75.74 - Sistemas Distribuidos I

TP2: Tolerancia a Fallos
Grupo 08

2° Cuatrimestre 2023

Nombre y Apellido	Mail	Padrón
Prada, Joaquín	jprada@fi.uba.ar	105978
Sotelo Guerreño, Lucas Nahuel	lsotelo@fi.uba.ar	102730

Índice

1. Introducción

Se creó un sistema distribuído para analizar 6 meses de registros de precios de vuelos para proponer mejoras en la oferta a clientes. Para el desarrollo del mismo se tuvo en cuenta que sea escalable y tolerante a fallos.

Este documento detalla las decisiones de arquitectura y de diseño que se tomaron, así como la resolución de las queries que se pidieron como requerimientos. Las secciones se pueden agrupar de la siguiente manera:

- Secciones 2, 3, 4 y 5: hablan de la arquitectura del sistema en general y explican el pipeline de resolución de cada query.
- Secciones 6 en adelante: todo lo referido a la resolución de los problemas relacionados con la tolerancia a fallos.

2. Arquitectura

Objetivo de la arquitectura

El objetivo de la arquitectura es poder paralelizar lo máximo posible el filtrado y procesado de cada entrada de vuelos. Es importante que solo se lea cada entrada una única vez, ya que al ser muchos vuelos sería muy ineficiente tener que volver a leerlas. La arquitectura apunta a ser escalable, en donde se necesite más cómputo se puede escalar y acelerar su procesado.

La cantidad de datos a procesar provienen de un archivo de aproximadamente 31GB.

Flujo de la arquitectura

La arquitectura está dividida en 3 principales partes, el cliente, encargado de la subida de los archivos y recepción de resultados. El servidor, encargado de recibir los mensajes del cliente para empezar el procesamiento, como también el envío de resultados al cliente. Y por último las entidades encargadas de la modificación y procesamiento de los mensajes, llamados *processors*.

A continuación vamos a contar como es el flujo de estas partes y como en conjunto forman la arquitectura.

Cliente

El flujo de la aplicación comienza en el cliente, quien empieza por el envío de los archivos necesarios.

El cliente envía dos archivos por el mismo socket de conexión con el servidor: el dataset de aeropuertos y el dataset de vuelos. La forma en que el servidor puede diferenciarlos es

mediante el primer byte del mensaje enviado por el cliente que indica de qué dataset proviene: 1 si es el dataset de aeropuertos y 2 si es el dataset de vuelos. De esta manera el servidor lee el primer byte de cada mensaje que recibe del cliente y ya puede saber cómo interpretarlo.
Cada batch de mensajes se delimita por el siguiente string: “\r\n\r\n” y, además, pueden contener varias líneas del archivo que se delimitan por el string: “\n”.
Ejemplos:

- “1EZE,102,234\nAEP,112,221\n\r\n\r\n”
- “2vuelo1,1h30m,EZE\n\r\n\r\n”

Para indicar que se terminó de enviar el archivo completo, el cliente envía el string: “\0” (también precedido por el byte que determina el tipo de dataset).

El cliente puede envíar sus mensajes en “batches” significando enviar múltiples mensajes juntos. Esto se logra simplemente separando los mensajes por el “\n” como se explicó anteriormente, al enviar los mensajes con batches se logra un aumento muy notable de performance.

Por último, el cliente recibe los resultados y los separa en sus archivos correspondientes, creados a partir del “tag” recibido, este tag hace referencia al resultado de la “query” correspondiente.

Server

Mientras que el cliente envía sus mensajes, es el servidor el encargado de recibirlos y empezar a distribuirlos por el sistema.

Cuando recibe una conexión nueva, la empieza a manejar y escuchar sus mensajes. Cuando recibe un nuevo mensaje, lo primero que hace es ver de qué tipo se trata, como explicamos previamente, este puede ser del dataset de aeropuertos o de vuelos. Luego de averiguar el tipo se lo pasa a su respectivo *uploader*, que sabe a qué entidad pasarlo.

Por otro lado, el servidor tiene otra entidad en un proceso independiente, para el recibo y envío de los resultados hacia el cliente.
El servidor le envía un mensaje el cual es precedido por un “tag” entre corchetes, por ejemplo: “[DISTANCIA]”, que indica de qué tipo de resultado se trata. La forma de delimitar los mensajes es la misma que en la comunicación cliente -> servidor.

Processors

Por último en la arquitectura, están las entidades encargadas de la modificación y procesamiento de la información/mensajes, su forma es muy similar entre sí, la diferencia que tienen es **cómo** modifican los datos. Es por eso que acá se explica cómo se componen y cómo se comunican entre sí, que son los 2 puntos más fundamentales en la arquitectura. Sus detalles de procesamiento de los datos se encuentran explicados en los diagramas de cada caso de uso.

Comencemos primero por cómo se componen estas entidades. Hay 3 principales partes dentro de cada entidad.
Donde en primer lugar tenemos la inicialización de su configuración, que gracias a la similitud que tienen entre sí, utilizamos variables de entorno para definirlas. Estas son procesadas con un “config parser” y pasadas a las distintas abstracciones. Después inicializan su logger con su determinado nivel.
Luego está la abstracción de **Communication** donde se abstrae todo sobre cómo se comunican entre distintas entidades.
Y por último, está el procesador en sí, donde está contenida las reglas de negocio de cada entidad en particular.

Ahora, ¿cómo hacen estas entidades para comunicarse entre sí? Para eso tenemos la abstracción de **Communication**, es acá donde se esconde el corazón de la arquitectura. Hay 2 tipos de comunicación, el **Receiver**, encargado de recibir mensajes entrantes, y el **Sender**, encargado del envío de mensajes. Una entidad puede tener tantos “receiver” como “sender” desee, creados utilizando el **Communication Initializer** utilizado en la inicialización (main).
Las entidades tienen que saber también qué tipo de entrada y salida tienen, esto significa que estos tipo puede ser una cola o un exchange, es por eso que los sender y “receiver” son de tipo **Queue**, o **Exchange**.

Pasemos entonces a explicar como funcionan estos **Communication**.
Empecemos con el receiver, lo que va a hacer un receiver, en su forma más básica, es leer de un “input”, puede ser el nombre de una cola o un exchange (lee un predeterminado número de “prefetch” para ayudar con el *throughput)*, y cuando recibe un mensaje nuevo se lo pasa a un “callback” para que haga lo que quiera con el mensaje. Este callback, junto al eof callback tienen que indicarse al llamar a la función **bind** luego de la creación del “receiver”. El mensaje que se le pasa al callback ya está “parseado”, significando que recibe una adaptación del mensaje, ocultando como es el formato de envío. Es necesario igual indicarle que es lo que se está esperando con una lista de campos cuando se hace el bind.

Por otro lado, el sender, encargado de enviar los mensajes, tiene 2 funciones importantes. Primero el **send_all** utilizado para mandar una lista de mensajes aprovechando el “batching” que llega desde el cliente. Dependiendo de qué tipo es a quien le estamos enviando, puede que sea necesario indicarle una **routing key** para saber a quien enviarle.
Luego está el **send_eof** que envía la notificación de que se terminó de mandar todo, y esto es muy importante para el dominio del problema, ya que hay entidades que solo pueden actuar cuando saben que ya recibieron todo.

3. Escenarios

Queremos cubrir las siguientes consultas:

1. ID, trayecto, precio y escalas de vuelos de 3 escalas o más.
2. ID, trayecto y distancia total de vuelos cuya distancia total sea mayor a cuatro veces la distancia directa entre puntos origen-destino.

3. ID, trayecto, escalas y duración de los 2 vuelos más rápidos para todo trayecto con algún vuelo de 3 escalas o más.
4. El precio avg y max por trayecto de los vuelos con precio mayor a la media general de precios.

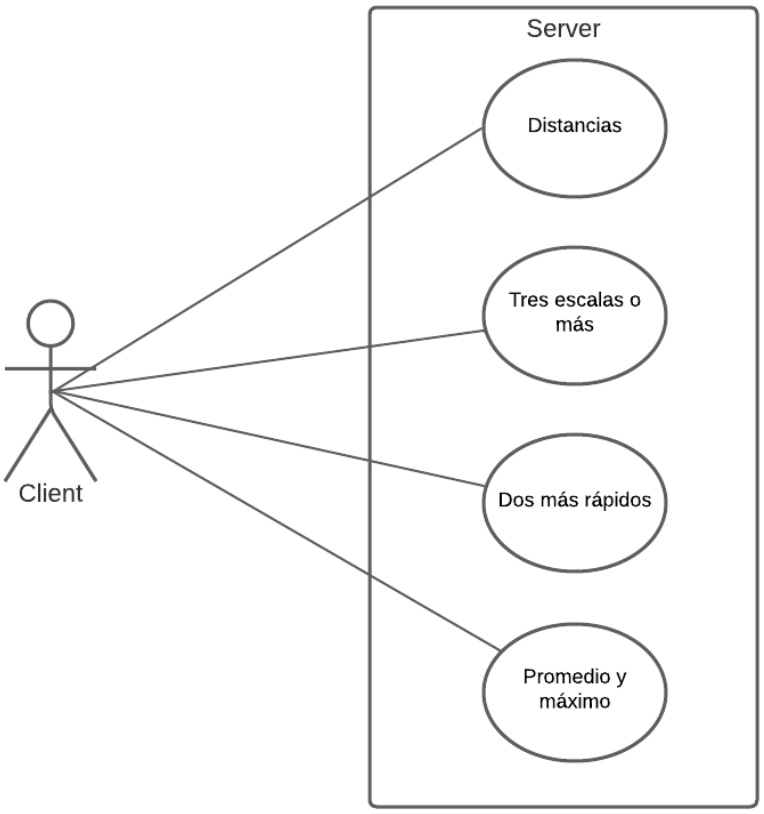

Figura 3.1: Diagrama de casos de usos

4. Desarrollo

A continuación se detalla el flujo de resolución de cada una de las consultas a resolver y las entidades que lo conforman.

Query 1: Tres escalas o más

Vista Lógica

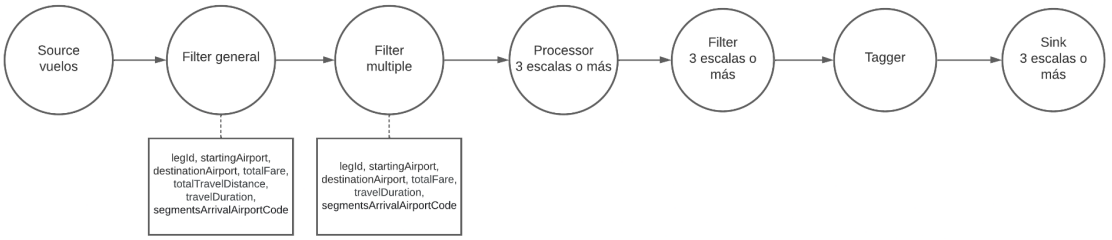

Figura 4.1: DAG de la consulta de vuelos con tres escalas o más

En esta consulta los vuelos primero pasan por dos **Filters** para que el **Processor** tenga solamente la cantidad de columnas necesarias para el procesamiento (son dos **Filters** porque el primero es compartido por otras consultas). Una vez que el **Processor** recibe un vuelo lo que hace es verificar si el mismo realiza tres escalas o más, en caso que así sea lo entrega a la siguiente entidad, en caso contrario lo descarta.

La siguiente entidad es un **Filter** que elimina las columnas innecesarias para la respuesta y envía el vuelo al **Tagger**. Finalmente, el **Tagger** le agrega el tag correspondiente al vuelo y lo envía al **Sink**.

