Documento de Arquitectura - Steam Analysis

Sistemas Distribuídos - Segundo cuatrimestre de 2024

Alumno	**Número de padrón**	**Email**
Masri, Micaela	108814	noahmasri19@gmail.com
Ayala, Camila	107440	cayala@fi.uba.ar
Danko, Tomás	107431	tdanko@fi.uba.ar

Índice

1. Introducción

1.1. Propósito

El presente Documento de Arquitectura de Software presenta la arquitectura del sistema a desarrollar a través de diferentes vistas, cada una de las cuales ilustra un aspecto en particular del mismo. Este sistema tiene como propósito responder ciertas consultas específicas sobre un conjunto de datos estructurados proporcionados por el usuario. Se espera que este documento brinde al lector una visión global y comprensible del diseño general del sistema.

2. Alcance

El sistema tiene como objetivo responder cinco consultas (*queries*) sobre dos conjuntos de datos correlacionados: uno que contiene información sobre juegos, y otro con reseñas de dichos juegos, ambos relacionados con la plataforma Steam. Las consultas a resolver son las siguientes:

Q1. Cantidad de juegos soportados en cada plataforma (Windows, Linux, MAC).

Q2. Nombres de los 10 juegos del género `Indie` publicados en la década del 2010 con mayor tiempo promedio histórico de juego.

Q3. Nombres de los 5 juegos del género `Indie` con mayor cantidad de reseñas positivas.

Q4. Nombres de los juegos del género `Action` con más de 5,000 reseñas negativas en idioma inglés.

Q5. Nombres de los juegos del género `Action` que se encuentren en el percentil 90 en cantidad de reseñas negativas.

Además, el sistema deberá ser capaz de atender múltiples clientes de manera concurrente.

3. Arquitectura de Software

El sistema seguirá una arquitectura cliente-servidor, donde el cliente se conecta al servidor a través de una cola de ZeroMQ (`ZMQ`), utilizando el patrón de colas `Dealer-Router`. Este patrón permite implementar un modelo de *request-response* más escalable, ya que garantiza que el servidor pueda identificar qué cliente envió un mensaje. En este esquema:

- Antes de cada mensaje, el servidor recibe un primer paquete con el identificador del cliente, seguido por la información enviada.
- El cliente envía un conjunto de datos sobre los que se deben ejecutar las cinco consultas, y el servidor devuelve las respuestas a dichas consultas, finalizando posteriormente la comunicación.

El cliente envía los datos al servidor en lotes (*batches*), cada uno de los cuales puede contener múltiples filas de un conjunto de datos. Para simular la comunicación en modo *streaming*, el servidor procesa y responde a las consultas a medida que recibe los datos.

En la comunicación interna del servidor se emplean diversas colas de RabbitMQ. Cada cola está asignada a un tipo específico de nodo, que procesa los paquetes que representan tareas.

4. Objetivos y Limitaciones Arquitectónicas

Esta sección define los principales objetivos que la arquitectura debe cumplir, junto con las limitaciones que condicionarán su diseño y desarrollo. Los puntos clave son:

- **Optimización para entornos multicomputadora:** El sistema debe estar diseñado para aprovechar al máximo los entornos distribuidos, mejorando el rendimiento en arquitecturas con múltiples nodos de procesamiento.
- **Escalabilidad:** La arquitectura debe permitir el escalamiento horizontal mediante la adición de recursos de cómputo, para gestionar un aumento en los volúmenes de datos.
- **Desarrollo de Middleware:** Se requiere implementar un middleware que abstraiga la comunicación entre nodos. Este middleware debe simplificar la interacción en grupos, reduciendo la complejidad operativa.
- **Ejecución única y manejo de señales:** El sistema debe garantizar la ejecución única de procesos y manejar de manera eficiente la terminación controlada (*graceful quit*) al recibir señales de finalización, como `SIGTERM`.

5. Vista de casos de uso

El presente sistema está diseñado para un propósito único: responder cinco consultas específicas sobre dos conjuntos de datos enviados por el cliente. En la figura 1, se presenta un diagrama de casos de uso que modela esta funcionalidad, donde las consultas están incluidas en el caso de uso principal "Realizar consulta sobre datos". Esto se debe a que todas las consultas se procesan conjuntamente al enviarse los datasets.

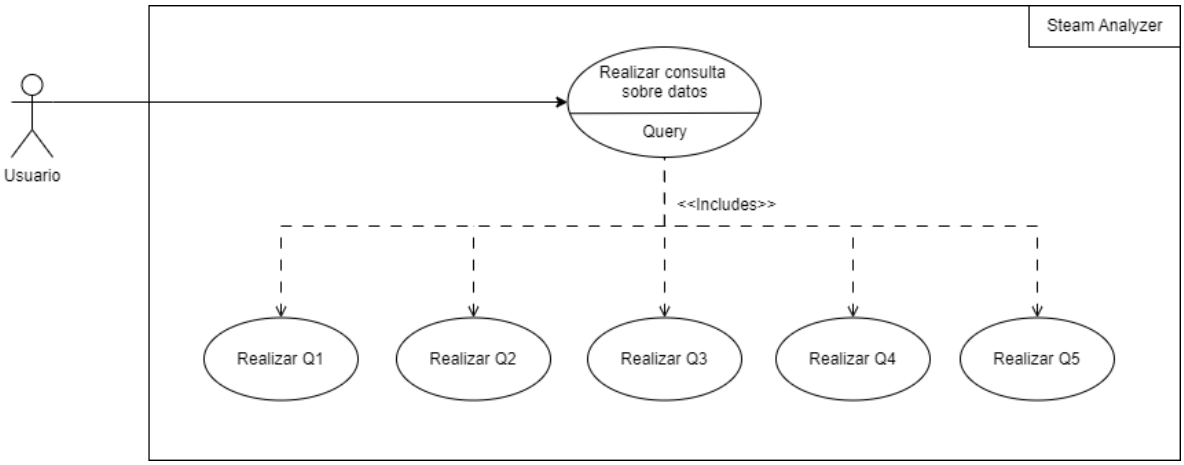

Figura 1: Diagrama de casos de uso

Realizar consulta sobre los datos - Caso de uso base

Nombre del caso de uso: Realizar consulta sobre los datos.

Resumen: El cliente realiza consultas sobre un conjunto de datos enviado.

Dependencia: Incluye realizar consulta Q1, Q2, Q3, Q4 y Q5.

Actor: Cliente.

Secuencia principal:

1. El cliente se conecta al servidor.
2. El cliente envía los datos sobre los cuales realizar las consultas.
3. El servidor procesa las respuestas a las consultas.
4. Se incluyen los casos de uso de realizar consulta Q1, Q2, Q3, Q4 y Q5.

5. El cliente se desconecta.

Realizar consulta Q1 - Caso de uso incluido

Nombre del caso de uso: Realizar consulta Q1.

Resumen: El cliente recibe la respuesta a la consulta 1.

Actor: Cliente.

Precondición: El cliente ha enviado los datos, pero no ha recibido la respuesta a la consulta 1.

Descripción del segmento de inserción:

1. El servidor envía la cantidad de juegos soportados en Windows, Linux y Mac.
2. El cliente recibe e interpreta los tres números de respuesta.

Realizar consulta Q2 - Caso de uso incluido

Nombre del caso de uso: Realizar consulta Q2.

Resumen: El cliente recibe la respuesta a la consulta 2.

Actor: Cliente.

Precondición: El cliente ha enviado los datos, pero no ha recibido la respuesta a la consulta 2.

Descripción del segmento de inserción:

1. El servidor envía hasta 10 nombres de juegos que cumplen con las características solicitadas.
2. El cliente recibe la cantidad de juegos (de 0 a 10) y sus nombres.

Realizar consulta Q3 - Caso de uso incluido

Nombre del caso de uso: Realizar consulta Q3.

Resumen: El cliente recibe la respuesta a la consulta 3.

Actor: Cliente.

Precondición: El cliente ha enviado los datos, pero no ha recibido la respuesta a la consulta

3.

Descripción del segmento de inserción:

1. El servidor envía hasta 5 nombres de juegos que cumplen con las características solicitadas.
2. El cliente recibe la cantidad de juegos (de 0 a 5) y sus nombres.

Realizar consulta Q4 - Caso de uso incluido

Nombre del caso de uso: Realizar consulta Q4.

Resumen: El cliente recibe la respuesta a la consulta 4.

Actor: Cliente.

Precondición: El cliente ha enviado los datos, pero no ha recibido la respuesta a la consulta 4.

Descripción del segmento de inserción:

1. El servidor envía los nombres de los juegos que cumplen con las características, mientras los procesa.
2. El cliente recibe el streaming de los nombres de los juegos.
3. El servidor anuncia la finalización del streaming.

Realizar consulta Q5 - Caso de uso incluido

Nombre del caso de uso: Realizar consulta Q5.

Resumen: El cliente recibe la respuesta a la consulta 5.

Actor: Cliente.

Precondición: El cliente ha enviado los datos, pero no ha recibido la respuesta a la consulta 5.

Descripción del segmento de inserción:

1. El servidor envía la cantidad y los nombres de los juegos que cumplen con las características.
2. El cliente recibe la cantidad y los nombres de los juegos.

6. Ciclo de vida de un cliente en el servidor

Para detallar el flujo de procesamiento de datos enviados por un cliente, se ha modelado un DAG (*Directed Acyclic Graph*), como se observa en la figura 2. Este diagrama ilustra el procesamiento de datos desde su recepción en el servidor de borde hasta su bifurcación para el tratamiento de datasets de juegos y reseñas.

Figura 2: DAG del sistema en su totalidad

6.1. Query 1

Para esta query, representada en la figura 3, únicamente son necesarios los datos de la tabla de juegos. El procesamiento es bastante lineal, similar a un patrón de pipe and filters.

Una vez que el boundary recibe un cliente, este se pasa al Initializer. Este componente se encarga de filtrar las columnas que no son necesarias para este caso particular, es decir, todo lo que no sea la información sobre si el juego en cuestión posee soporte para Windows, Linux y Mac. Una vez realizado el filtrado, los batches a procesar son enviados a los Groupers. Estos se ocupan de obtener cuentas parciales a partir de cada batch, devolviendo:

- La cantidad de juegos en dicho batch que soportan cada sistema operativo (SO).
- La cantidad total de juegos en ese batch.

Luego del agrupamiento, los datos son enviados al Aggregator OS. Este componente se encarga de unificar las cuentas parciales obtenidas por los Groupers en una única entrada que contiene los resultados para el conjunto de datos completo.

Finalmente, dicha información es enviada al Response Dispatcher. Este último componente se encarga de enviar los resultados en el formato correspondiente y al cliente en cuestión. Todas las entidades previamente mencionadas están explicadas en detalle en la sección 8.

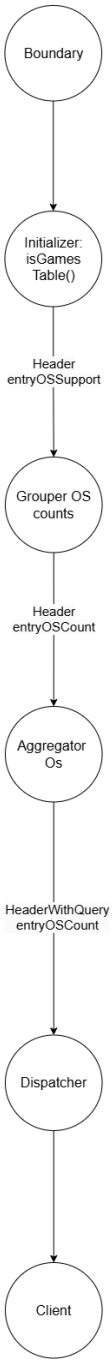

Figura 3: DAG de la query 1

6.2. Query 2

Para la query 2, representada en la figura 4, únicamente es necesaria la tabla de juegos, por lo que su procesamiento también es bastante lineal.

Al llegar data de un cliente al boundary, esta se pasa al Initializer. Este componente se encarga de filtrar cada entrada de la tabla para mantener únicamente las columnas referentes al ID, el nombre, los géneros del juego, su fecha de salida y la cantidad promedio de horas jugadas.

Luego, la información es enviada a los Filterers, que se encargan de filtrar los juegos a partir de la columna 'genres', enviando únicamente aquellos que forman parte de la categoría 'indie'.

A continuación, estas respuestas son enviadas a otros Filterers donde se aplica un nuevo filtro, esta vez basado en la fecha de salida de cada juego. En este caso, se dejan solo aquellos juegos que hayan sido lanzados en la década del 2010. La decisión de separar estos dos filtros está relacionada con la posibilidad de ahorrar procesamiento, que también es requerido por la Query 3, que necesita el filtrado inicial por el género 'indie'.

Posterior a los filtrados, la información se shardea en función del ID de juego y se envía a los Sorters. Estos ordenan los juegos que les corresponden y obtienen los Top 10 parciales (es decir, para ese shardeo) en función del tiempo histórico de juego.

Una vez realizado el ordenamiento, los resultados se envían a un Sorter Consolidator, que espera recibir los Top 10 calculados por cada Sorter y obtiene el Top 10 total. Finalmente, este resultado final es enviado al Dispatcher, quien lo manda al cliente correspondiente.

