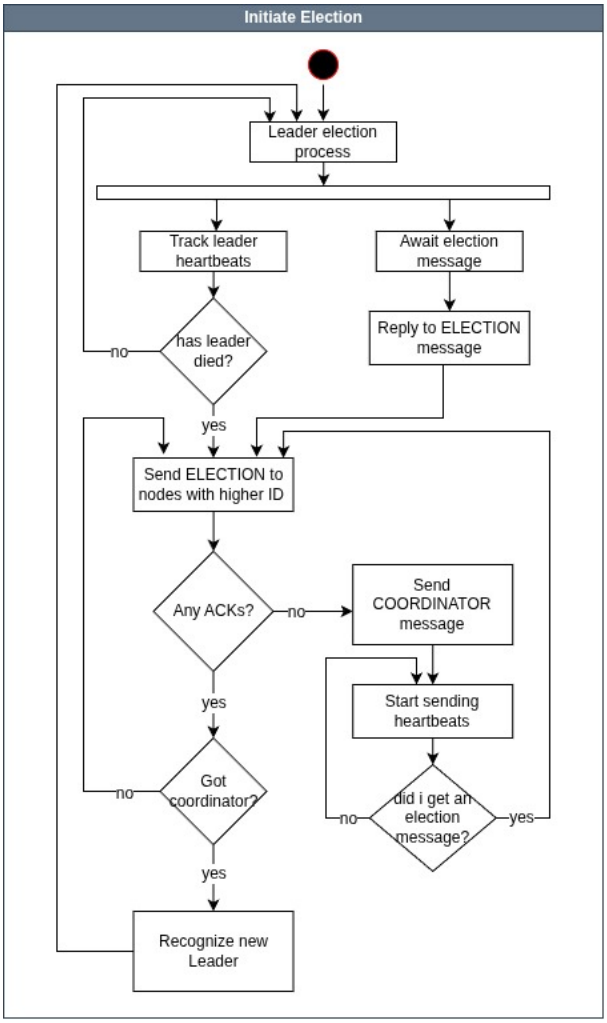


Figura 24: Actividades principales en la conexión de un nuevo nodo.

7 Tareas a ejecutar

7.1. Definición de tareas

Definición y separación de tareas a realizar para el desarrollo del sistema por categoría.

7.1.1. Infraestructura principal

- Definir estructura del proyecto y paquetes de Go.
- Implementación de mensajes y serializado (protobuf, mensajes, manejo de errores).
- Uso de RabbitMQ e implementación de un wrapper.
- Creación de *worker* simple como base para los demás.

7.1.2. Gateway

- Implementación de interfaz de usuario con manejo *uploads* y *responses*.
- Re-envió de *batches* a la cola de los datasets correspondientes.
- Recolectado de datos procesados y envío al cliente.

7.1.3. Workers

- **Filter Worker:** Lógica de filtrado con lectura, y escritura de las colas.
- **GroupBy Worker:** Lógica de agrupado de batches.
- **Reducer Worker:** Lógica de reducción de batches, con los tipos *sum* y *count*.
- **Join Worker:** Lógica de unión de tablas, consumiendo datos de la cola correspondiente al dataset necesario.
- **Aggregator Worker:** Lógica de ordenamiento (Top N), acumulado de datos, confección y envió de resultados.

7.1.4. Middleware

- Configurar imagen de Docker de RabbitMQ.
- Definición de colas necesarias del sistema.

7.1.5. Despliegue y escalado

- Dockerizar cada paquete necesario para desplegar el sistema.
- Configuración de despliegue (numero de workers, etc) y Docker-compose.
- Plan de replicas para gateway y aggregators.

7.1.6. Pruebas

- Pruebas de protocolo y Networking.
- Pruebas punta a punta cliente-servidor.
- Validación con lo esperado del notebook de Kaggle.

7.2. División entre integrantes

División tentativa de tareas generales entre integrantes (**Sujeto a modificaciones**).

Sección	Máximo Utrera	Santiago Sevitz	Federico Genaro
Infraestructura	Estructura Go, mensajes (protobuf), wrapper RabbitMQ	Worker base	–
Gateway	API cliente, uploads/responses, batches → colas, resultados → cliente	–	–
Workers	–	Filter, GroupBy, Reducer	Join, Aggregator
Middleware	Docker de RabbitMQ y definición de colas	–	–
Despliegue & Escalado	–	Dockerización de workers	Configuración y Docker-compose, réplicas gateway/aggregator
Pruebas	Protocolo y networking	–	punta a punta, validación con Kaggle

Cuadro 1: División de tareas entre integrantes

8 Referencias

1. Gerald Ooi. (2025). G Coffee Shop Transaction 202307 to 202506 [Dataset]. Kaggle. `https://www.kaggle.com/datasets/geraldooizx/g-coffee-shop-transaction-202307-to-202506`.

2. Gabriel Robles. (2025). FIUBA - Distribuidos 1 - Coffee Shop Analysis [Notebook]. Kaggle. `https://www.kaggle.com/code/gabrielrobles/fiuba-distribuidos-1-coffee-shop-analysis`.