Vista de Proceso

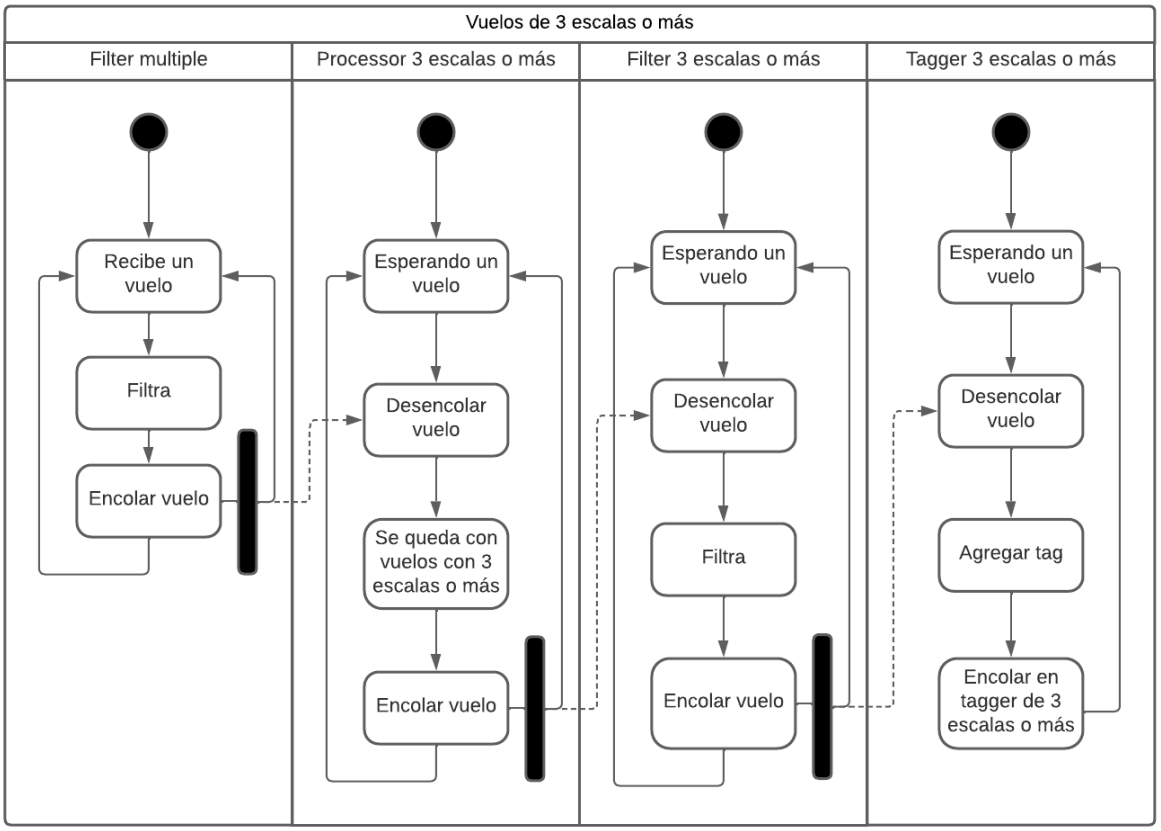

Figura 4.2: Diagrama de actividad de la consulta de vuelos con tres escalas o más

Como se puede apreciar en el diagrama de actividad de la Figura 4.2, el flujo es bastante sencillo, simplemente van pasándose los vuelos y cada uno realiza un acción sobre el mismo.

Query 2: Distancias

Vista Lógica

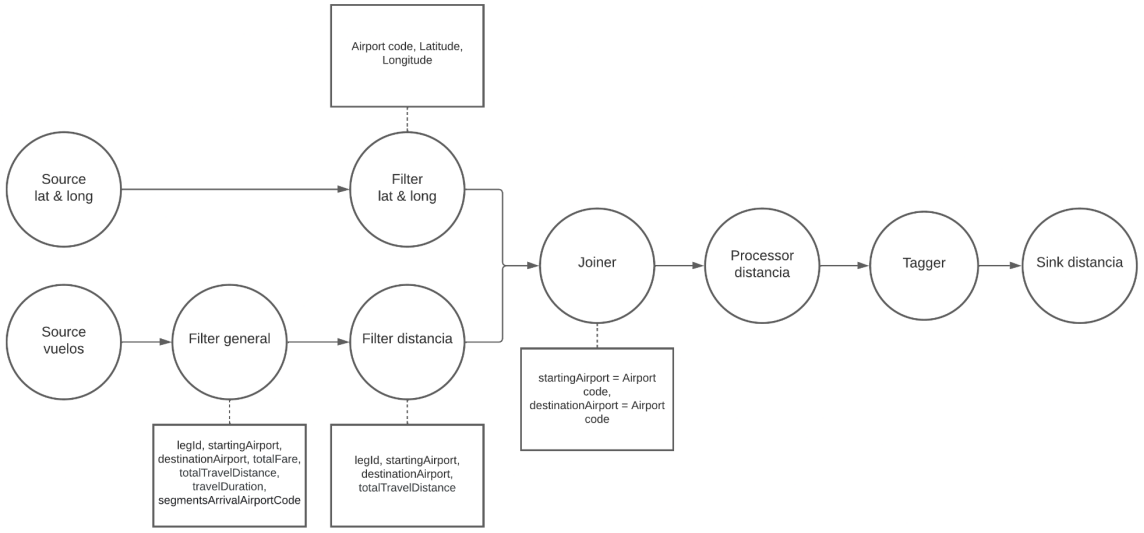

Figura 4.3: DAG de la consulta de vuelos cuya distancia total sea mayor a cuatro veces la distancia directa entre puntos origen-destino

Para realizar esta query necesitamos la latitud y longitud de cada aeropuerto, que no se encuentran en el dataset de vuelos. Por esto realizamos una junta de cada vuelo con dichos datos que provienen de otro dataset.

El **Joiner** es el encargado de hacer la junta, por lo tanto, primero espera recibir todos los datos del dataset de latitud y longitud, al ser pequeño (~875kB) lo mantiene guardado en memoria.

Una vez recibido todo el dataset de latitud y longitud, comienza a juntar pedidos del dataset de vuelos y por cada vuelo recibido le agrega la latitud y longitud del aeropuerto de salida y el aeropuerto de llegada.

El vuelo junto a las latitudes y longitudes se envían al **Processor** de distancias que compara la distancia total con la distancia directa y si cumple la condición es enviado al **Tagger**. El **Processor** de distancias mantiene un diccionario en memoria que funciona como *cache* en el cuál se van guardando las distancias entre aeropuertos ya calculadas, esto se realiza porque dicho cálculo consume un tiempo de procesamiento no despreciable. Al tener una baja cardinalidad de aeropuertos, este *cache* no crece demasiado y no representa un problema en el uso de la memoria.

El **Tagger** finalmente agrega un tag al vuelo para indicar que es un resultado de este tipo de query y lo envía al **Sink** correspondiente.

Vista de Proceso

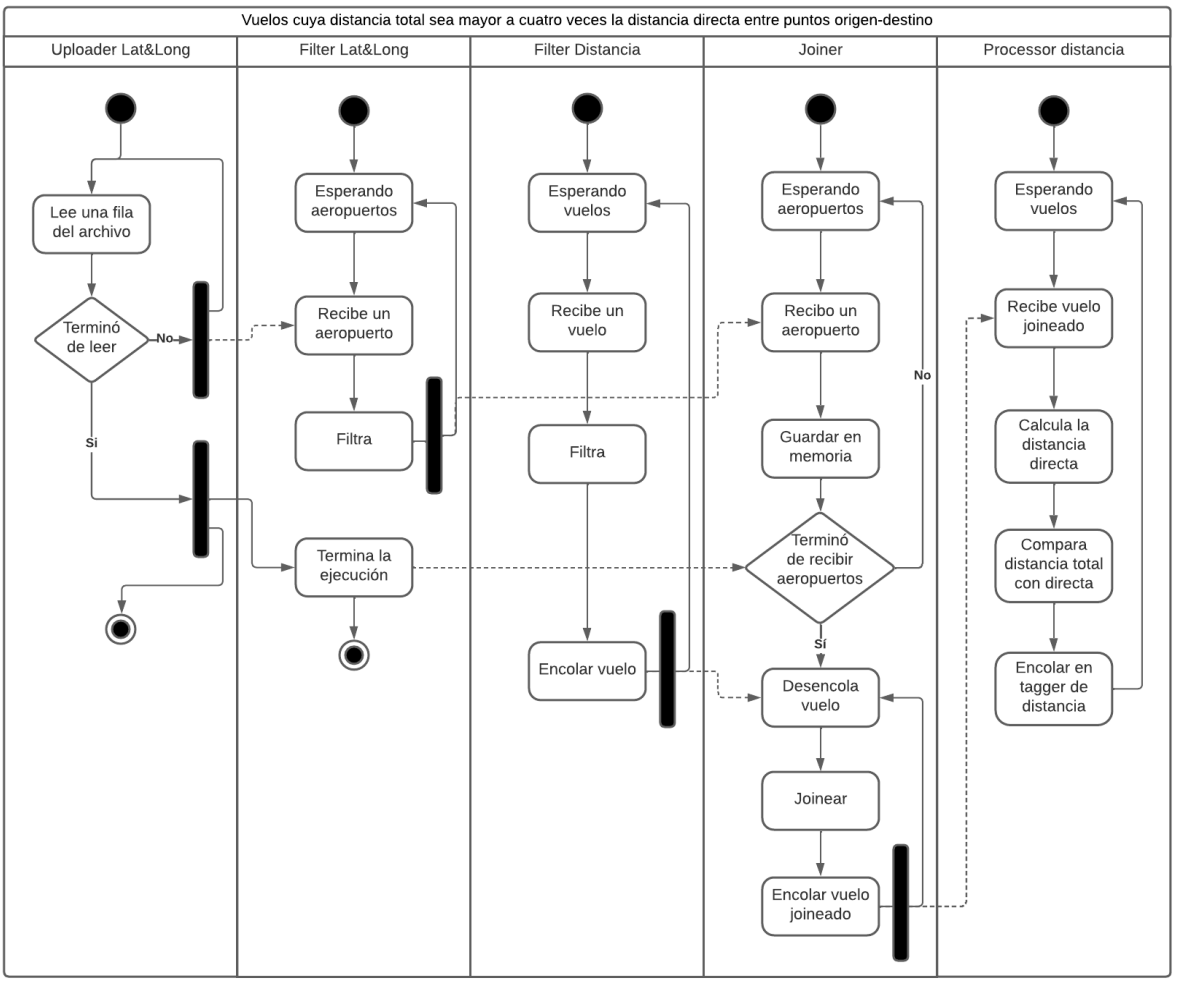

Figura 4.4: Diagrama de actividades de la consulta de vuelos cuya distancia total sea mayor a cuatro veces la distancia directa entre puntos origen-destino

Query 3: Dos vuelos más rápidos

Vista Lógica

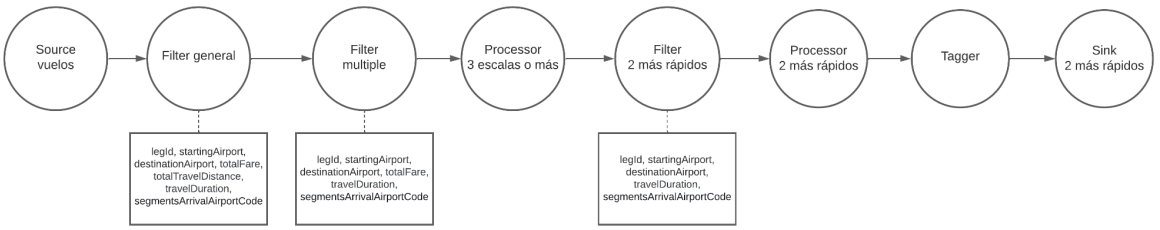

Figura 4.5: DAG de la consulta de los dos vuelos más rápidos que tengan tres escalas o más

Para esta consulta se reutiliza lo procesado para la consulta de tres escalas o más hasta el **Processor** de tres escalas o más. La salida de dicho **Processor** pasa por un **Filter** para eliminar columnas innecesarias y luego lo toma el **Processor** de dos más rápidos.