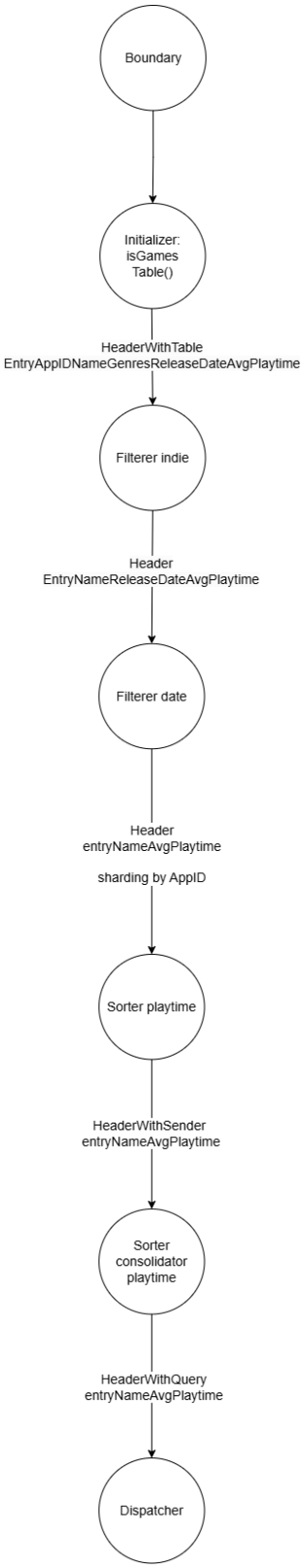

Figura 4: DAG de la query 2

6.3. Query 3

En el caso de la Query 3, representada en la figura 5, se requiere de ambas tablas para su procesamiento.

En cuanto a la tabla de juegos, el procesamiento es el mismo hasta el filtrado de juegos del tipo 'indie', por lo que se reutiliza dicha información.

Respecto a la tabla de reseñas, el procesamiento comienza por el Initializer, que se encarga de separar las reseñas positivas (de score 1) de las negativas (de score -1) y deshacerse de las columnas innecesarias. En este caso, nos quedamos con la columna App ID de las reseñas positivas. Estos batches van a ser posteriormente enviados a los Groupers, que van a .ªplastar"las filas en función de los ID para contar la cantidad de reseñas positivas que posee cada juego en cada batch.

Una vez hecho esto, los batches correspondientes a ambas tablas son enviados a los Joiners con el objetivo de asignar a cada reseña la información correspondiente a su juego. Todo esto es posible gracias al shardeo por ID del juego, que asigna a cada Joiner los juegos de los que debe ocuparse.

Posterior a dicha unión, la información es enviada a un Aggregator, cuya función es la de unificar la metadata y los números de secuencia.

Luego de esto, la información vuelve a ser shardeada para ser enviada a los Sorters, que se encargan de ordenar los juegos que les son asignados a cada uno para obtener el Top 5 parcial.

Finalmente, estos Top 5 parciales se envían al Sorter Consolidator, que toma cada uno de los Top 5 calculados por cada Sorter para calcular el Top 5 total y enviarlo al Dispatcher.

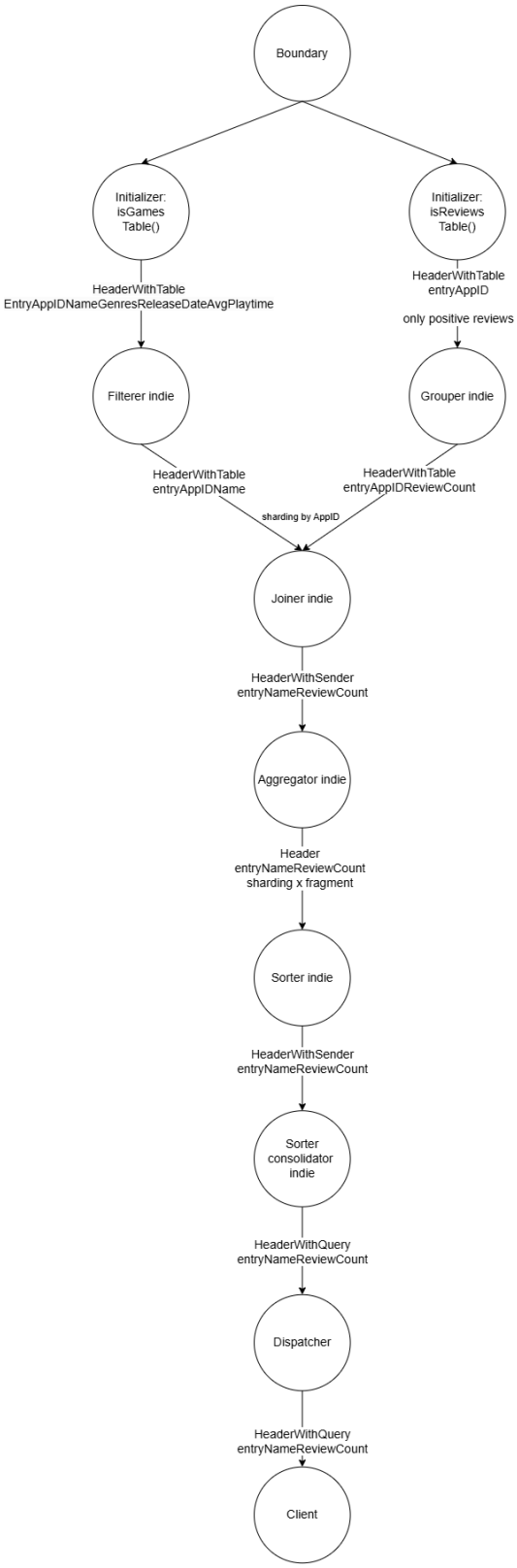

Figura 5: DAG de la query 3

6.4. Query 4

Para la query 4, presentada en la figura 6, se requiere de ambas tablas. Del lado izquierdo, observamos el procesamiento de la tabla de juegos, empezando como siempre por el filtrado de columnas innecesarias por parte del Initializer, y luego pasando dicha tabla por Filterers con el objetivo de deshacernos de aquellas entradas de juegos que no pertenecen al género 'action'.

Por otro lado, a la derecha vemos el procesamiento de la tabla de reseñas, donde el Initializer separa dichas reseñas según si el puntaje es positivo o negativo y filtra las columnas innecesarias. Para esta query en particular, necesitamos sólo las entradas correspondientes a reseñas negativas, y como columnas sólo requerimos del ID del juego en cuestión (para poder unir dicha reseña con el juego correspondiente) y del texto de la misma (para poder filtrarla según el lenguaje en el que está escrita).

Una vez hecho esto, los batches correspondientes a ambas tablas son enviados a los Joiners con el objetivo de asignar a cada reseña la información correspondiente a su juego (todo esto siendo posible a partir del shardeo por ID del juego, que asigna a cada Joiner los juegos de los que debe ocuparse).

Posteriormente, cada una de estas filas es enviada a los Filterers, donde filtramos aquellas reseñas escritas en el idioma inglés, y luego pasan a los Groupers, que calculan resultados parciales para cada batch en los que se incluyen la cantidad de reseñas negativas que posee cada juego para dicho batch.

La información es luego enviada a un Aggregator, que se encarga de recopilar la información de todos los Joiners e ir enviando al cliente a medida que van llegando, los juegos que cuentan con 5.000 reseñas. Una vez que el conteo de alguno de los juegos alcanza las 5.000 reseñas, se deja de acumular la información para dicho juego y se streamea la información correspondiente a dicho juego.

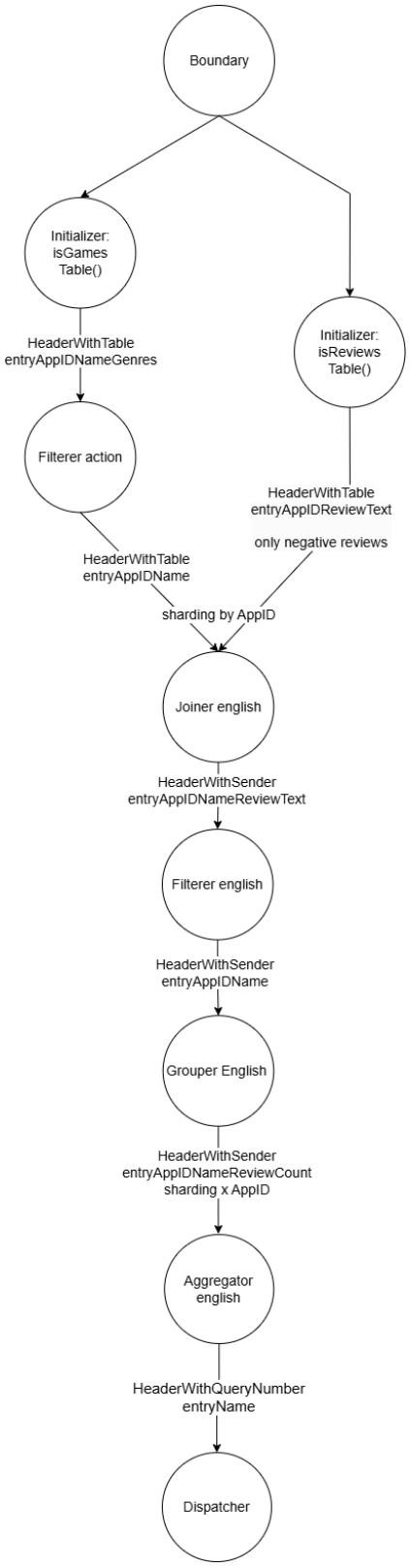

Figura 6: DAG de la query 4

6.5. Query 5

La query diagramada en la figura 7 es igual a la 4 del lado de los juegos, de forma que los Filterers correspondientes a los juegos del género 'action' envían a los Joiners siguientes la misma información que envían a los Joiners iniciales de la Query 4.

En cuanto a las reseñas, en este caso separamos las reseñas negativas al igual que como en la Query 4, pero en este caso sólo requerimos del ID, para posteriormente enviarlas a los Groupers que se ocupan de 'aplastar' la cantidad de filas por ID que contiene cada batch.

Una vez hecho esto, los batches correspondientes a ambas tablas son enviados a los Joiners con el objetivo de asignar a cada conteo de reseñas la información correspondiente a su juego, similar a las queries anteriores.

Luego de esto, la información pasa a enviarse a un Sorter Consolidator, que espera hasta recibir todos los batches para determinar la cantidad total de reseñas negativas por cada juego, y posteriormente calcular el percentil.

Una vez calculado el percentil manteniendo las entradas ordenadas por cantidad de reseñas, se envía al Dispatcher sólo aquellos que forman parte del percentil 90 (es decir, el 10 % de los juegos con mayor cantidad de reseñas negativas).

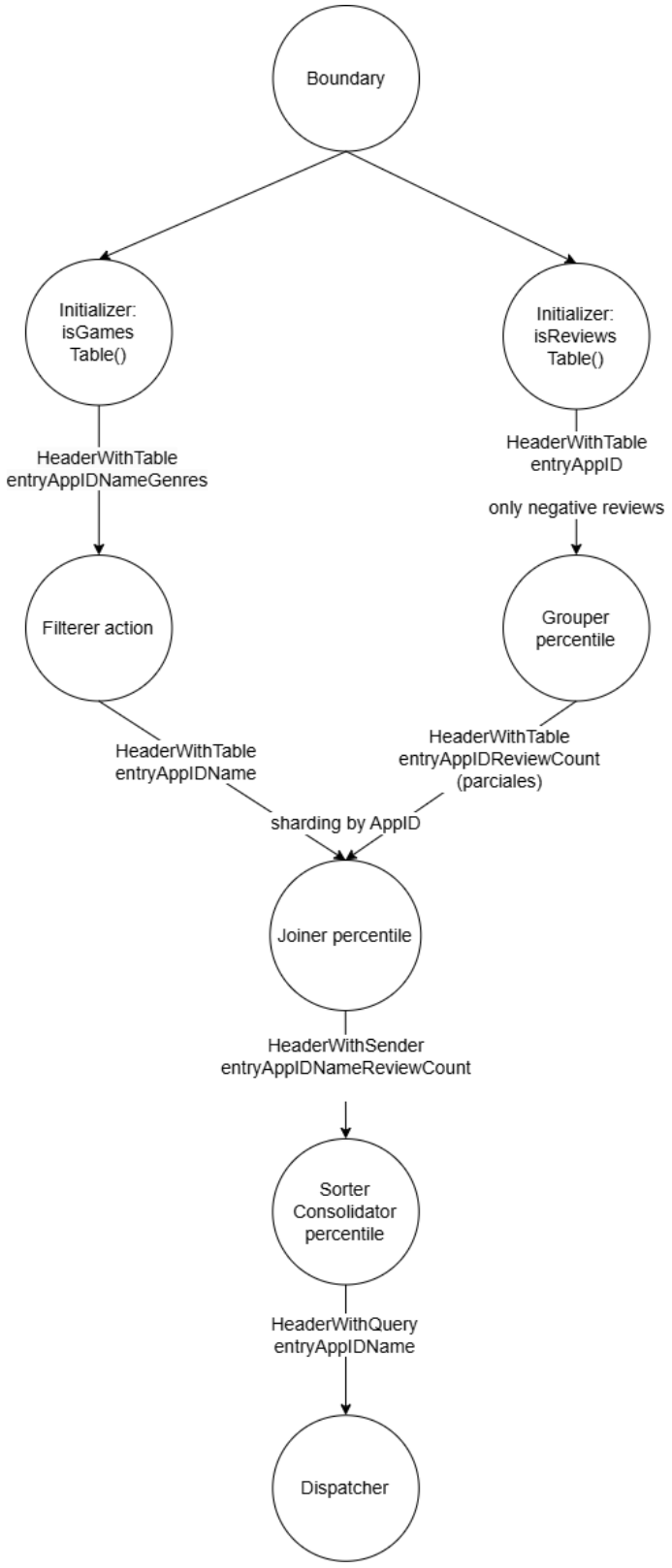

Figura 7: DAG de la query 5

7. Vista de procesos

En la figura 8 se ilustra el comportamiento de las distintas entidades en tiempo de ejecución en un diagrama de actividades. Para cada entidad se muestra como reacciona a los estímulos provocados por las demás.