Este **Processor** lo que hace es ir guardando los dos vuelos más rápidos por trayecto en un diccionario en memoria, donde la clave es el trayecto y el valor una lista ordenada con los dos vuelos más rápidos de dicho trayecto. El valor de los vuelos más rápidos se va pisando, si corresponde, a medida que van llegando los vuelos, hasta que llegue el mensaje de *end of file* que señala el fin del proceso del dataset de vuelos. En ese caso, el **Processor** envía al **Tagger** el resultado de cada trayecto. El **Tagger** le agrega el tag que corresponde y lo envía al **Sink**.

Vista de Proceso

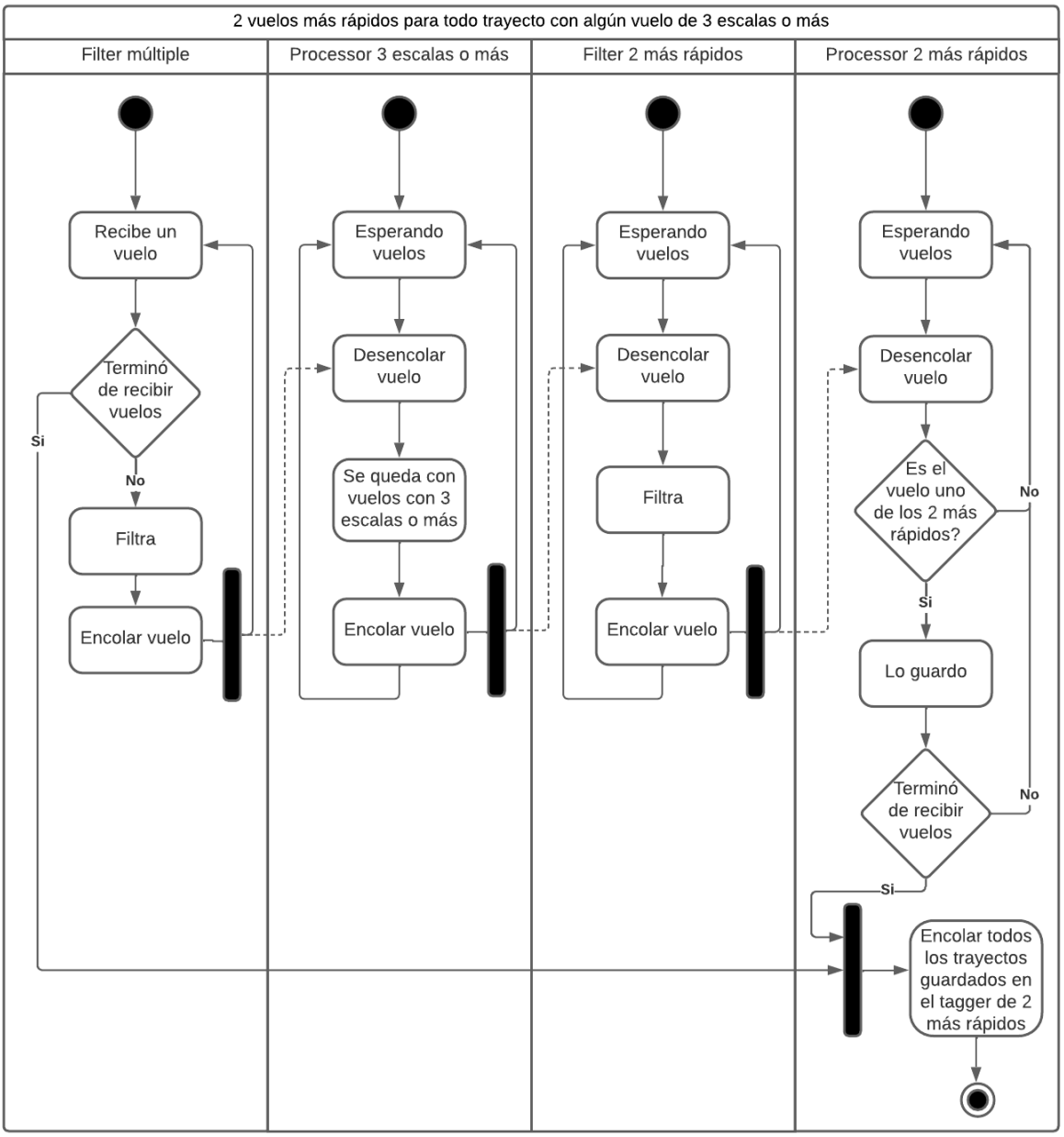

Figura 4.6: Diagrama de actividades de la consulta de los dos vuelos más rápidos que tengan tres escalas o más

Query 4: Promedio y máximo de precios

Vista Lógica

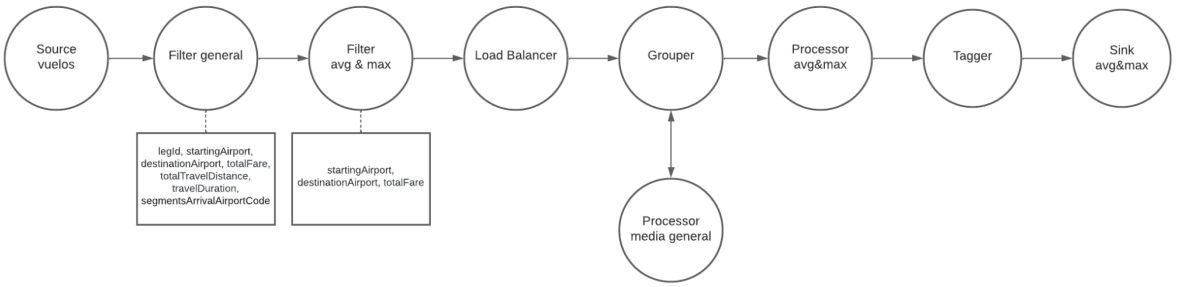

Figura 4.7: DAG de la consulta del promedio y máximo de precios cuyo precio está por encima de la media general (por trayecto)

Para esta consulta contamos con la entidad **Load Balancer** que realiza un balanceo de carga particionando el dataset de vuelos hacia cada instancia de **Grouper**. Cada **Grouper** es encargado de procesar una partición del dataset de vuelos, esta partición se hace por trayecto: se aplica una función de hash al trayecto (combinación entre aeropuerto de salida y aeropuerto de destino) y a ese resultado se le aplica un módulo por la cantidad de instancias de **Groupers** para decidir a cuál de ellos debería ser enviado.

Los **Groupers** agrupan los precios de los vuelos por trayecto en un diccionario en memoria, donde la clave es el trayecto y el valor es la lista de precios. Una vez recibido el *end of file* que indica la finalización del dataset de vuelos, cada **Grouper** suma todos los precios de todos los trayectos y calcula la media. Este valor es envíado al **Processor** de media general.

Cuando el **Processor** de media general recibe todas las medias parciales de los **Groupers**, calcula la media general y le envía la respuesta a cada **Grouper**.

Cada **Grouper** recibe la media general y pasa a realizar un filtrado quedándose con los precios que se encuentran por encima de dicha media. Luego, envía un mensaje por trayecto, con los precios que quedaron, al **Processor** avg & max.

Este **Processor** simplemente calcula la media de precios y el precio máximo de cada trayecto que recibe (recordar que en este punto un mensaje equivale a un trayecto). Por último, envía este resultado al **Tagger** que finalmente lo envía al **Sink** con su tag correspondiente.

Vista de Proceso

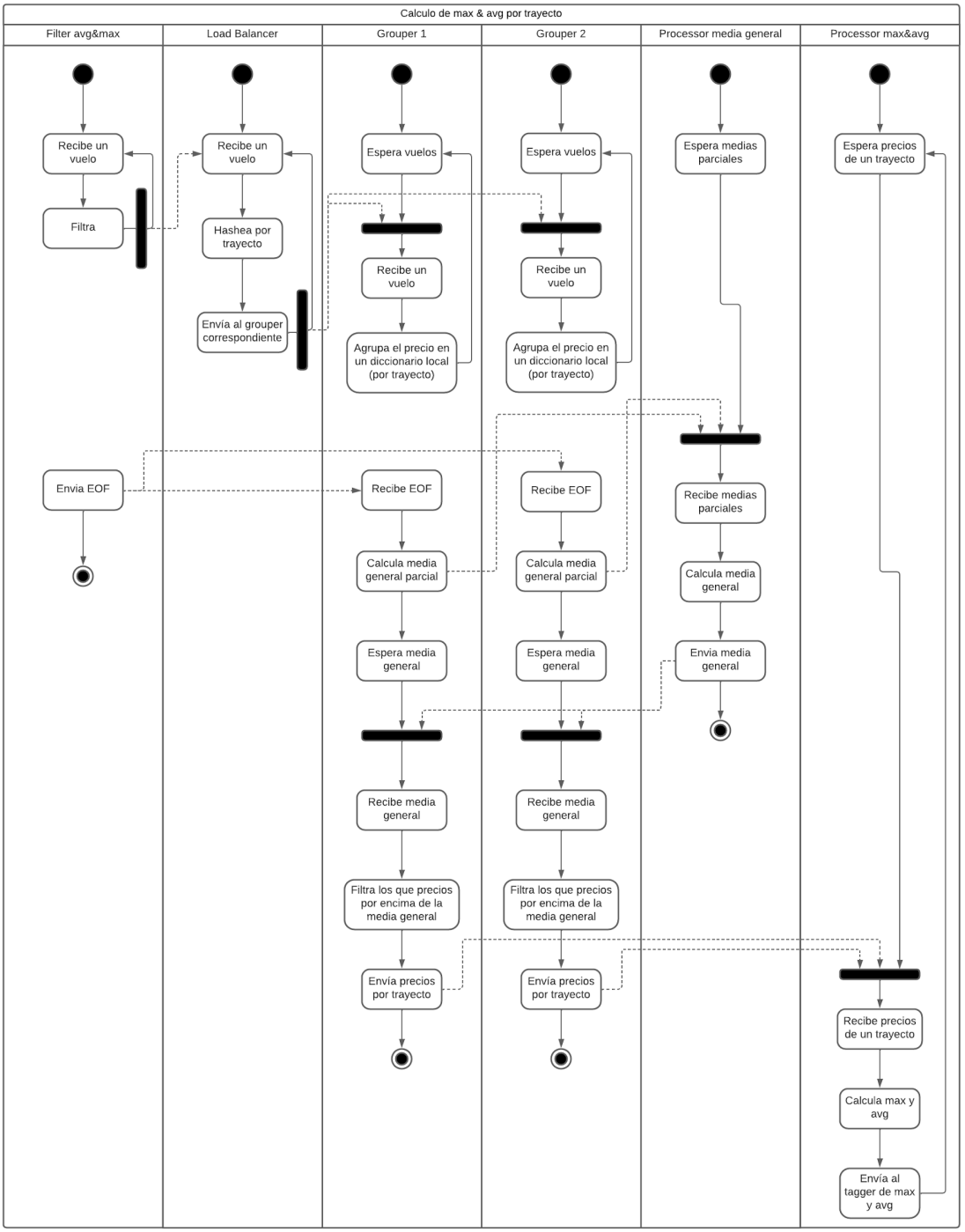

Figura 4.8: Diagrama de actividades de la consulta del promedio y máximo de precios cuyo precio está por encima de la media general (por trayecto)

5. Vista Física

Diagrama de despliegue

Figura 5.1: Diagrama de despliegue de todo el sistema

Diagrama de robustez

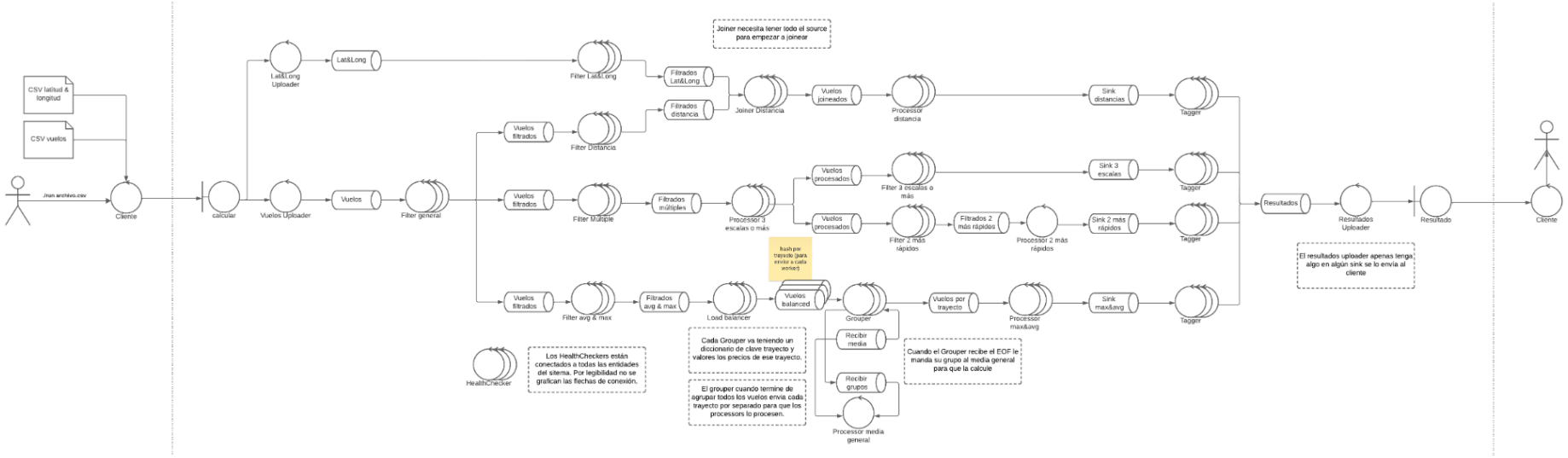

Figura 5.2: Diagrama de robustez de todo el sistema

6. Tolerancia a fallos

Habiendo introducido la arquitectura que se utiliza, pasamos a hablar de cómo el sistema logra ser tolerante a fallos ante diversas situaciones.

Las fallas en un sistema distribuído son un factor muy importante a tener en cuenta a la hora de diseñarlos. Al utilizar **RabbitMQ** como middleware y broker de mensajes, este nos garantiza la **persistencia** de los mensajes ante la caída de una entidad que no llegó a procesar el mensaje en su totalidad. Esto es muy bueno ya que significa que los mensajes no se van a perder hasta recibir su *acknowledgment*, re-encolandose para volver a ser procesados. Sin embargo, esto sólo resuelve uno de los problemas que se generan y hasta introduce otros nuevos, pasemos a hablar de uno de ellos.
Una de las consecuencias más comunes al tener una falla es la creación de un mensaje duplicado.Esto significa que una entidad puede llegar a recibir multiples veces el mismo mensaje, y si el sistema no está preparado para tolerarlo, este mensaje puede causar problemas en el funcionamiento y resultado final del procesamiento.

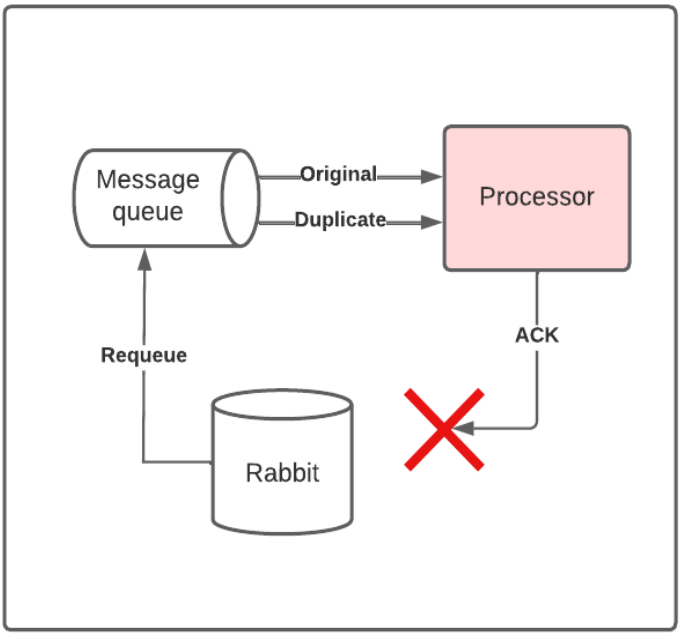

Figura 6.1: RabbitMQ no recibe el ACK y reencola el mensaje

Como apreciamos en la figura 6.1, si el broker no recibe el *acknowledgment* del mensaje que se tomó, este va a volver a encolarse y procesarse, generandose asi un duplicado.

Un mensaje duplicado por sí solo no es un problema, sin embargo el sistema debe estar preparado para recibir estos mensajes y actuar en consecuencia.

7. Processors

Cómo introducimos anteriormente, nuestro sistema cuenta con distintos *processors* para crear, modificar y redirigir los distintos mensajes que llegan al sistema. Los processors se dividen en 2 categorías, los processors **Stateless** y los processors **Stateful**.

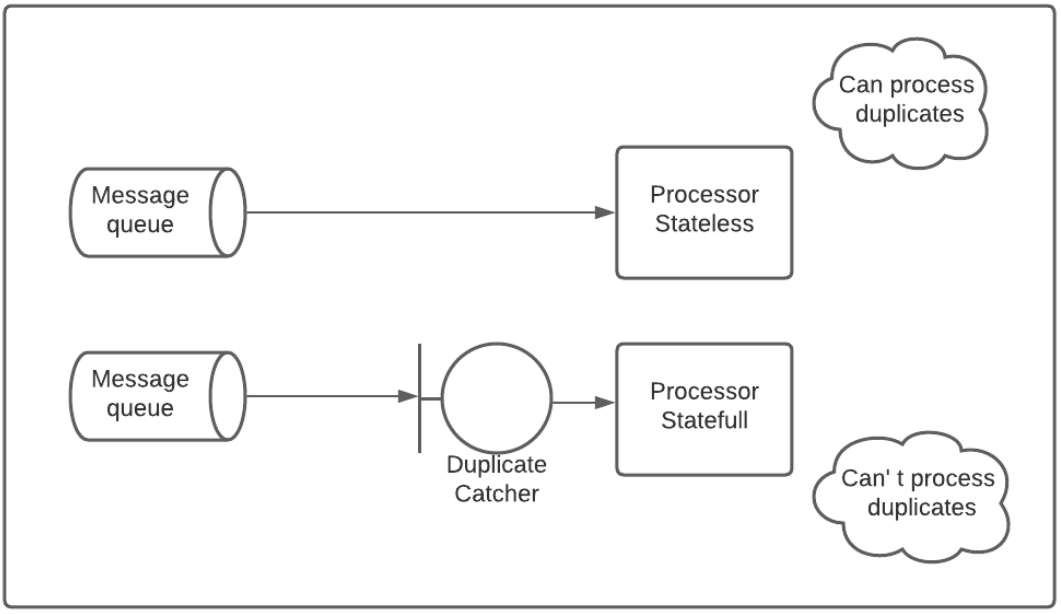

Figura 7.1: los Processors Stateful utilizan un Duplicate Catcher para evitar procesar mensajes duplicados

La diferencia principal entre ambos es la tolerancia que tienen con los mensajes repetidos. Un processor *Stateless* puede tolerar recibir y procesar mensajes duplicados sin que esto lo afecte. Por otro lado, los processors *stateful,* no pueden permitirse recibir duplicados, ya que estos mantienen un **estado interno** el cual es modificado por todos los mensajes entrantes, significando que un mensaje repetido modificaría este estado interno y cambiaría su comportamiento y resultados finales.

Duplicate Catcher

Como podemos ver en la figura 7.1, delante del processor *stateful* está un **Duplicate Catcher**. Esta característica que se le puede agregar a los processors ayuda que los *stateful* nunca procesen un mensaje duplicado, guardando los **IDs** de los mensajes que van llegando y descartando todo mensaje que ya haya sido procesado. Esto es muy útil, sin embargo requiere que una misma réplica, dentro de un grupo de los mismos processors, siempre reciba los mismos mensajes. Esto significa que si un mensaje es reencolado, este se va a volver procesar por la misma réplica. Esto sucede ya que no podemos garantizar que todas las réplicas sepan en todo momento cuáles mensajes fueron procesados y cuáles no. Para lograrlo, en algunas entidades, se hace uso del **Load Balancer**.

Guardando el Estado

Solucionado el problema de los duplicados, aún nos queda el problema de cómo recuperamos el estado interno de los processors *stateful* cuando la entidad se cae y tiene que volver a iniciarse.

Para resolver este problema es necesario que se guarden en disco todos los mensajes que llegaron hasta el momento de la falla. Una vez que el sistema se vuelve a levantar, lo que se hace es volver a ejecutar todos los mensajes guardados y así obtener el mismo estado previo a la falla.

Por el mismo lado, el estado actual del ***duplicate catcher***, también se guarda a disco para poder recuperarlo. La forma detallada de cómo se guarda y cómo recuperamos el estado está explicada en la sección de **Restauración.**

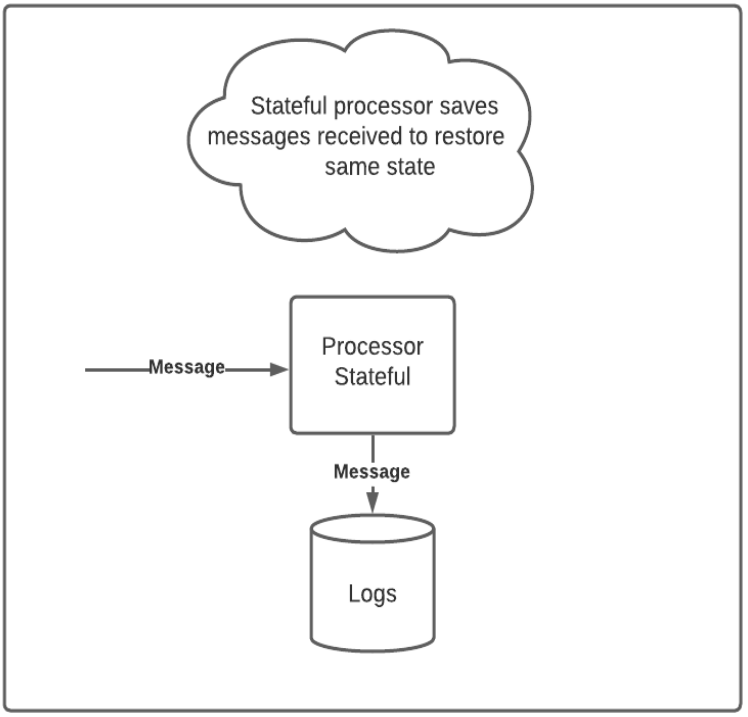

Figura 7.2: los processors stateful guardan los mensajes que van llegando en un log

8. End of File

Como se explicó anteriormente, el **EOF** es un punto clave para la ejecución de nuestro sistema, cuando un processor *stateful* lo recibe y procesa, sabe que ya no quedan más mensajes por recibir y puede, si lo necesita, envíar un resultado final luego de haber procesado todos los mensajes, como es el caso del **Grouper**, entre otros.
Ahora bien, cómo logramos que el **EOF** se procese solo si se recibieron todos los mensajes **reales**, sin importar si se recibieron repetidos y sin cambiar el estado de ninguna entidad.
Vamos a utilizar un algoritmo para detectar los repetidos y poder calcular con certeza si se recibieron correctamente todos los mensajes que la entidad anterior dice haber mandado.

Algoritmo EOF & Detección de Repetidos

Cuando recibimos el EOF, este puede venir con un aviso de repetidos de la fase anterior o también, por un fallo, puede que se haya generado un repetido en esta fase. Es por eso que es necesario un algoritmo para detectar si efectivamente se procesó múltiples veces un mismo mensaje.
Este algoritmo nos va a ayudar a saber si efectivamente recibimos y procesamos todos los mensajes reales antes de haber recibido el EOF, ya que es posible recibirlo prematuramente y no queremos propagarlo ni ejecutarlo hasta haber recibido todo.
Por otro lado nos va a ayudar también a saber cuantos mensajes **reales** nosotros mandamos, para que la próxima entidad que reciba el EOF pueda hacer el algoritmo correctamente.