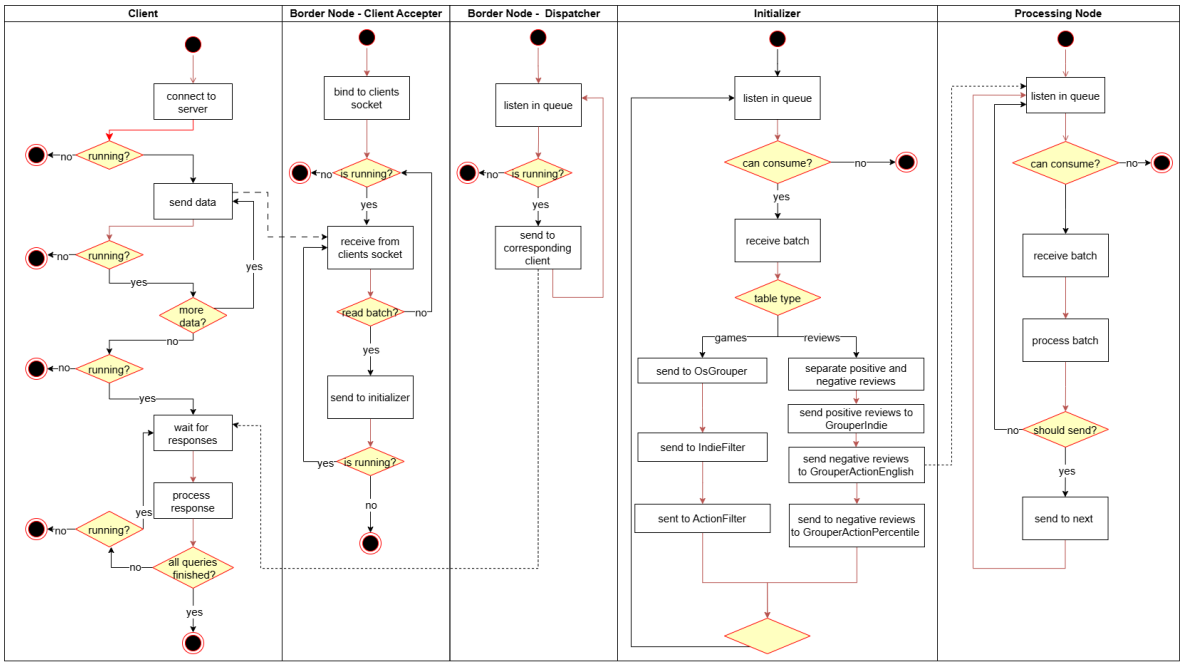

Figura 8: Diagrama de actividades

El Client Accepter y el Response Dispatcher se encuentran en el mismo nodo de ejecución, siendo este el Border Node, por lo cual se comunican mediante memoria compartida. El resto de las entidades se encuentran en distintos nodos y se comunican mediante la red. Este servidor unicamente finaliza cuando se recibe una señal de SIGTERM. El cliente por otro lado, puede terminar naturalmente cuando recibe los resultados de todas sus consultas, o terminar debido a la recepción de un mensaje inesperado del servidor, que implica que o bien este está fallado, o el servidor lo está. Cabe destacar que no se diferencio entre los distintos tipos de procesamiento ya que su ciclo de vida es similar, y se los englobó en la abstracción Processing Node para explicar su comportamiento. A su vez, el camino mostrado en este diagrama es el camino "feliz", en donde el cliente se conecta y manda la información correctamente. Se obvio la fase de handshake, en donde se le asigna un ID al cliente, al igual que el manejo de la caída de un

cliente. Esto será explicado en la sección posterior 10.

Se realizaron a su vez diagramas de secuencia para explicar el intercambio de mensajes entre entidades. Para mostrar el control de los datos por parte de cada entidad en mejor detalle, se optó por realizar diagramas de secuencia correspondientes a cada una de las queries que el sistema se encargará de realizar.

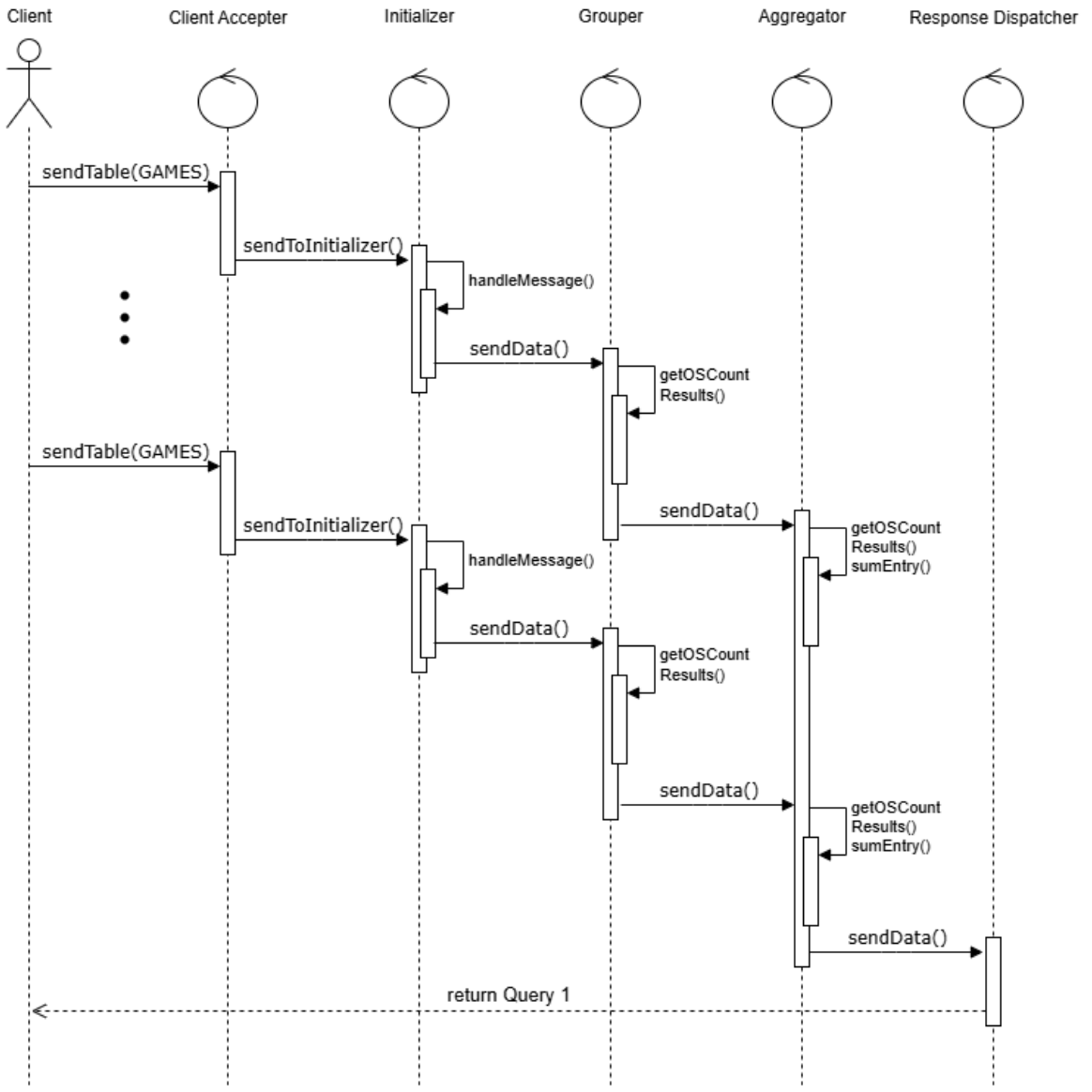

Figura 9: Diagrama de secuencia para la query 1

Para la query 1, el cliente envía la información en varios batches de datos, que van a través

del Client Accepter al nodo inicializador. Dicho nodo se encargará de filtrar las columnas necesarias para dicha query. Luego, el control de los datos de juegos pasará al Grouper, donde se realizará el conteo por batch de los juegos en función de los sistemas operativos soportados. El Aggregator esperará a recibir todos los batches que corresponden a la tabla de juegos para acumular los resultados, y una vez terminado enviará los datos al Response Dispatcher quien devolverá la información al cliente.

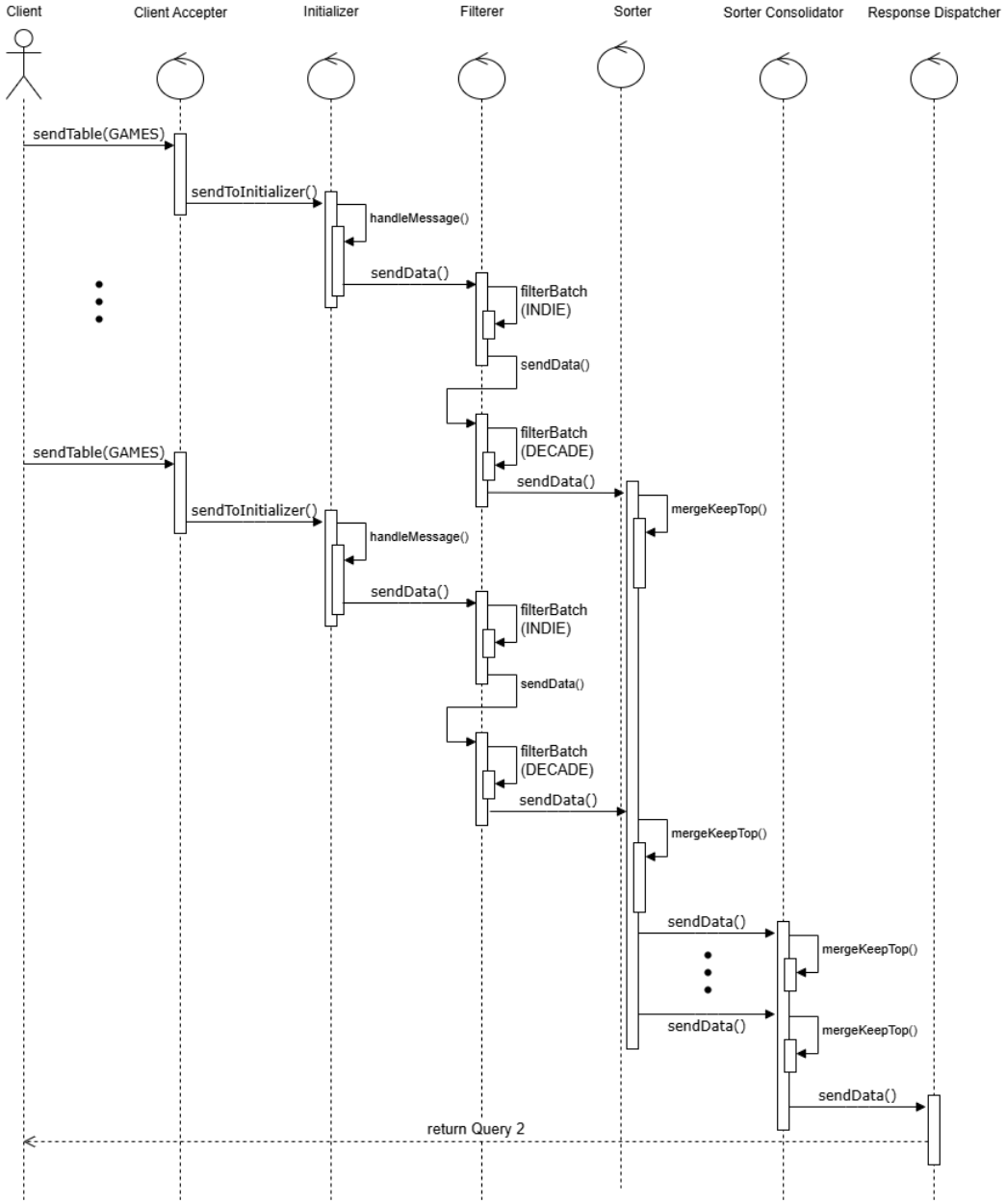

Figura 10: Diagrama de secuencia para la query 2

Para la query 2, el cliente envía la información en varios batches de datos, que van a través del Client Accepter al nodo inicializador. Dicho nodo se encargará de filtrar las columnas necesarias para dicha query. Luego, el control de los datos de juegos pasará a un Filterer, que en principio filtrará los juegos del género “Indie”, y luego reenviará la información nuevamente a otro Filterer (que podría ser o no el mismo nodo) para filtrar los juegos salidos en la década de los 2010’s. Los Sorters recibirán la información shardeada por número de fragmento para ocuparse de calcular el top 10 parcial de juegos con mayor tiempo histórico de juego. Al manejar durante todo el sistema un número de fragmento para cada paquete, y teniendo a su vez un ID para cada nodo de una capa, se hace un reparto de estos de forma ’fair’, es decir, dando a cada nodo los fragmentos que sean divisibles por su ID. Una vez calculado este top 10 parcial, enviarán dichos resultados al Sorter Consolidator, quien debe determinar el top 10 definitivo. Una vez terminado, enviará los datos al Response Dispatcher quien devolverá la información al cliente.