La idea principal de este algoritmo es que sea **stateless** significando que los mensajes que enviemos o recibamos no van a afectar al estado de la entidad que los reciba, lo único que va a cambiar, a medida que avanza, es el mensaje reencolado y es aquí donde va a estar el estado del algoritmo.

Pasos del algoritmo

Todo comienza con el primer mensaje EOF que recibimos de la etapa anterior, el cual contiene los siguientes parámetros principales: **messages_sent** & **possible_duplicates**.
En las siguientes etapas, estos parámetros pasan a llamarse: **original_messages_sent** & **original_possible_duplicates**.
Con este mensaje la entidad que lo recibe comienza con el algoritmo.

Etapa de Descubrimiento

La primera etapa es la de descubrimiento, consiste en que cada réplica diga cuales son sus posibles repetidos locales. Este mensaje va a pasar por todas las réplicas para así obtener todos los posibles repetidos que hay para revisar. Para saber si un mensaje ya pasó por una réplica, se agrega su ID al mensaje re-encolado.A su vez, cada réplica va agregando cuántos mensajes enviaron y recibieron, para así saber al final si se recibieron todos los mensajes.
Los parámetros principales en esta etapa son: **possible_duplicates**, **messages_sent**, **messages_received**, **replica_id_seen**.

Etapa de Agregación

La segunda etapa es la de agregación, consiste en que cada réplica agregue si procesaron, y cuantas veces, cada mensaje posible repetido.
Al llegar a la última réplica que agrega sus procesados, esta puede contar cuantos repetidos hubo finalmente. Por cada repetido (par del mismo ID) se resta en 1 el número de recibidos y en 1 el de enviados, si es que se envió algo con ese mensaje, y así se puede saber si efectivamente se recibieron todos los mensajes **reales**.
El parámetro principal en esta etapa es: **possible_duplicate_processed_by**

Si el número no coincide significa que todavía quedan mensajes por recibir, por lo tanto la réplica crea el mensaje original y lo re-encola para iniciar el algoritmo de nuevo.
Si el número de recibidos coincide con el enviado en el primer EOF entonces significa que se recibieron con éxito todos los mensajes.

Etapa de Finalización

Cuando se recibieron con éxito todos los mensajes, la última réplica lanza un último mensaje para que todas las réplicas sepan que se recibieron con éxito todos los mensajes.

Una desventaja que se tiene es que puede que se "duplique" el mensaje de EOF si justo hay una falla entre el reencolado y el envío del ACK del mensaje anterior, sin embargo esto no afecta al resultado final, únicamente se va a repetir el procesamiento del EOF y que se propague 2 veces, ya que este algoritmo no modifica el estado interno al ejecutarse.

Es importante que el algoritmo lo ejecuten todos los processors, no solo los **Stateful**, ya que el número de mensajes enviados varía según cada processor, como pasa con el *tres escalas o más*, donde se filtran y envían un conjunto reducido de mensajes, y al ser stateless, tiene que saber si se envió algún mensaje repetido y así calcular el número **real** enviado.

Ejemplos

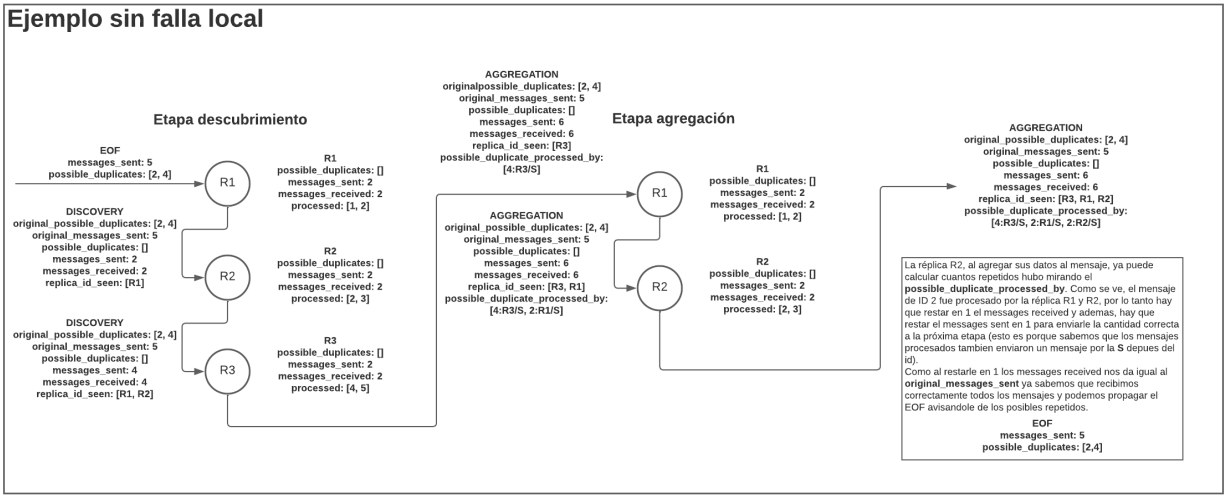

Figura 8.1: ejemplo de algoritmo de EOF sin falla local

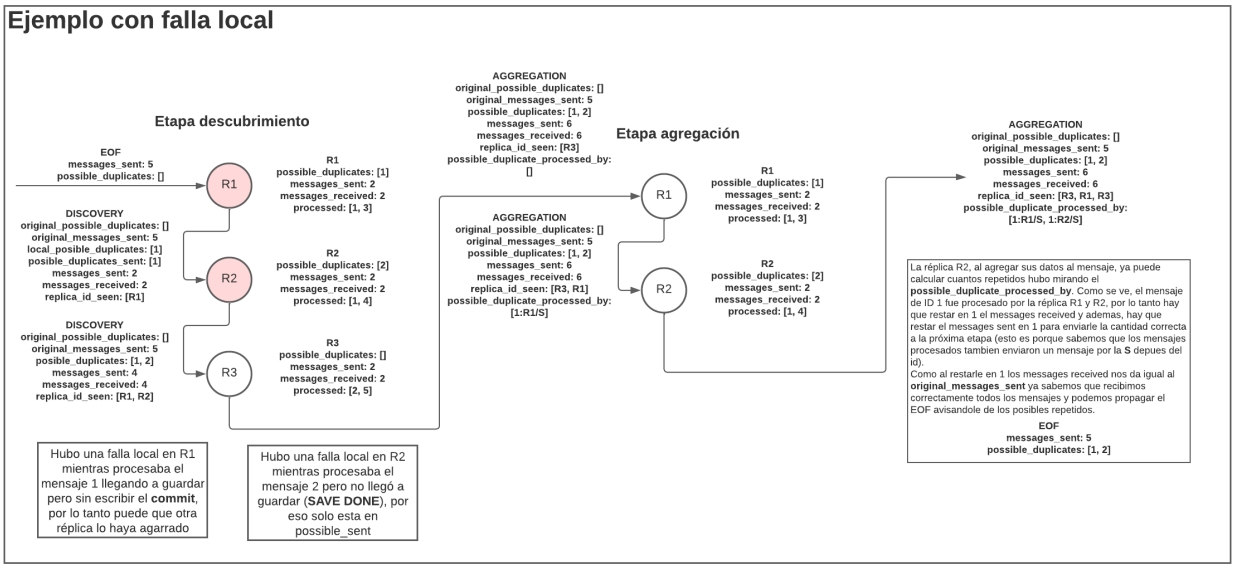

Figura 8.2: ejemplo de algoritmo de EOF con falla local. Las réplicas R1 y R2, pintadas de rojo, son las que fallaron.

9. Restauración

Para la restauración, en caso de alguna falla en el sistema, se utilizan tres tipos de logs que guardan (en disco) información respecto al estado del mismo:

Tipos de Logs

- **Communication Log**: guarda la información correspondiente al estado actual del *communication*, usado en el algoritmo de EOF (mensajes enviados, posibles duplicados, etc.). Se genera un solo archivo de este tipo por entidad (réplica), compartido por todos los clientes.

```
START <message_id> / <client_id>
SENT <message_id> / <client_id>
SAVE BEGIN
<json of save information>
SAVE DONE
COMMIT <message_id> / <client_id>
```

Listing 9.1: estructura general de un registro en el Communication Log

- **Stateful Processor Log**: contiene los mensajes que fueron procesados por la entidad. Sólo lo utilizan los processors stateful. En los processors que lo utilizan, se cuenta con un log de estos por cliente (para separar los mensajes correspondientes a cada uno).

```
<message_id>/<json of save information>
```

Listing 9.2: estructura general de un registro en el Stateful Processor Log

- **Duplicate Catcher Log**: este log mantiene un registro de los ids de mensajes que ya fueron procesados por la entidad. Las entidades que implementan un *duplicate catcher* tienen uno de estos logs por cliente.

```
<message_id>
```

Listing 9.3: estructura general de un registro en el Duplicate Catcher Log

Algoritmo de Restauración

Cuando una entidad se levanta comienza su algoritmo de restauración.Lo primero que hace una entidad es leer de su archivo de logs para poder restaurarse. Siempre lee el log de abajo para arriba, o sea empezando por lo último logeado.

El algoritmo busca palabras clave para guiarse. Cuando lo lee puede encontrarse con uno de los siguientes escenarios:

1. Se encuentra primero con un **COMMIT**: significa que el último mensaje fue ejecutado en su totalidad, por lo tanto, restaura a partir del START de este último mensaje y continúa con su ejecución.

```
START 184 / 10
SENT 184 / 10
SAVE BEGIN
{"communication": { "messages_received": {"10": 100},
"messages_sent": {"10": 50}, "possible_duplicates": {} }}
SAVE DONE
COMMIT 184 / 10
```

Listing 9.4: ejemplo de un Communication Log en el cual se llegó a hacer el COMMIT del mensaje 184 del cliente 10

2. Se encuentra primero con un **SAVE DONE**: significa que el último mensaje procesado se guardó correctamente, sin embargo puede que se haya mandado el ACK a rabbit, por lo tanto se restaura desde el SAVE BEGIN del mensaje fallido y además (este mismo mensaje que está en el START) se agrega a los posibles repetidos.

3. Se encuentra primero con un **SAVE BEGIN**, **SENT** o **START**: significa que el mensaje falló antes de terminar de guardarse, por lo tanto, es necesario buscar el último COMMIT anterior y restaurar desde ahí. También se agrega el mensaje fallido a los posibles repetidos.

4. No se encuentra con ninguna palabra clave: significa que es su primera corrida, por lo tanto empieza de cero.

Los processors stateful, además de restaurar el Communication Log, también leen el Stateful Processor Log y vuelven a procesar todos los mensajes que encuentran para volver al estado en el que se encontraba el processor antes de la caída.

Por último, el Duplicate Catcher Log es leído por los processors que utilicen un DuplicateCatcher para restaurar en memoria los ids de los mensajes que ya fueron procesados.