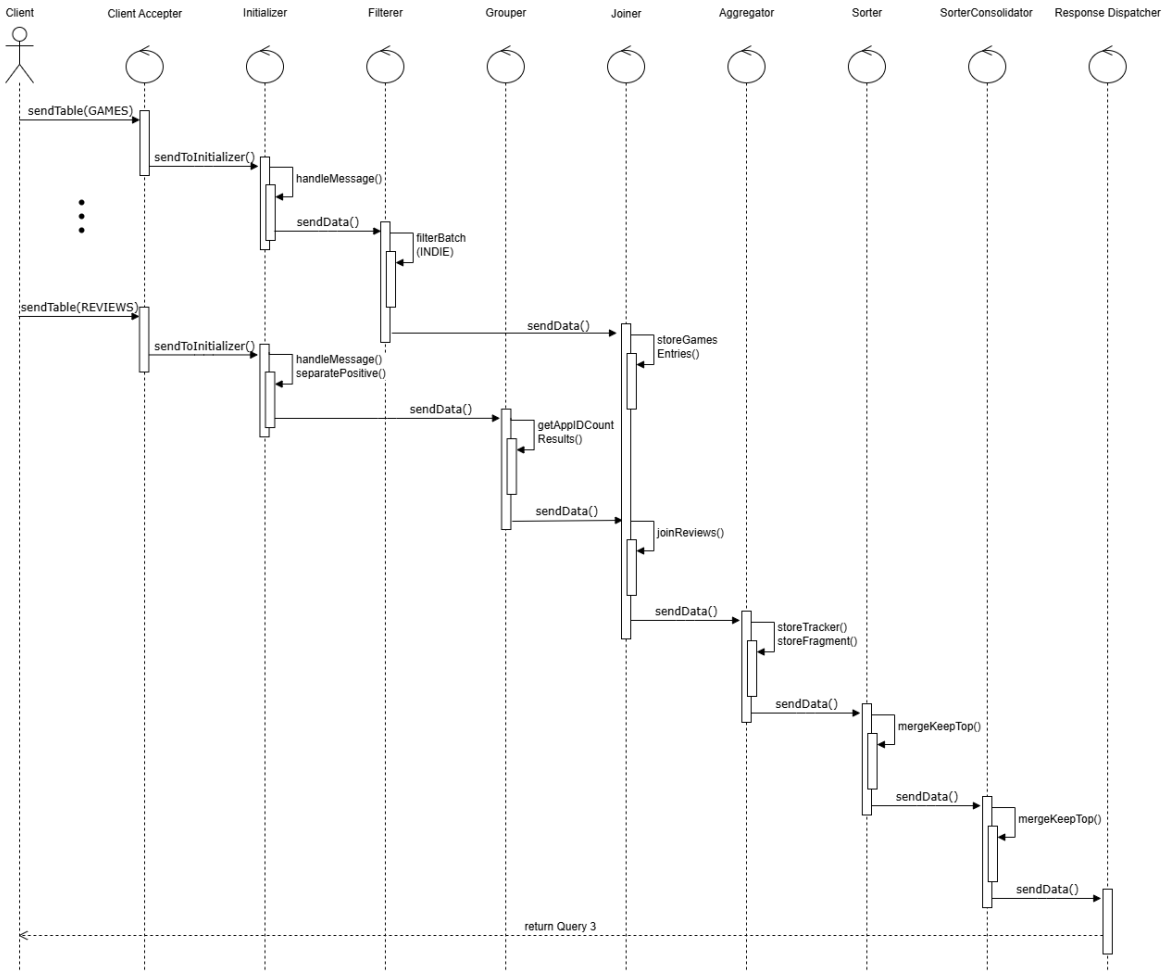

Figura 11: Diagrama de secuencia para la query 3

Para la query 3, al igual que para el resto, el cliente envía la información en varios batches de datos, que van a través del Client Accepter al nodo inicializador. Dicho nodo se encargará de filtrar las columnas necesarias para dicha query. En función de si el batch se trata de información correspondiente a los juegos o a las reviews, recorrerá un distinto camino de nodos:

- si el batch es de juegos, el control del mismo pasará al Filterer para filtrar los juegos del género “Indie” (siendo la misma situación hasta aquí que para la segunda query), y luego se enviarán a los Joiners para ser almacenados hasta la llegada de los batches de reviews, shardeados por ID
- si el batch es de reviews, el control del mismo pasará al Grouper, donde se contarán la

cantidad de reseñas positivas por ID de aplicación en un único batch, para luego pasar también a los Joiners junto con las batches de juegos

En este punto, los Joiners unen cada batch de reseñas recibido con sus juegos correspondientes, y luego envían sus resultados correspondientes al Aggregator con el fin de unificar la metadata y que sea posible shardearla nuevamente, esta vez bajo un nuevo criterio. Una vez recibido todo, el control de los datos pasa a los Sorters, que a cada batch recibido actualizarán el top 5 parcial con los juegos con mayor cantidad de reseñas positivas. Estos resultados parciales se envían al Sorter Consolidator, que se ocupa de calcular el top 5 final. Al finalizar, la información pasa al Response Dispatcher, quien la entregará al cliente.

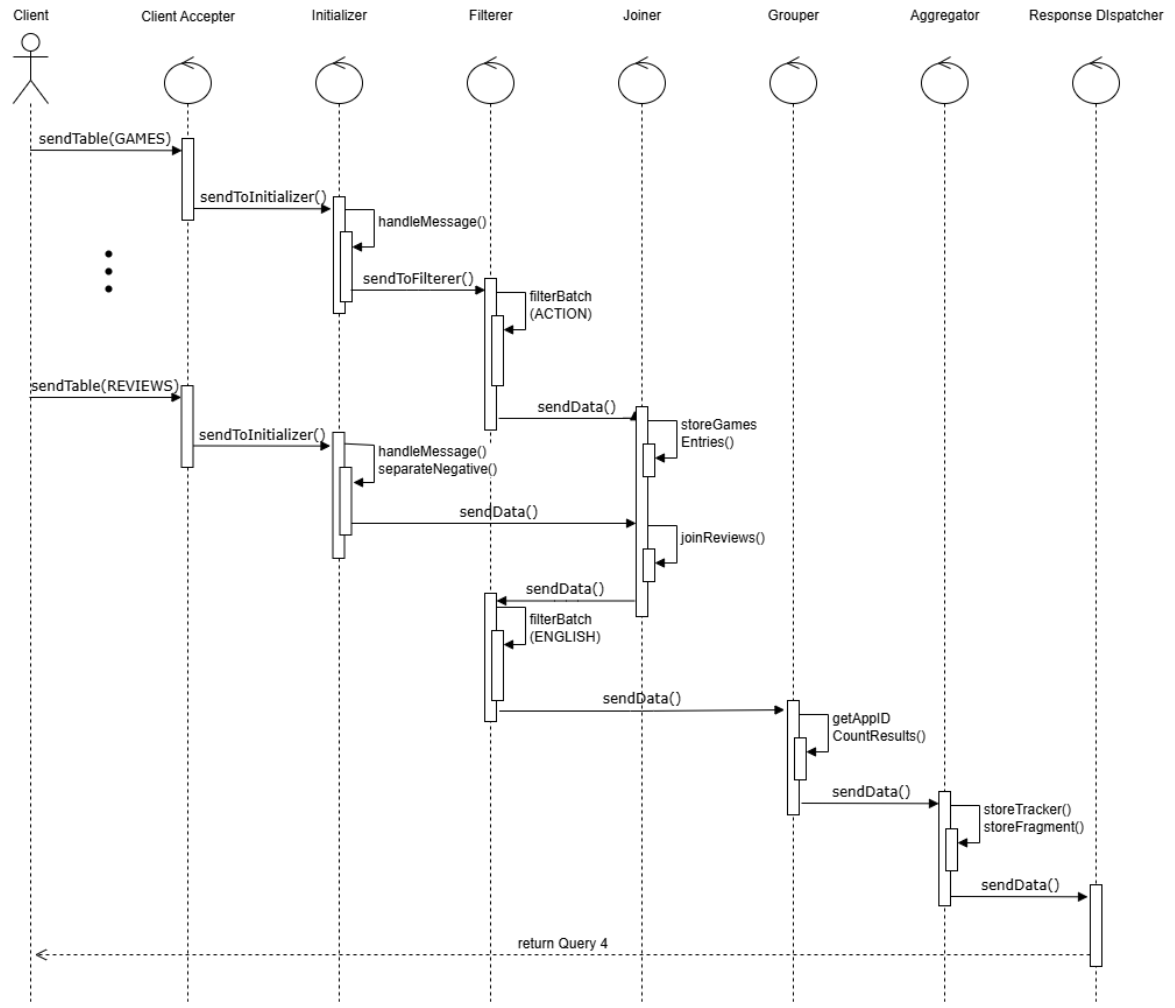

Figura 12: Diagrama de secuencia para la query 4

Para la query 4, al igual que para query 3 en función de si el batch se trata de información correspondiente a los juegos o a las reviews, recorrerá un distinto camino de nodos:

- si el batch es de juegos, el control del mismo pasará al Filterer para quedarse únicamente con los juegos del género "Action", y luego se enviarán a los Joiners para ser almacenados hasta la llegada de los batches de reviews
- si el batch es de reviews, el control del mismo pasará directamente a los Joiners, shardeándose por ID

En este punto, los Joiners unen cada batch de reviews recibido junto con la información almacenada de los juegos. Luego, cada batch resultante es enviado al Filterer, donde nos deshacemos de aquellas filas que no posean reviews en inglés, y el resto son enviadas al Grouper, donde cada batch se "aplasta" indicando la cantidad de reviews en inglés por juego (obviamente, para ese batch en particular). Luego, la información es enviada al Aggregator, quien envía los juegos a medida que cumplen con la condición de tener más de 5.000 reviews en inglés. Estos resultado se envían al Response Dispatcher, quien a medida que le llegan, va devolviendo la información al cliente.