Clases

Las clases encargadas de guardar, obtener y buscar registros en los logs anteriormente mencionados son las siguientes:

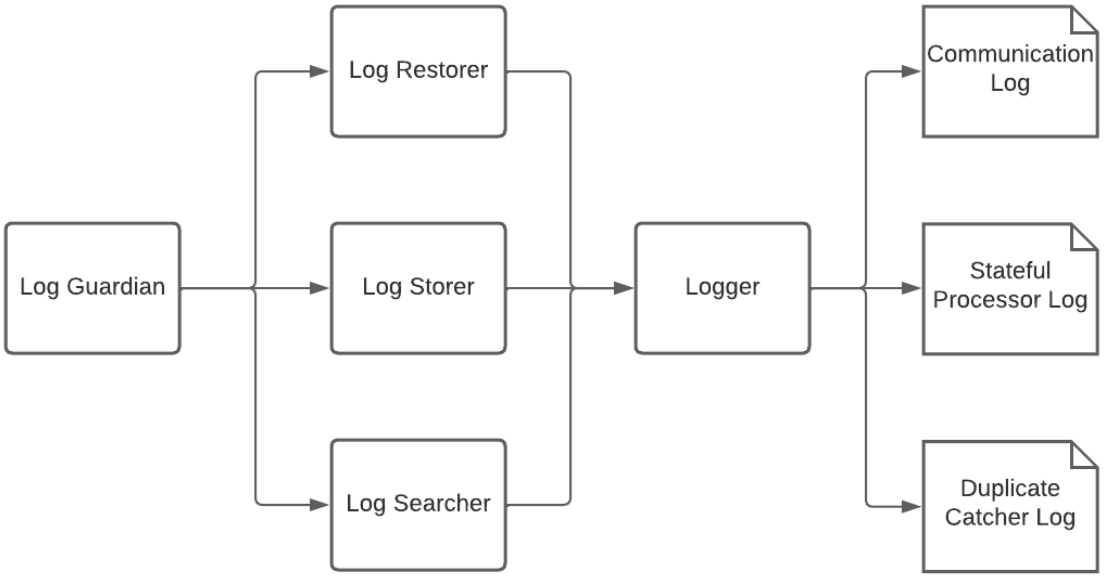

Figura 9.1: clases involucradas en el proceso de guardado, restaurado y búsqueda de logs

- **Log Guardian**: interfaz que expone métodos para interactuar con el Log Restorer, el Log Storer y el Log Searcher.
- **Log Restorer**: utiliza el Logger para obtener los datos del CommunicationLog y agrega los posibles duplicados según el caso de restauración que corresponda.
- **Log Storer**: se utiliza para coordinar el orden de guardado en los distintos tipos de logs.
- **Log Searcher**: funciona como capa de abstracción, convirtiendo los strings devueltos por el Logger a una clase ProcessedMessage.
- **Logger**: esta es la clase que interactúa directamente con los archivos de logs. Expone métodos para guardar en cada log, obtener los datos necesarios para la restauración según lo que encuentra en el CommunicationLog y para buscar mensajes repetidos.

10. Cliente & Servidor

Los **Clients** son los que envían el archivo de vuelos y el archivo de aeropuertos al Server. Estos se conectan mediante sockets.

El **Server** permite una cantidad máxima (configurable) de **Clients** en simultáneo. Para lograr esto se utiliza un semáforo en el Server, si ya se llegó al límite máximo de **Clients** activos, el próximo **Client** que se conecte al **Server** quedará esperando en el semáforo hasta que algún **Client** activo se desconecte.

Protocolo

El protocolo utilizado entre el **Client** y el **Server** consiste de los siguientes mensajes:

- **Announce**(client_id)
- **AnnounceACK**
- **ClientProtocol**(message_id, protocol_type, content)
- **ACK**(message_id, protocol_type, content)
- **Result**(tag_id, message_id, result)
- **ResultACK**
- **ResultEOF**(tag_id, messages_sent)

Cuando el **Client** logra conectarse con el Server lo primero que envía es un mensaje Announce con el cual indica cuál es su client_id. El **Client** reenvía este mensaje, hasta que recibe un AnnounceACK del **Server**.

Una vez recibido el AnnounceACK, el **Client** puede comenzar a enviar los datos de los archivos de vuelos y de aeropuertos. Para esto utiliza el mensaje ClientProtocol, indicando el message_id, el protocol_type (es decir, el tipo de archivo que se está enviando: *Flights* o *Airports*) y el contenido. Hasta que el **Client** no reciba el ACK del mensaje enviado, lo reenvía.

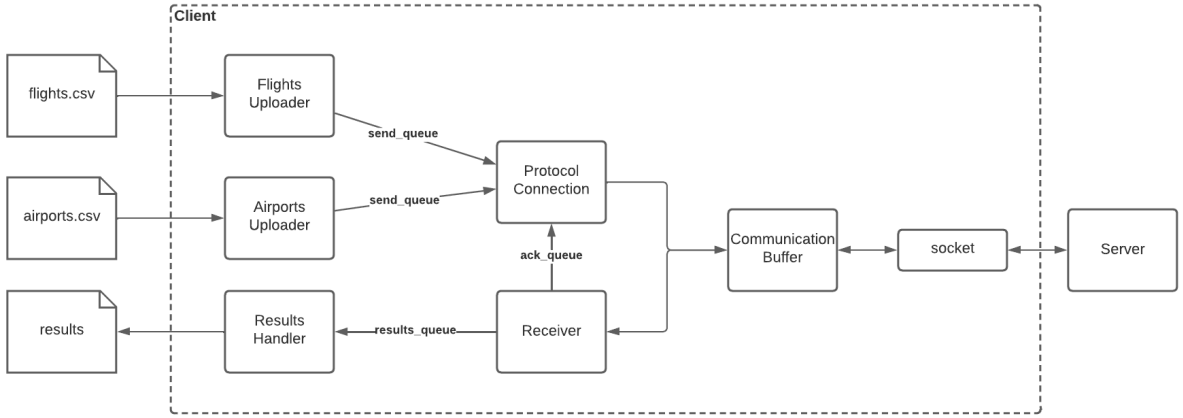

Figura 10.1: estructura interna del Client

Para los resultados el **Server** envía un mensaje Result que contiene el tag_id (esto indica la query a la que corresponde este resultado), el message_id (un id incremental que el Server

le asigna a este resultado de esta query) y el contenido del resultado. El **Client** debe enviar un ResultACK una vez recibido el resultado, esto se hace para garantizar que el **Client** lo recibió. En el caso que el **Server** no reciba este ResultACK o sufra una caída, no se hace ACK al resultado obtenido por el **Server** de la cola de resultados de rabbit y entonces es reencolado para que pueda ser envíado en un momento posterior.

El **Client** detecta que el **Server** se cayó si obtiene un error al intentar recibir o enviar un mensaje por el socket. Una vez detectada la caída del **Server**, el **Client** intenta reconectarse. Cuando logra conectarse al **Server** vuelve a envíar el mensaje Announce para indicar su client_id y reenvía el mensaje perdido (si es que no se recibió un ACK a un mensaje envíado). Luego continúa con la ejecución normal.

Finalmente, para que el **Client** pueda desconectarse necesita saber que ya tiene todos los resultados, por esto el **Server** envía el mensaje ResultEOF con el tag_id y messages_sent (es decir, la cantidad de mensajes que corresponden a los resultados de esta query). El **Client** lleva una cuenta de cuántos mensajes recibió para cada query, si recibe un ResultEOF pero todavía no recibió la cantidad de resultados indicado por el messages_sent, entonces se queda esperando hasta recibir todos. Cuando el **Client** recibe todos los resultados, se desconecta del **Server**.

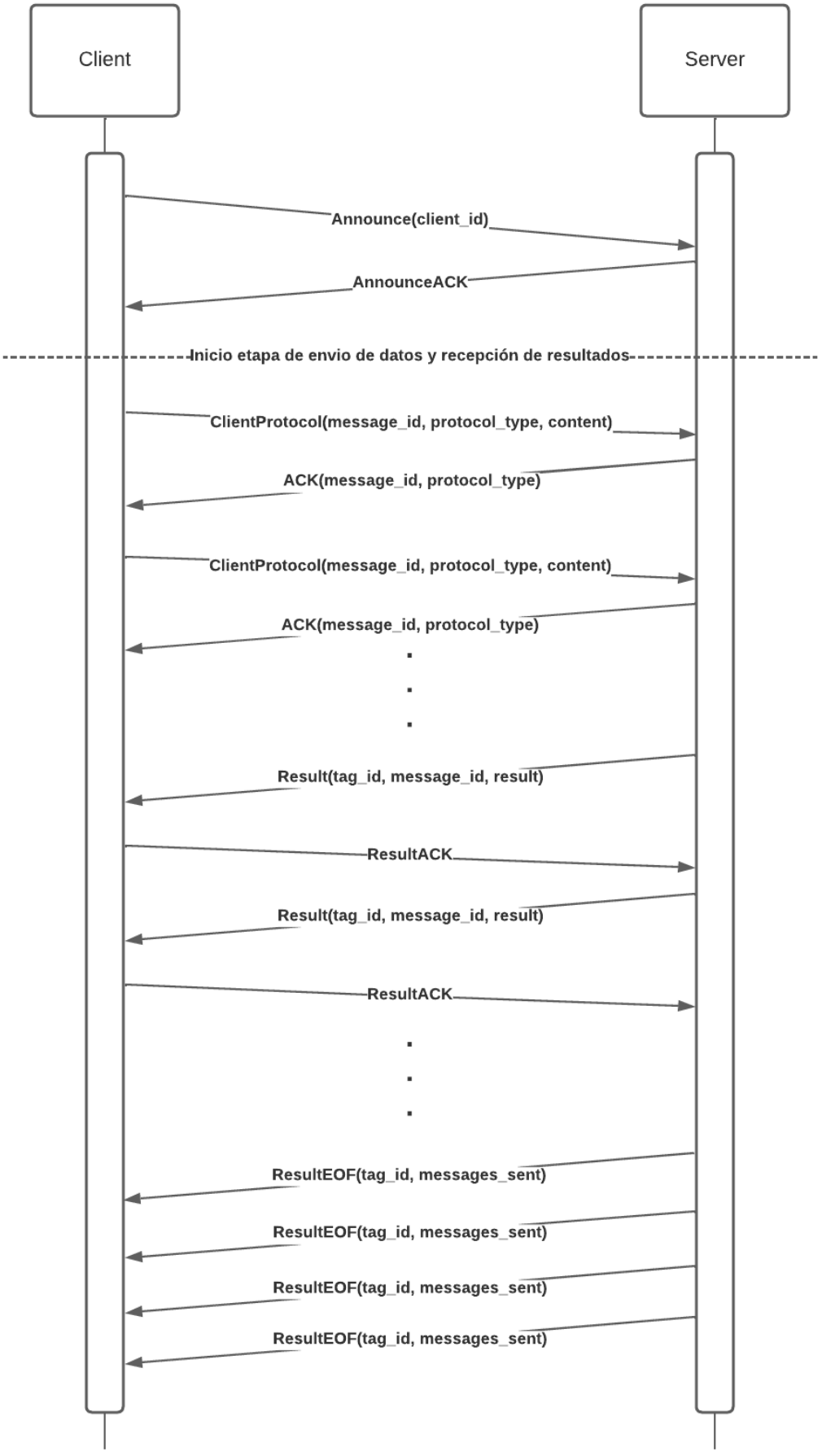

*Figura 10.2: mensajes del protocolo utilizado entre el **Client** y el **Server***