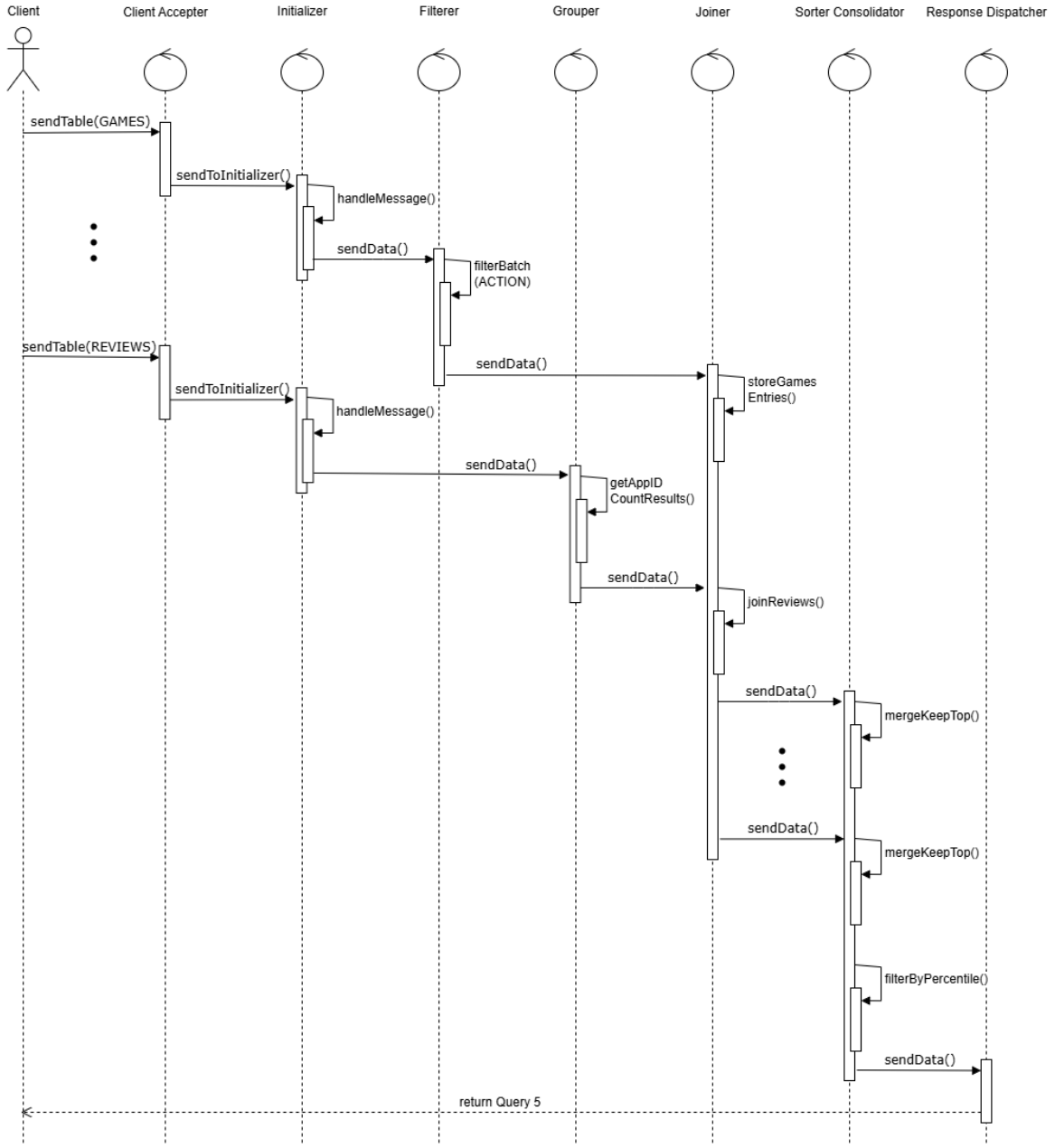

Figura 13: Diagrama de secuencia para la query 5

Para la query 5, en función de si el batch se trata de información correspondiente a los juegos o a las reviews, recorrerá un distinto camino de nodos:

- si el batch es de juegos, el control del mismo pasará al Filterer para filtrar los juegos del género “Action”, y luego se enviarán a los Joiners para ser almacenados hasta la llegada

de los batches de reviews (como ocurre en la cuarta query)

- si el batch es de reviews, el control del mismo pasará al Grouper, donde se contarán las filas con reseñas negativas por ID de aplicación, para finalmente pasar también a los Joiners junto con las batches de juegos

En este punto, los Joiners unen cada batch de reviews recibido junto con la información almacenada de los juegos. Una vez recibido todo, el control de los datos pasa al Sorter Consolidator, cuya función es la de esperar a recibir el procesado por parte de todos los Joiners para luego poder determinar aquellos datos que corresponden al percentil 90. Una vez terminado, solo resta enviar la información obtenida al Response Dispatcher para hacérsela llegar al cliente.

8. Vista de desarrollo

En el diseño del sistema, se han identificado diversas entidades, cada una con responsabilidades específicas que contribuyen al procesamiento y análisis de los datos. Estas entidades operan de manera colaborativa para asegurar un flujo eficiente de información desde la recepción inicial hasta la generación de resultados finales. A continuación, se describen las funciones y características de cada nodo, resaltando su papel en el manejo de los datos de juegos y reseñas. Los nodos que encontramos en el sistema son:

- Initializer: una entidad que se encarga de la distribución inicial de todos los paquetes. Este multiplexa los paquetes al lado correspondiente, dependiendo de si la data es de juegos o de reseñas, separando también por si la reseña es positiva o negativa. Envía a cada nodo subsecuente únicamente las columnas que necesita para su procesamiento. Para la tabla de reseñas se encarga de la separación de las entradas entre las que son positivas y negativas, enviándolas al lugar correcto. Este tipo de nodo no tiene estado ni restricciones de unicidad.
- Filterer: se encarga del filtrado de las entradas a nivel fila. Es el encargado por ejemplo, de solamente dejar los juegos que pertenecen al género indie, como también de filtrar las entradas que no estén en inglés. Encontramos 4 tipos especificos de este nodo: reseñas que estén en ingles, juegos de la década del 2010, juegos de género “indie” y juegos de género “action”. Ninguna de estas variantes tiene estado ni restricciones de unicidad.
- Grouper: se encarga de agrupar entradas haciendo un count como operación de agrega-

ción. Encontramos 3 variantes de estos nodos: grouper de sistemas operativos soportados, grouper de reseñas por App ID y grouper de entradas ya joineadas. Ninguna de las variantes de este tipo de nodo no tiene estado ni restricciones de unicidad.

- Joiner: Se encarga de unir las entradas de la tabla de juegos con las de reseñas mediante el identificador de la aplicación. Se almacena todos los batches de juegos, y una vez que tienen todos y van llegando los de reseñas, compara batch a batch contra lo que está almacenado. Este tipo de nodos tienen estado, pero no restricciones de unicidad. Hay 2 variantes de este nodo: el joiner regular, que acumula cuentas parciales de juegos, uniendolas a su vez con su respectiva entrada de juegos, y el joiner english, que une reseñas con su respectivo juego. Este último va mandando las reseñas joineadas a medida que le llegan, mientras que el otro espera terminar de recibir reseñas para enviar resultados. Es importante destacar que no almacena de a un paquete, sino que acumula de a 50 batches para bajar a disco solo cuando llega a ese número, o cuando no tiene más paquetes para procesar.
- Aggregator: Se encarga de acumular entradas mediante algún parámetro, para realizar la cuenta de la cantidad que hay, o regenerar un número de fragmento para las capas siguientes. Hay 3 variantes de este: el OS, que junta todas las cuentas parciales de los groupers, el indie, que regenera el numero de secuencia de los paquetes, pasando la información ni bien la recibe, y el english, que tiene un comportamiento similar al joiner regular, pero sin recibir de ambas tablas, y quien envía nombres de juegos ni bien estos llegan a un count de 5.000 reseñas.
- Sorter: Se encarga tanto de ordenar los resultados, como de buscar el top de una columna. Hay 5 variantes: el playtime, quien busca el top 10 parcial de juegos con mayor playtime, el indie, quien busca el top 5 parcial de juegos con mayor cantidad de reseñas positivas, el consolidator playtime, quien obtiene los resultados de los de tipo playtime y consolida un resultado, el consolidator indie, quien hace lo mismo pero para los sorter indie, y por último el consolidator percentile, quien espera a recibir todas las reseñas, y se queda con los que esten en el percentil 90 de reseñas negativas. Todas las variantes tienen estado, y a su vez las que tienen "consolidator" en el nombre, también tienen restricciones de unicidad.

Estos nodos se comunican entre sí utilizando colas de RabbitMQ. Estas colas se encuentran abstraídas en un módulo de comunicación interna de uso común. El módulo de comunicación interna nos provee una abstracción para la comunicación para con cualquier tipo de nodo existente en nuestro sistema.

Se realizo el diagrama de paquetes ubicado en la figura 14 donde se visualizan los distintos subsistemas existentes en nuestro diseño.

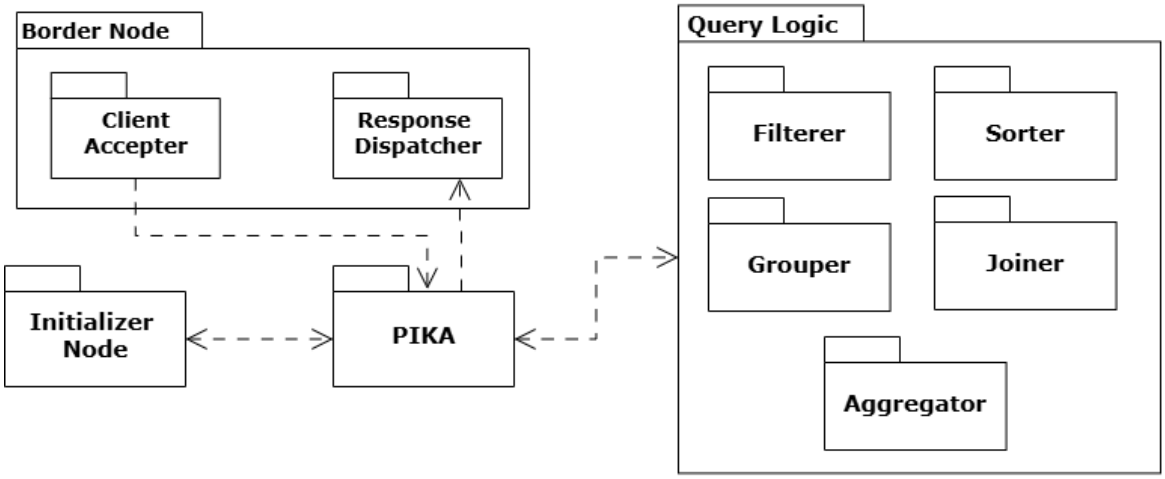

Figura 14: Diagrama de paquetes - Comunicación internodo

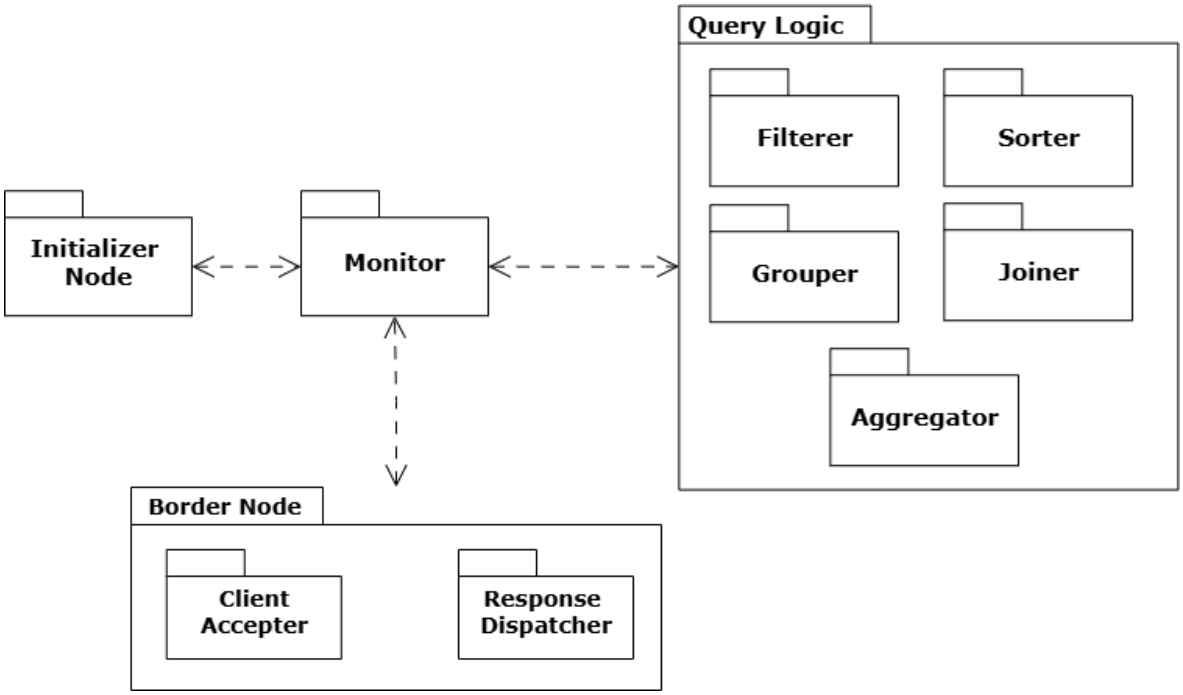

Figura 15: Diagrama de paquetes - Comunicacion con monitor

En este diagrama se representa la interacción entre las distintas entidades que conforman el sistema diseñado. En principio, el cliente se comunica con el Border Node, cuya función es la de administrar las nuevas comunicaciones e indicar el inicio del procesamiento de las queries, y luego devolver una respuesta mediante un dispatcher al cliente. Este nodo se comunica con el Initializer, que formatea la información recibida por el cliente para realizar las queries. Así, el Initializer Node re-envía dichos datos a las entidades correspondientes a la lógica de la query. Dichas entidades se comunican entre ellas, siguiendo el orden de pasos de cada query, hasta que la entidad encargada del último paso del procesamiento envía los datos al ResponseDispatcher para hacer la entrega. Todas las entidades diagramadas se comunican mediante pika, el módulo de python para el uso de colas de RabbitMQ, salvo por el cliente, que usa ZMQ para comunicarse con el BorderNode.