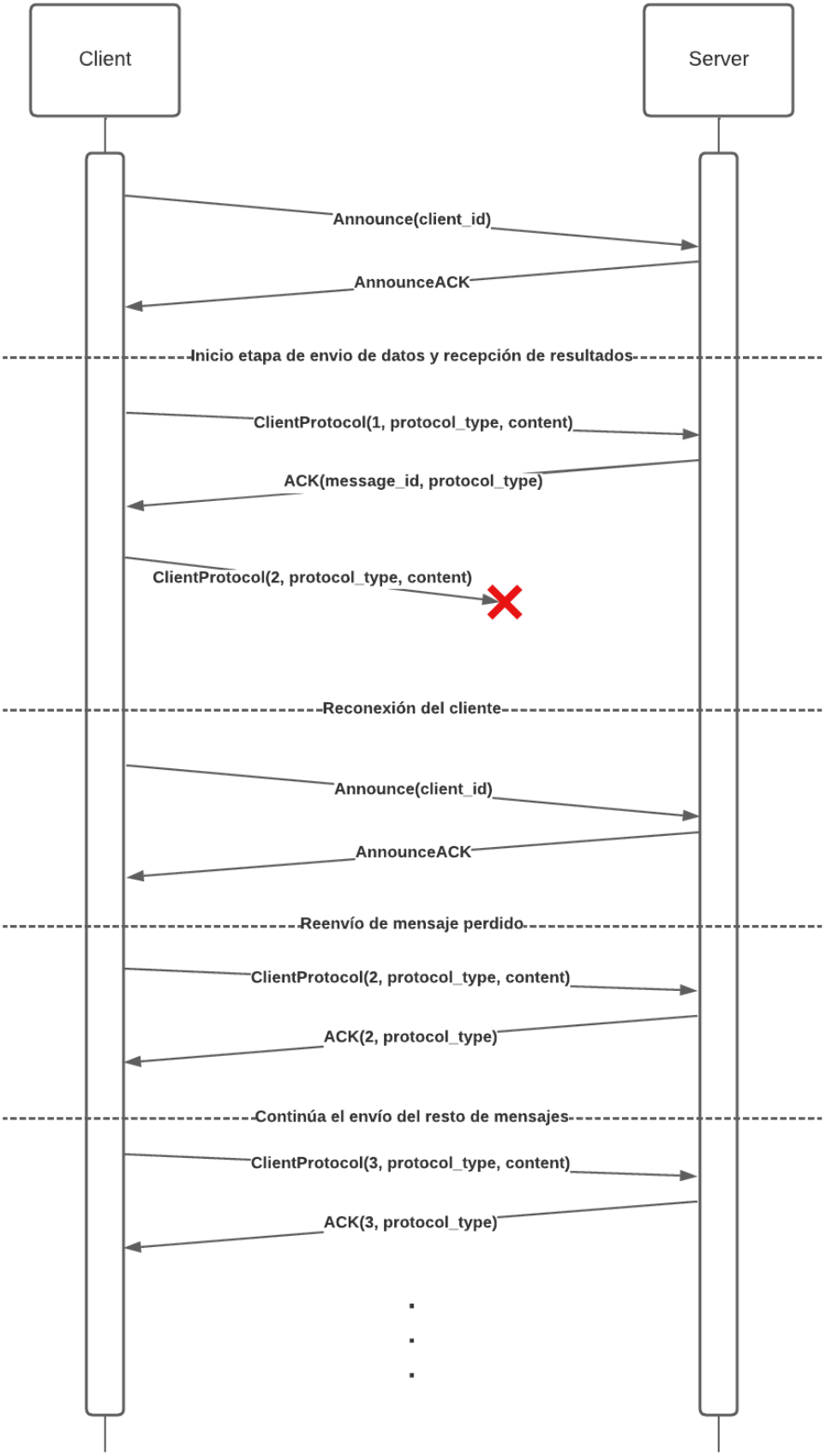

*Figura 10.3: ejemplo de caída del **Server** cuando el **Client** envía un mensaje ClientProtocol. Se muestra que el **Client** reenvía el Announce indicando su client_id y luego reenvía el mensaje que no recibió el ACK.*

Múltiples clientes

El sistema soporta la ejecución de querys de múltiples clientes simultáneamente. Para esto se implementó lo siguiente:

Conexión del Cliente

Como ya fue mencionado, el server mantiene un semáforo para garantizar una máxima cantidad de clientes conectados al mismo tiempo. Cuando un cliente se conecta pasado el límite, el mismo queda bloqueado en el semáforo y recién continúa su ejecución cuando otro cliente se desconecta.

Resultados

Para los resultados el servidor tiene el proceso ResultsListener que escucha de la cola de rabbit en común que se usa para todos los resultados. Este proceso redirige cada resultado al tópico de resultados de cada cliente en particular. Luego se tiene un proceso ResultsUploader por cada cliente, esté lee de esa cola de resultados y envía el resultado al cliente a través del socket.

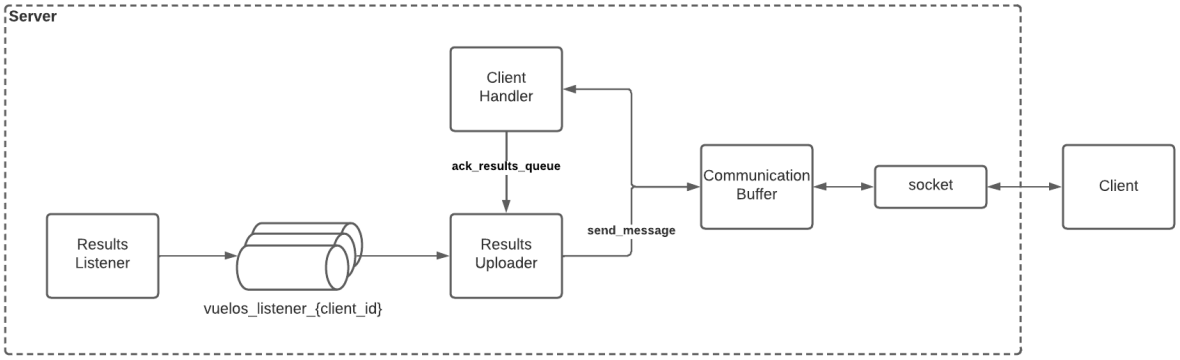

Figura 10.4: estructura interna del Server

Guardando los resultados

Como se sabe, el acceso a disco es muy costoso y requiere de mucho tiempo de espera.
Además, la cantidad de resultados que le llegan al cliente es elevada y no queremos que se gaste mucho espacio en memoria en resultados a la espera de poder guardarse.
Es por eso que para mejorar la optimización del procesamiento de mensajes se creó una mejora a la hora de guardar los resultados que llegan al cliente.

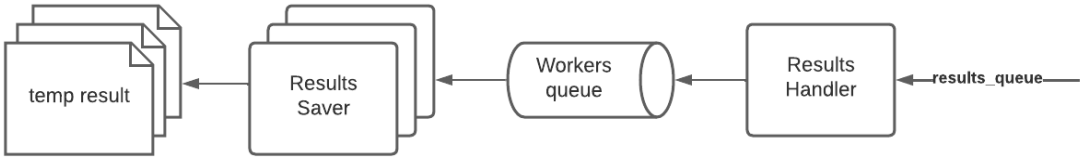

Figura 12.1: los Results Saver generan archivos de resultados temporales

Por cada resultado que le llega al **Results Handler** este se lo pasa a una cola de workers que su trabajo consiste en guardar paralelamente cada resultado en un archivo temporal individual. Esto se hace para que no haya ningún problema de concurrencia ya que se usan archivos independientes.

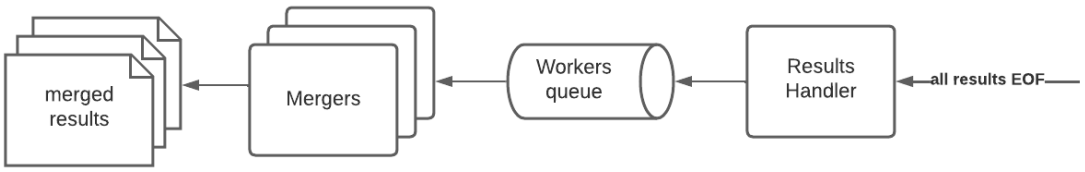

Figura 12.2: los Mergers usan los archivos de resultados temporales para generar los archivos de resultados finales

Cuando se reciben todos los **ResultEOF**, el *results handler* finaliza a los *savers* para dar inicio a los **mergers**, encargados de tomar todos los archivos temporales de cada query y juntarlos en un único archivo por cada query. Obteniendo así los resultados finales.

Entidades

Cada entidad del sistema cuenta con la clase Connection que funciona de proxy entre el módulo Communication y el Processor en sí (que es el que tiene la lógica de negocio de la entidad en particular). Este Connection, además, es el encargado de mantener un diccionario (“client_id” -> Processor) que mapea un client_id con una instancia de la clase Processor, de esta manera dicha instancia sólo maneja el estado de ese cliente. Cuando un mensaje llega al Connection, este revisa en el diccionario a cuál instancia de Processor enviarlo y se lo entrega.

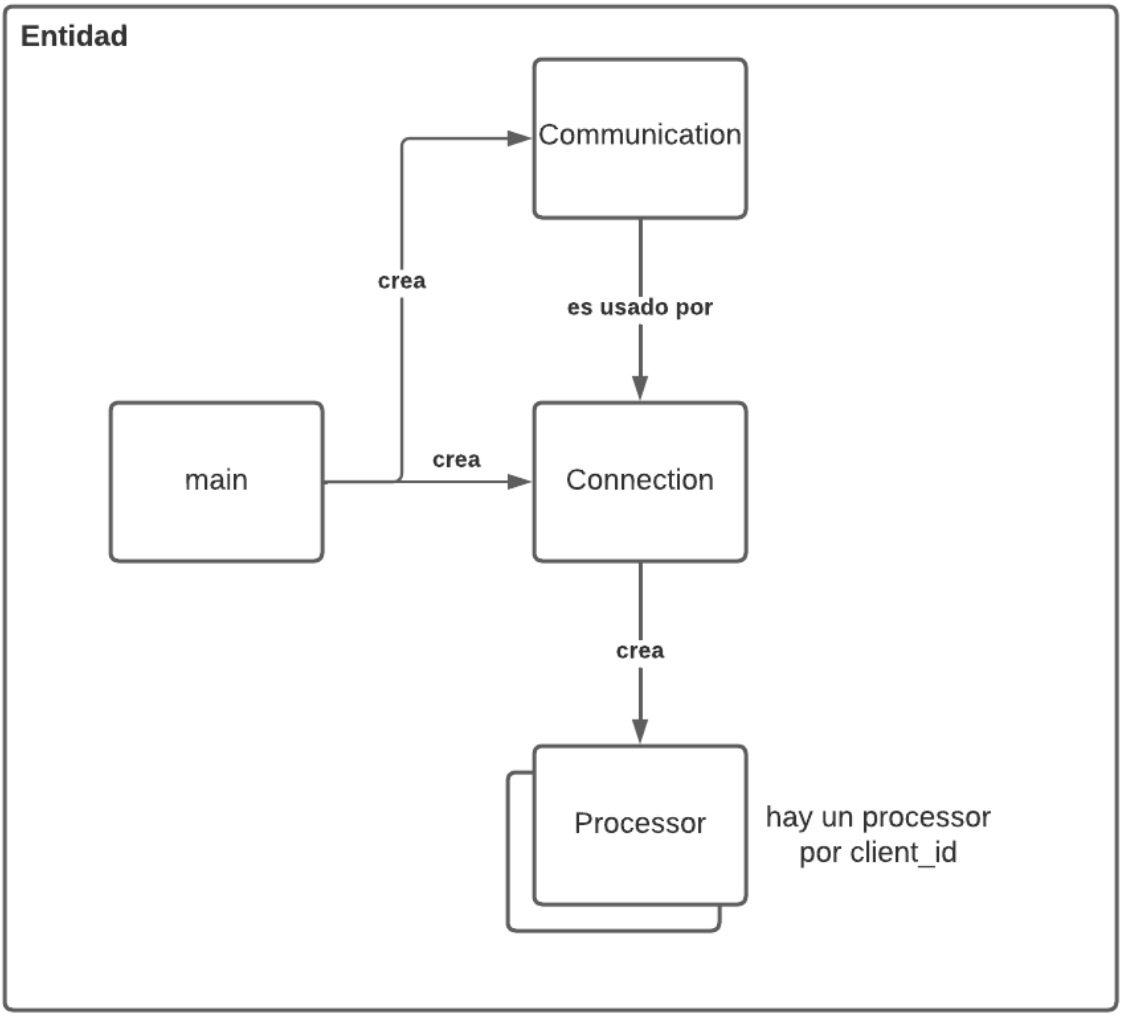

Figura 10.5: clases involucradas en el proceso de un mensaje de una entidad

11. Health Checker

Para detectar que una entidad está caída y reiniciarla se cuenta con una entidad particular llamada **HealthChecker**, que mantiene una conexión por socket con el resto de las entidades y réplicas del sistema, cada una en un proceso separado.

Protocolo

El **HealthChecker** envía un mensaje "CHECK" y queda esperando a un timeout configurable (por ej, 20 segs). Luego cada entidad responde con "OK" si está funcionando correctamente. En el caso que no se reciba un "OK" de esa entidad luego del timeout, el **HealthChecker** asume que está caída y la intenta levantar (haciendo `docker restart <entidad>`).Puede hacer esto porque se monta el socket de docker (/var/run.docker.sock) como volumen en el docker compose de esta entidad.