Por otro lado, en el diagrama 15 se muestra la interacción con el nodo monitor, con quien se comunican todos los nodos por UDP.

9. Vista física

En esta vista mostraremos cómo los componentes del sistema específicos para algunas queries se encuentran distribuídos. Para todas las queries sin embargo vale que en la entrada al sistema tenemos la comunicación con los clientes. La comunicación se realiza mediante una cola de Zero MQ, la cual nos permite unificar la entrada de datos, en lugar de tener distintos sockets por cliente. Es el client accepter, como llamamos a uno de los hilos del servidor de borde, quien lee de esta cola. Luego, el cliente recibe los resultados por la misma queue en que hizo la consulta, por lo que al finalizar el envío se queda esperando por el recibo de resultados.

Los paquetes enviados por el cliente contienen la operación a realizar. Las variantes son: hacer handshake, enviar data, ya sea de juegos o reseñas, y anunciar el final de la transferencia de información). Para el envío de data, se especifica también la tabla a la que corresponde la data, un flag para saber si es el último paquete de la tabla, el número de fragmento del paquete y las entradas que se envían en este. Este tipo de paquetes de información son propagados al siguiente nodo, en conjunto con un identificador de cliente, al initializer, y de ahi se propaga a los distintos nodos para continuar con su procesamiento.

En la figura 16, se observa como fluye la información a través del sistema y las distintas entidades existentes en este para la query 1. Las entidades que no se encuentran ubicadas dentro de una caja es porque estas son desplegadas de forma independiente.

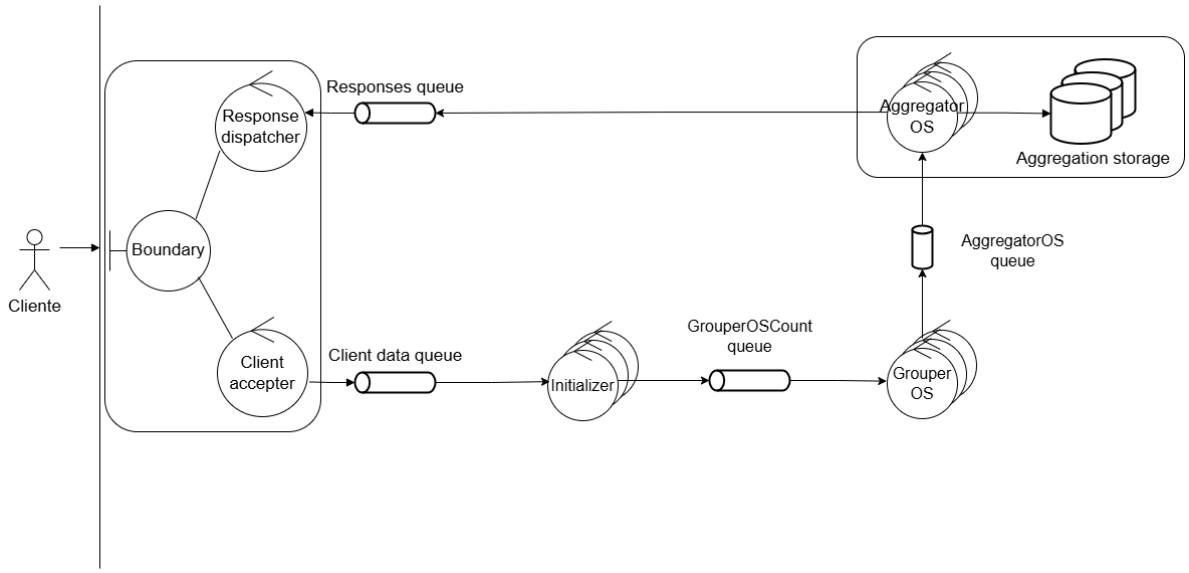

Figura 16: Diagrama de robustez para query 1

Para esta query unicamente se necesitan dos tipos de nodos específicos. El aggregator, como ya fue mencionado anteriormente, tiene la función de acumular los cálculos parciales de los groupers, y, si bien estos resultados, al ser 4 números de 4 bytes entran en memoria, el estado debería ser persistido en disco para poder ser tolerante a fallos.

En la figura 17, se observa el flujo de información para la query 5. Las entidades que están en la misma caja es porque se encuentran corriendo en la misma computadora o nodo físico.

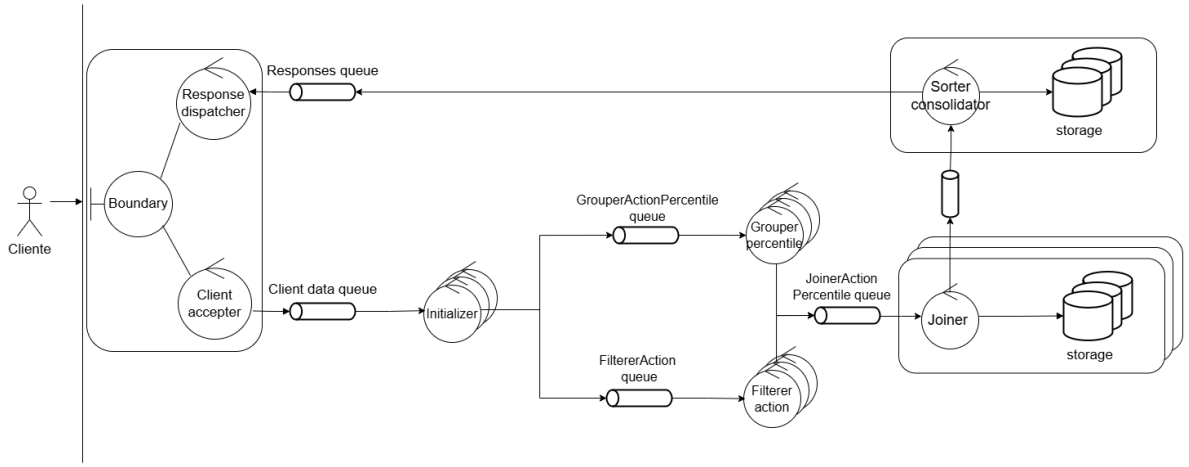

Figura 17: Diagrama de robustez para query 5

Se diagraman los storages debido a que para esta query ya la información entra en memoria. En líneas generales, el flujo funciona de la misma forma, solo que para esta query encontramos que los nodos shardean la información por el hash del appID para permitir múltiples joiners, pero dado que el cálculo del percentil se debe realizar sobre la información completa, no es posible tener múltiples sorters. Una vez obtenido el resultado, se envían las respuestas al servidor de borde, y este las envía directamente al cliente.

Por otro lado, se uso el diagrama de despliegue para ilustrar como estarán dispuestos los distintos nodos de procesamiento, ubicado en la figura 18.

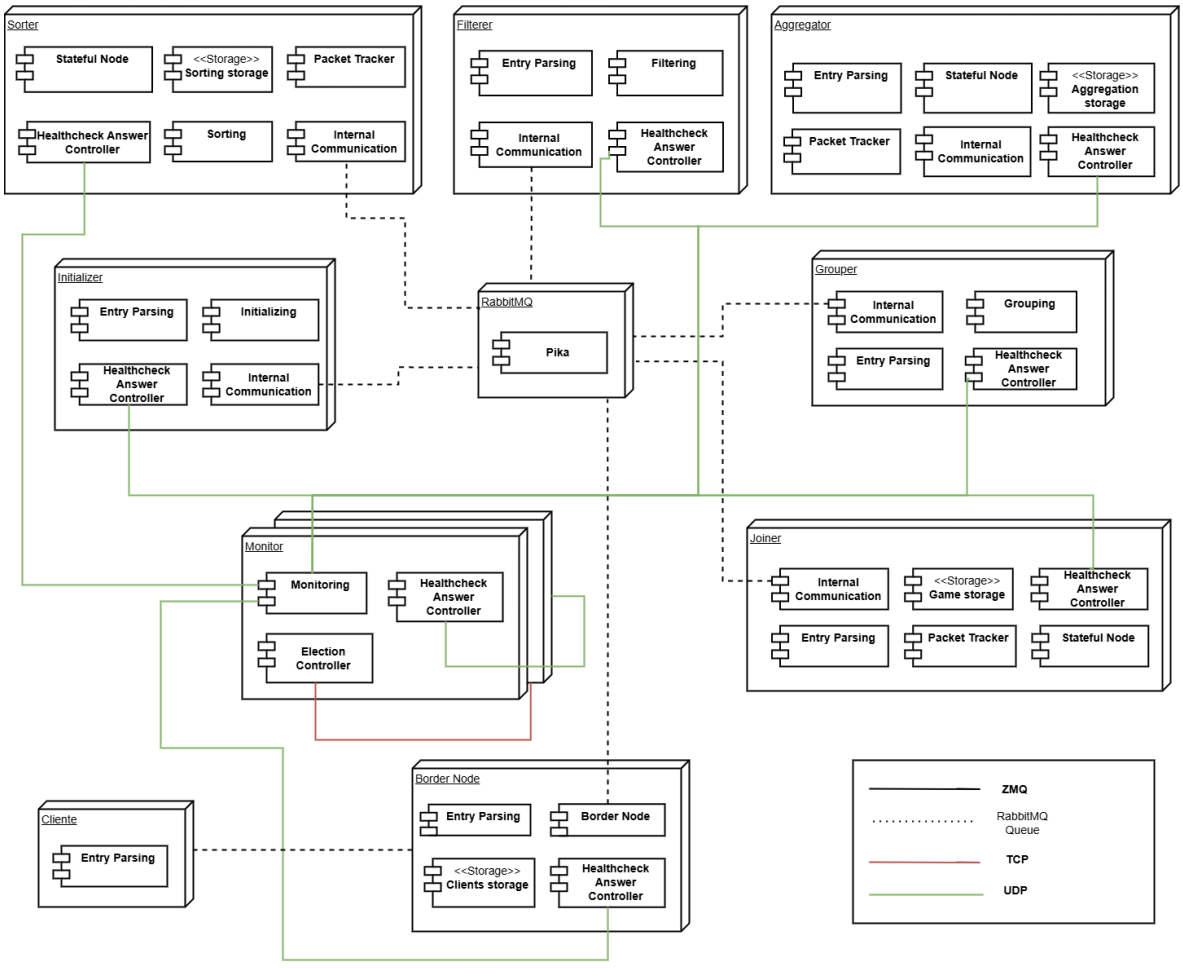

Figura 18: Diagrama de despliegue

A partir del diagrama, podemos ver dos tipos de comunicación utilizados entre los distintos nodos. Por un lado, los clientes son capaces de conectarse con el Border Node por medio de una

cola de ZMQ, que nos ofrece facilidades como 'identificadores' de sockets para el envío de los resultados. Por otro lado, los nodos de nuestro sistema se comunican entre sí por medio de colas de RabbitMQ. Para usar estos canales, los distintos nodos hacen uso del middleware 'Internal Communication', a diferencia del Border Node que utiliza el 'Border Communication', que internamente utiliza 'Internal Communication' (distinción realizada con el fin de que el resto de nodos no pueda acceder a la cola de ZMQ).

10. Detalles de implementación

10.1. Librería Internal Communication

La librería *Internal Communication* gestiona la comunicación entre los distintos nodos de procesamiento. Incluye un módulo de comunicación base que utiliza colas de RabbitMQ para el envío de mensajes. Además, este módulo se abstrae mediante estrategias denominadas *Sending Strategies*, que permiten:

- Shardear datos por `AppId` o por número de fragmento.
- Fragmentar información de gran tamaño en partes más pequeñas, leyendo los datos desde un generador de un archivo.
- Modificar las entradas recibidas, permitiendo la eliminación o modificación de columnas según sea necesario.

La configuración de estas estrategias se realiza mediante las variables de entorno `NEXT_NODES`, `NEXT_ENTRIES` y `NEXT_HEADERS`, que especifican las propiedades de los nodos siguientes. Estas variables están separadas por el carácter `;`, que delimita los nodos posteriores al actual. A continuación, se describe cada variable:

- `NEXT_NODES`: Define los nodos siguientes, el número de instancias de cada nodo y la estrategia de *sharding* (si aplica). Formato: `NEXT_NODE,NEXT_NODE_COUNT,SHARDING_ATTR;` donde `NEXT_NODE_COUNT` y `SHARDING_ATTR` son opcionales y se especifican solo si se realiza *sharding*.
- `NEXT_ENTRIES`: Define el tipo de entrada que se enviará a cada nodo. Si no se especifica, se utiliza el mismo tipo de entrada que recibe a la hora de mandarlo.
- `NEXT_HEADERS`: Define el tipo de header enviado a cada nodo. Si no se especifica, se

utiliza el mismo encabezado recibido.

El initializer hace uso de este modulo para poder cómodamente multiplexar y modificar los tipos de entradas. Este, a diferencia de los otros, usa 6 variables de entorno dedicadas, ya que la estrategia si el paquete es de la tabla de juegos no es igual a la estrategia para reseñas.

10.1.1. Ejemplo de uso

El nodo *Filterer Indie* debe enviar juegos filtrados a dos nodos siguientes: el *Filterer Date* y el *Joiner Indie*. Los detalles son los siguientes:

- El *Filterer Indie* recibe una entrada con los campos: `AppId`, `Name`, `Genres`, `ReleaseDate` y `AveragePlaytime`.
- El *Filterer Date* requiere únicamente los campos `Name`, `ReleaseDate` y `AveragePlaytime`.
- El *Joiner Indie* necesita los campos `AppId` y `Name`, y además debe recibir los datos shardeados por `AppId`.
- El encabezado recibido por el *Filterer Indie* indica la tabla a la que corresponde la información. El *Filterer Date* solo procesa datos de la tabla de juegos, por lo que no necesita este detalle, mientras que el *Joiner Indie* sí.

Con base en estas necesidades, las variables de entorno se configurarían de la siguiente forma:

- `NEXT_NODES=FilterDecade;JoinerIndie,3,0`. Esto se debe a que para los nodos siguientes de debe:
  - `FilterDecade`: Utilizar una estrategia de envío simple.
  - `JoinerIndie`: Utilizar una estrategia de shardeo por `AppId` (tipo 0), teniendo en cuenta que hay 3 instancias de este nodo.
- `NEXT_ENTRIES=EntryNameDateAvgPlaytime;EntryAppIdName` La primera entrada corresponde al nodo `FilterDecade`, y la segunda al nodo `JoinerIndie`.
- `NEXT_HEADERS=Header;` Esto equivale a `NEXT_HEADERS=Header;HeaderWithTable`. Dado que el *Filterer Indie* ya recibe el encabezado `HeaderWithTable`, se puede omitir este segundo argumento.

10.2. Manejo del End of File

En nuestro sistema, el cliente envía un número de fragmento y un flag EOF (End Of File) con cada uno de los batches de datos. El número de fragmento se genera y actualiza a lo largo del proceso, pero siempre se incluye en cada paquete enviado. Este número es fundamental para rastrear el orden de los fragmentos y asegurar que los datos sean reconstruidos correctamente al final del proceso. Además, es necesario para que los nodos que sólo pueden avanzar al completar el procesamiento total de un cliente, puedan identificar cuándo deben limpiar los datos del cliente de manera adecuada. Por último, es necesario para la recuperación ante la caída de un nodo stateful, para llevar un tracking de los paquetes ya recibidos y no procesarlos múltiples veces.

Cuando los datos se distribuyen entre los nodos mediante alguna estrategia de sharding, cada nodo recibe y procesa una parte de la información. Todos los nodos también reciben el paquete que contiene el flag EOF de la capa anterior. Si un nodo no tiene datos correspondientes en un batch que corresponde con el fin del archivo (porque el fragmento no es divisible por su ID o porque los AppIDs no coinciden con los que le corresponden), recibirá un header que indica que es un EOF, pero sin payload (sin datos).

Ahora, para estos mismos nodos, tuvimos que determinar una forma de comunicar el flag EOF a los nodos de procesamiento consecutivos. Inicialmente, habíamos optado por manejar esto incluyendo, por variables de entorno, la cantidad de unidades de procesamiento entre las que la información era shardeada, y que cada una de ellas mande su propio mensaje de EOF. De esta forma, los nodos consecuentes deberían esperar a recibir una cantidad de fragmentos con flag EOF activado igual a la cantidad de nodos previos. Dicha solución tiene un problema, pues esto supone un acople entre capas que sería ideal evitar.

De esta forma, decidimos implementar una librería 'EOFController', utilizada por aquellos nodos entre los que se shardea la información. A través de la misma, dichos nodos se comunican entre sí para decidir qué nodo debe enviar el EOF a la capa siguiente. Para ello, se utiliza una suerte de algoritmo de tipo ring", con algunas modificaciones. En este esquema, los nodos forman un círculo lógico, y van propagando mensajes entre sí. En principio, al terminar algún nodo en específico, el mismo se encarga de enviar un mensaje al siguiente, con el objetivo de indicar que ya terminó con su procesamiento (para lo cual, obviamente, incluye su ID). El

siguiente nodo en recibir dicho mensaje no lo procesará hasta no terminar él mismo con sus propios batches. Una vez que lo haga, enviará también su propio mensaje comunicando su finalización, y propagará el mensaje del nodo del que recibió un mensaje SÓLO en caso de poseer un ID menor. De esta forma, se espera que sea el nodo con el ID mayor el encargado de realizar el envío del EOF, que esperará a que su propio mensaje "gire.ªlrededor del ring hasta llegar a sí mismo. Una vez enviado el EOF, se inyectará en el ring un mensaje ACK (Acknowledgment), que girará alrededor del mismo con el objetivo de que dichos controladores eliminen toda la información almacenada respecto al cliente en cuestión.

Esta metodología nos garantiza que los datos se transmitan de manera ordenada y controlada, evitando duplicaciones y asegurando que solo un nodo indique el fin de los datos para la capa siguiente.

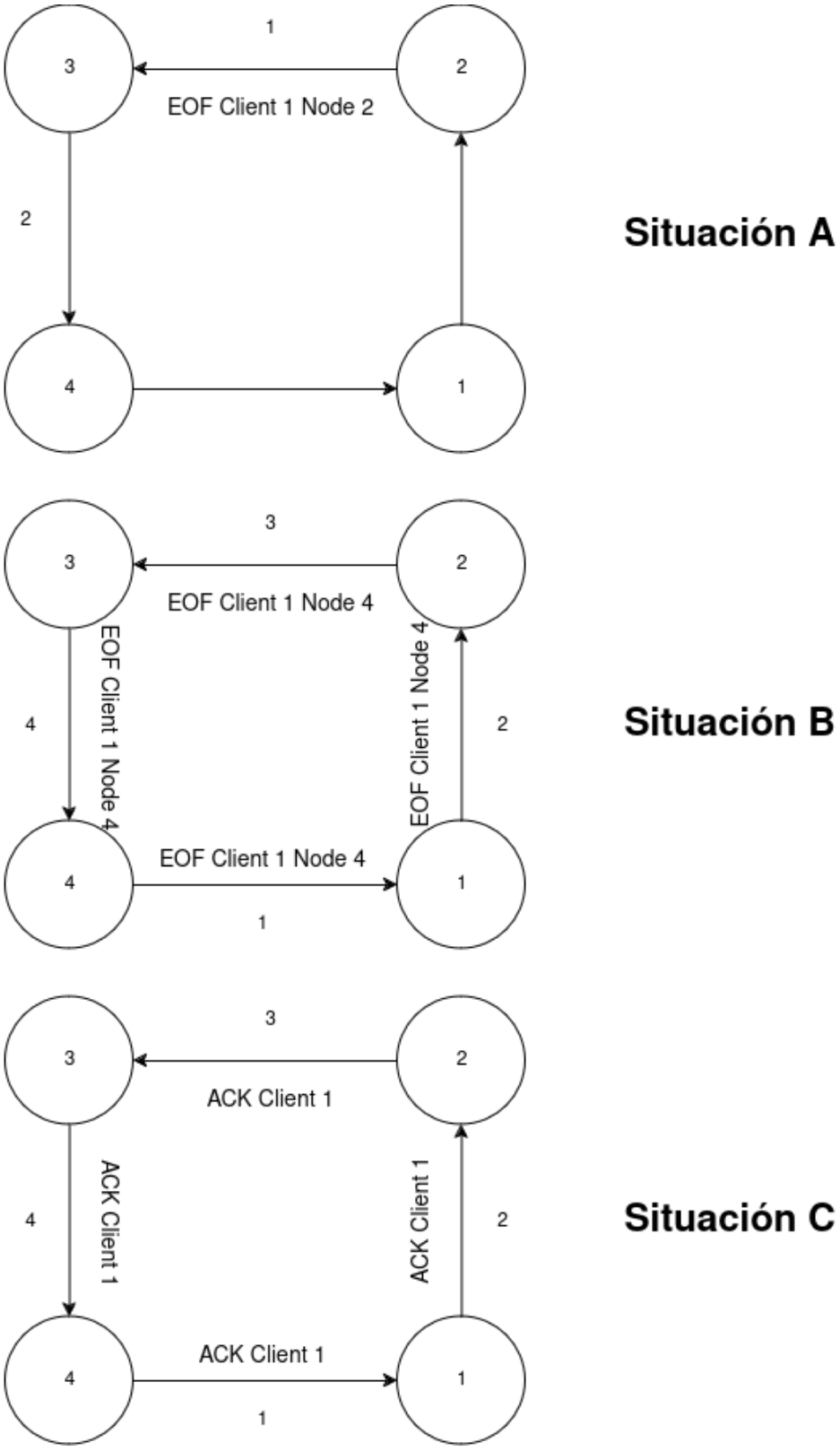

Figura 19: Diagrama de funcionamiento del EOF-Controller

En la figura 19 se muestra el funcionamiento de dicho sistema. En la situación A, vemos que el nodo 2 comunica su finalización, pero el nodo 3 no propaga dicho mensaje pues se trata de un nodo de ID menor. En la situación B, vemos que el nodo 4 comunica su finalización, y dicho mensaje se propaga alrededor del ring, pues dicho nodo ostenta el ID más grande. En la situación C, vemos como el nodo 4 comunica el ACK que se propaga por todo el ring hasta llegar a sí mismo.

10.3. Librería PacketTracker

El módulo **PacketTracker** es responsable de hacer seguimiento a los fragmentos de datos recibidos, y es utilizado para el manejo del EOF explicado en la sección anterior. Su objetivo principal es asegurarse de que todos los fragmentos correspondientes sean procesados, sin importar el orden en que se reciban, y además, que sean procesados sin duplicaciones. Todos los nodos stateful hacen uso de este módulo.

Fragmentos Pendientes: Uno de los elementos clave del tracker es la gestión de los fragmentos pendientes. El conjunto `_pending` mantiene un registro de los fragmentos que aún no se han recibido. Esto es necesario cuando se saltan fragmentos, es decir, cuando un fragmento con un número mayor llega antes que uno con un número intermedio (por ejemplo, si se recibe el fragmento 5 antes que el 3 y 4). En este caso, el rastreador marca estos fragmentos intermedios como pendientes.

Actualización de Fragmentos: Cada vez que se recibe un fragmento, se ejecuta el método `update`, que realiza lo siguiente:

- Si el fragmento recibido tiene un número mayor que el fragmento más grande procesado hasta el momento (`_biggestFragment`), se añaden al conjunto `_pending` todos los fragmentos intermedios que faltan. Esto asegura que se espera la llegada de los fragmentos que aún no se han recibido, pero que son necesarios para completar el procesamiento.
- Si el fragmento recibido es menor que el número de fragmento más grande, simplemente se elimina de `_pending` si está presente, y si no está simplemente se lo ignora, ya que es un repetido.

Finalización del Proceso: El método `isDone` determina si el rastreador ha recibido to-

dos los fragmentos necesarios para completar el procesamiento. Esto se consigue comprobando que no haya fragmentos pendientes (`_pending` vacío) y que se haya recibido el fragmento que contiene el flag EOF en true, lo que indica que todos los datos han sido procesados.

11. Tolerancia a fallos

11.1. Border Node

El servidor de borde implementa un mecanismo de *stop-and-wait* para garantizar la fiabilidad en la comunicación con los clientes, particularmente debido a las características de `ZeroMQ`, que no asegura la retransmisión de mensajes leídos en caso de fallos. Este mecanismo es esencial para evitar la pérdida de paquetes en escenarios donde el servidor pueda experimentar una caída después de recibir datos de un cliente.

El proceso de *stop-and-wait* opera de la siguiente manera:

1. El cliente envía un mensaje al servidor.
2. El servidor recibe el mensaje y lo transmite inmediatamente al nodo siguiente en la cadena de procesamiento, en este caso el *initializer*.
3. Solo después de haber transmitido este mensaje al *initializer*, el servidor envía un *acknowledgment* (ACK) al cliente para confirmarle la recepción del mensaje.

Si el cliente no recibe el ACK, dispone de un mecanismo de reintentos para reenviar el mensaje. Este flujo garantiza que, en caso de una caída del servidor, los datos no se pierdan. Dado que `ZeroMQ` considera un mensaje como "leído" tan pronto como se recibe, cualquier fallo posterior podría provocar la pérdida de datos si no se empleara este esquema. Retrasar el envío del ACK hasta confirmar la transmisión al *initializer* asegura que el mensaje está seguro en la siguiente etapa antes de informar al cliente.

Esta forma de procesamiento debe ser complementada con el uso del *packet tracker* mencionado anteriormente para el correcto manejo de posibles mensajes duplicados enviados por el cliente: si el mensaje ya fue procesado, será descartado automáticamente.

El manejo de tolerancia a fallos en relación con el registro de los clientes y cómo se gestiona la caída de un cliente se detalla en la sección 11.4.

11.2. Monitor

Los monitores son un grupo de nodos responsables de supervisar el estado del sistema, capaces de reiniciar nodos usando Docker in Docker. Para mantener una visión global centralizada del sistema se elige entre ellos un líder quien va a realizar *healthchecks*. Este es entonces el encargado de:

- Supervisar el estado de todos los nodos del sistema, incluidos los demás monitores.
- Reiniciar cualquier nodo que se detecte como caído.

Los *healthchecks* se realizan por medio de UDP para una mayor velocidad y simpleza. Para intentar mitigar el número de falsas caídas debido a la pérdida de paquetes, se incorporó una política de *retries*.

Para medir el tiempo se utilizaron los timers de ITIMER_REAL junto con el *signal handler* de la señal SIGALRM. Cada cierto tiempo se ejecuta la función *checkStatus*, la cual se encarga de enviar *healthchecks*, revisar si algún nodo no respondió luego del máximo de *retries* y en caso afirmativo reiniciarlo.

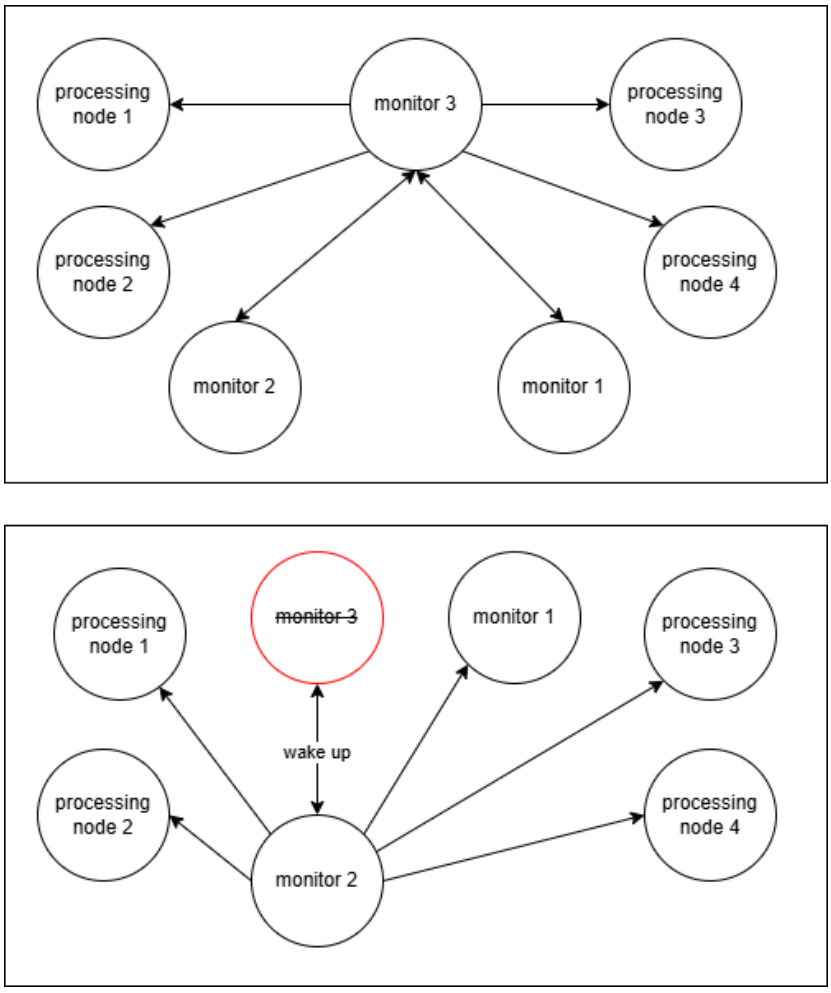

Figura 20: Funcionamiento de monitores

Los monitores que no son líderes verifican periódicamente si el líder sigue activo mediante el monitoreo de los *healthchecks* que este envía. La manera de determinar si este está caído es idéntica a la del líder pero difieren en cómo se maneja dicha caída. Siguiendo la política de retries, si detectan que ningún *healthcheck* luego de determinada cantidad de retries, dan por muerto al monitor líder, e inician un proceso de elección de líder utilizando el algoritmo *bully*. Para esto, a diferencia de los *healthchecks*, se utiliza TCP, debido a que el algoritmo elegido toma como hipótesis que se desarrolla utilizando Reliable Data Transfer. Una vez elegido, el nuevo líder toma el control y comienza a supervisar y recuperar nodos caídos, levantando en primer lugar al anterior líder. Cuando el líder original termina su proceso de recuperación, anuncia su disponibilidad para retomar su posición como líder.

En la figura 20, podemos observar en la imagen superior cómo el sistema opera cuando todos los nodos están activos. En la imagen inferior, se muestra el caso en que el monitor con ID 3, que es el líder, falló. Durante este escenario, el monitor con ID 2, siendo el segundo con mayor ID, asume el liderazgo temporal hasta que el monitor original complete su recuperación.

11.3. Persistencia en nodos stateful

En los nodos *stateful*, la persistencia de datos, en conjunto con el correcto manejo de los ACKs de paquetes de RabbitMQ, es clave para garantizar la tolerancia a fallos y evitar pérdidas de información.

A diferencia del servidor de borde, los nodos de procesamiento se comunican entre sí usando colas de Rabbit, las cuales nos proveen persistencia de datos y un mecanismo nativo para el ACK. En nuestro sistema, el ACK de un mensaje se realiza únicamente después de que su información ha sido correctamente registrada y, de ser necesario, transferida al siguiente nodo en la cadena de procesamiento. Esto asegura que los mensajes no sean eliminados prematuramente de la cola.

El *packet tracker* desempeña un papel fundamental en la persistencia de los datos. Su información se almacena como un encabezado en los archivos de datos, lo que permite reanudar el procesamiento tras una interrupción, pudiendo notar fácilmente qué paquetes ya habían sido procesados ante una caída previa al ACK, y cuales aún quedan por serlo. Específicamente, el encabezado del archivo incluye:

- El número del fragmento más alto recibido.
- Indicadores de recepción de un paquete de `EOF`.
- La lista de fragmentos pendientes

Para almacenar los datos, se utiliza un esquema basado en archivos temporales en lugar de realizar escrituras continuas (*append*) en el archivo principal. Esto evita la corrupción de datos en caso de fallos durante el proceso de escritura. El flujo de trabajo es el siguiente:

1. Los nuevos datos se escriben en un archivo temporal junto con el *packet tracker* actualizado para reflejar el paquete procesado. Este archivo incluye tanto la información recién recibida como los datos existentes en el archivo principal.

2. Una vez completada la escritura del archivo temporal, este reemplaza al archivo principal mediante una operación atómica de `rename`.

En caso de una caída del nodo, al reconectarse, este descarta cualquier archivo temporal y revisa los archivos de datos de todos los clientes para cargar en memoria los clientes que estaba gestionando y sus respectivos *packet trackers*.

11.4. Caídas de clientes

El servidor de borde implementa un mecanismo para detectar y manejar la desconexión de clientes mediante el uso de temporizadores y señales del sistema operativo. Los clientes son registrados en el sistema durante el proceso de *handshake*, momento en el que se los agrega a una lista de clientes activos, junto con un *timestamp* inicial que marca el momento del registro. Cada vez que un cliente envía un mensaje, el servidor actualiza su *timestamp* correspondiente, permitiendo monitorear su actividad.

El sistema utiliza señales como `SIGALRM` junto con un temporizador periódico configurado con `ITIMER_REAL`. Cada dos segundos, se genera una alarma que invoca el método `handleTimeoutSignal`, encargado de revisar los clientes activos. Un cliente se considera desconectado si la diferencia entre el tiempo actual (`now`) y el *timestamp* del último paquete recibido (`lastTimestamp`) supera la frecuencia del temporizador (`timerFrequency`), es decir, se cumple la condición:

$$\texttt{now} - \texttt{lastTimestamp} > \texttt{timerFrequency}$$

Cuando se identifica un cliente inactivo, el servidor inicia un proceso de limpieza para liberar los recursos asociados, asegurando que no queden datos residuales que afecten el rendimiento del sistema. Este proceso incluye la propagación de un mensaje de `flush` a los nodos correspondientes.

Si un cliente completa su transmisión, este se elimina de la lista de clientes activos, ya que no se espera que envíe más mensajes. El sistema identifica esta finalización de dos maneras: mediante un paquete de `data transfer` marcado con el indicador EOF asociado a la tabla de reseñas, o a través de un mensaje explícito de tipo `finish data transfer`. En escenarios

de reinicio del servidor, este último mensaje actúa como una capa adicional de seguridad, confirmando que el cliente fue eliminado correctamente.

Los identificadores de los clientes activos se persisten en el almacenamiento del sistema, pero no así sus *timestamps*. Tras un reinicio, todos los *timestamps* se inicializan al momento del arranque, lo que implica que el sistema asume que los clientes registrados aún están conectados y comienza a monitorear su actividad desde ese instante. El mensaje de finalización, combinado con el mecanismo de comunicación *stop and wait*, garantiza que los estados del servidor y del cliente permanezcan siempre consistentes.

11.4.1. Limpieza ante la caída

El proceso de limpieza varía según el tipo de nodo involucrado. Los nodos *stateless* simplemente transmiten el mensaje de tipo `flush` a la capa siguiente sin realizar procesamiento adicional. Por otro lado, los nodos *stateful* implementan una lógica más elaborada para garantizar la integridad de los datos antes de completar la limpieza definitiva.

Cuando un nodo *stateful* recibe un mensaje de tipo `flush` por primera vez, ejecuta las siguientes acciones:

1. **Liberación de recursos**: El nodo libera todos los recursos asociados al cliente caído, tanto en memoria como en disco.
2. **Almacenamiento temporal**: El nodo almacena en memoria ("cachea") el identificador del cliente caído junto con un *timestamp* que registra el momento de recepción del mensaje de `flush`.
3. **Retransmisión del mensaje**: Transmite el mensaje de `flush` a todos los nodos de la capa siguiente.
4. **Reencolado del mensaje**: Reencola el mensaje de `flush` en la cola de procesamiento, garantizando que no se pierda información en caso de que el nodo sufra una caída.

La lógica del reencolado se justifica por un posible desfase entre el procesamiento del mensaje de `flush` y el de otros mensajes asociados al cliente. Aunque los mensajes son procesados en orden FIFO, el mensaje de `flush` podría ser más liviano y transmitirse antes de que se completen los mensajes de procesamiento en curso. Esto podría provocar inconsistencias en

los nodos de la capa siguiente, que recibirían el mensaje de `flush` antes de procesar toda la información relacionada con el cliente.

Para mitigar este riesgo, el nodo *stateful* asigna un tiempo de espera antes de descartar el mensaje definitivamente. Cuando el mensaje de `flush` vuelve a ser procesado tras el reencolado, el nodo verifica si ha transcurrido el tiempo necesario desde su recepción inicial. Si se cumple este umbral temporal, el nodo descarta el mensaje. De lo contrario, lo reencola nuevamente, repitiendo el ciclo hasta que sea seguro eliminarlo. Es importante destacar que ante la caída de un nodo, la primera recepción del mensaje, aunque haya estado reencolado, deberá llevar a cabo todos los pasos previamente mencionados. Sin embargo, como las operaciones son idempotentes, no hay riesgo de efectos no deseados; es decir, no importa si el mismo mensaje se procesa varias veces, ya que los resultados serán los mismos. Eventualmente, el temporizador alcanzará el umbral necesario, y el mensaje será descartado de forma definitiva, sin causar inconsistencias.

12. Referencias

[1] Gomaa, Hassan: Software Modeling & Design. UML, Use cases, pattern & software architectures. Cambridge, 2011.

[2] Rumbaugh, James, Jacobson, Ivar, Booch, Grady. The Unified Modeling Language Reference Manual, 1999.

[3] Philippe Kruchten, Architectural Blueprints - The "4+1" View Model of Software Architecture, 1995

[4] Fowler, UML Distilled, 1997