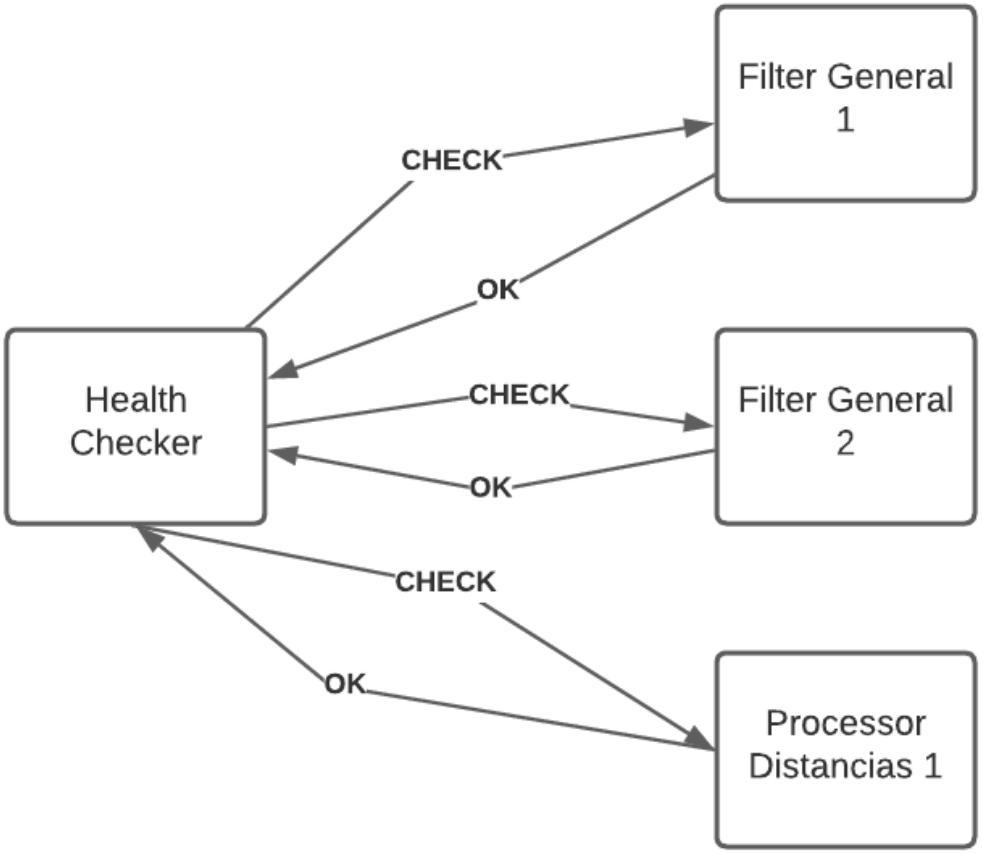

*Figura 11.1: ejemplo del protocolo de un **HealthChecker** conectado a dos réplicas del **FilterGeneral** y una réplica del **ProcessorDistancias***

Replicación

Las réplicas del **HealthChecker** están conectadas en forma de anillo. Si una no recibe el "OK", levanta a la que está caída como si fuese cualquier otra entidad del sistema. Además,

para evitar condiciones de carrera en el reinicio de las entidades caídas, cada réplica se encarga de monitorear sólo un subconjunto de entidades.

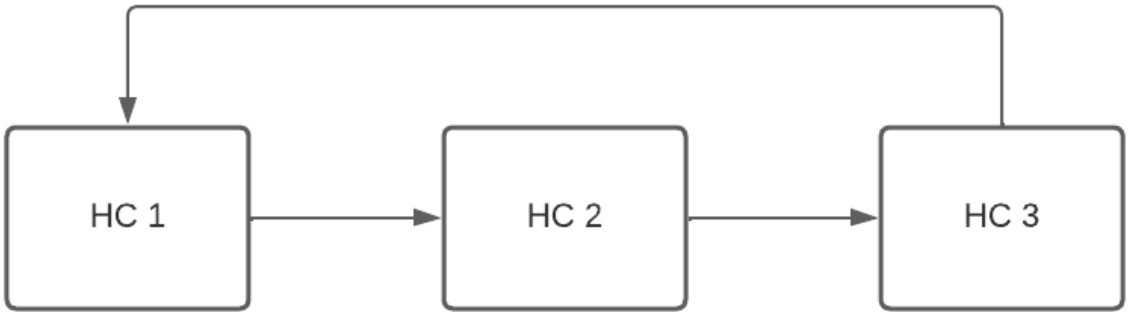

*Figura 11.2: tres réplicas del **HealthChecker** conectadas en forma de anillo*

12. Ejecución

Para probar el funcionamiento del sistema, se ejecutó el archivo de 31GB para analizar 6 meses de registros de precios de vuelos, junto al archivo de coordenadas de aeropuertos.
Se corrió con 2 clientes en simultáneo enviando ambos los mismos archivos.
Se completó todo el procesamiento en 2 horas y 10 minutos, a continuación mostramos distintas imágenes que reflejan la ejecución.

Figura 12.1: mensajes encolados en RabbitMQ

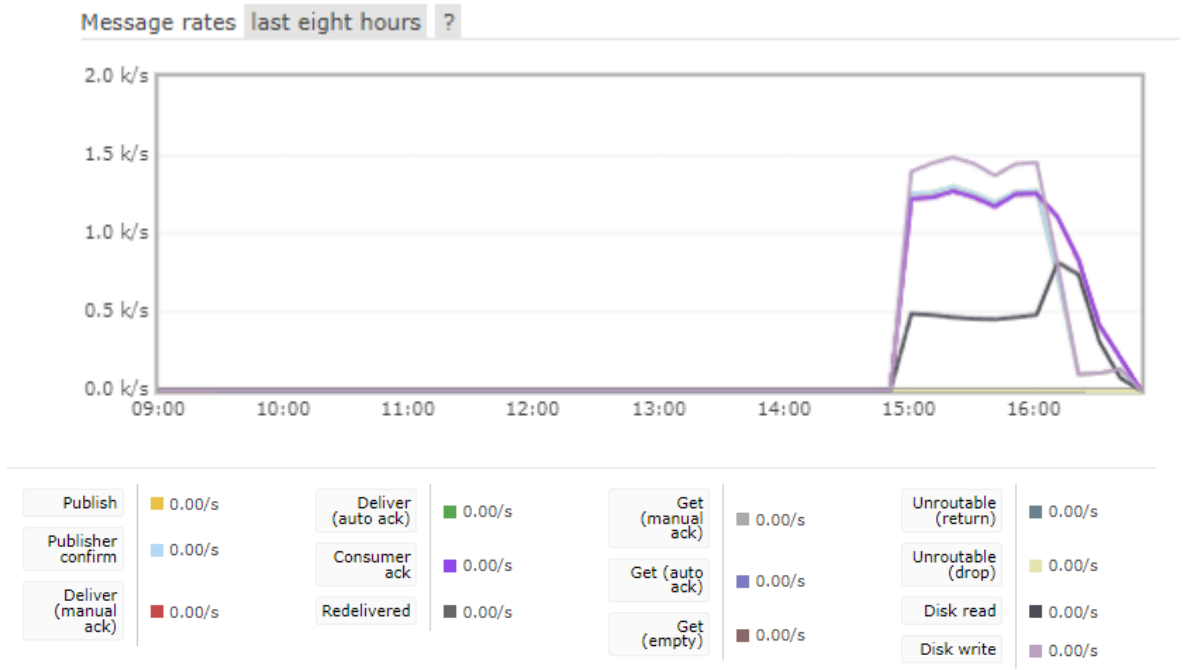

Figura 12.2: tasa de mensajes en RabbitMQ

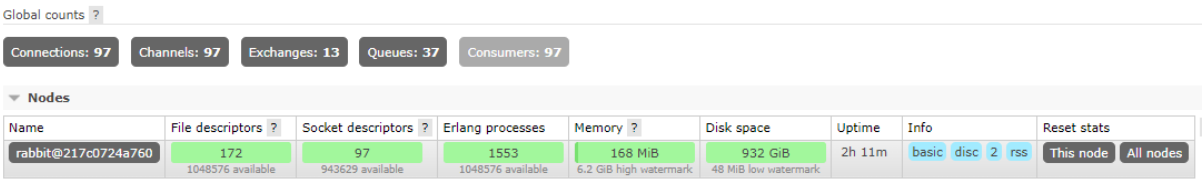

Figura 12.3: tiempo de ejecución reportado por RabbitMQ

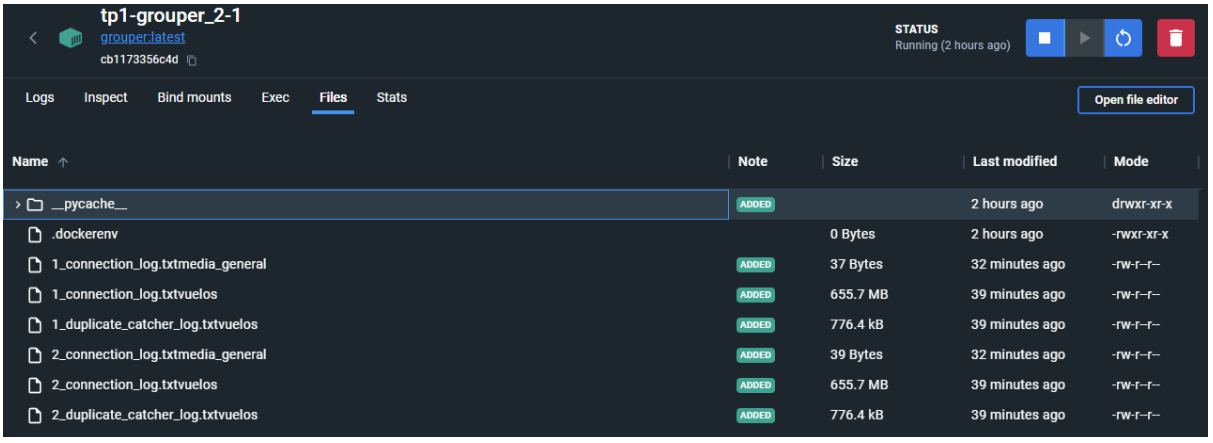

Figura 12.4: tamaño de los logs dentro de un Grouper

```
joaquin@DESKTOP-1UM22AF:/mnt/c/Github/tp-distribuidos$ docker system df
TYPE            TOTAL     ACTIVE    SIZE      RECLAIMABLE
Images          252       14        2.332GB   1.984GB (85%)
Containers      97        97        14.64GB   0B (0%)
Local Volumes   57        1         2.405GB   2.382GB (99%)
Build Cache     567       0         308kB     308kB
```

Figura 12.5: espacio en disco utilizado para la ejecución

```
joaquin@DESKTOP-1UM22AF:/mnt/c/Github/tp-distribuidos$ wc results/client_1/20231207175510_distancias.txt -l
4115780 results/client_1/20231207175510_distancias.txt
joaquin@DESKTOP-1UM22AF:/mnt/c/Github/tp-distribuidos$ wc results/client_1/20231207175510_dos_mas_rapidos.txt -l
245 results/client_1/20231207175510_dos_mas_rapidos.txt
joaquin@DESKTOP-1UM22AF:/mnt/c/Github/tp-distribuidos$ wc results/client_1/20231207175510_max_avg.txt -l
234 results/client_1/20231207175510_max_avg.txt
joaquin@DESKTOP-1UM22AF:/mnt/c/Github/tp-distribuidos$ wc results/client_1/20231207175510_tres_escalas.txt -l
199910 results/client_1/20231207175510_tres_escalas.txt
joaquin@DESKTOP-1UM22AF:/mnt/c/Github/tp-distribuidos$
joaquin@DESKTOP-1UM22AF:/mnt/c/Github/tp-distribuidos$
joaquin@DESKTOP-1UM22AF:/mnt/c/Github/tp-distribuidos$ wc results/client_2/20231207175510_distancias.txt -l
4115780 results/client_2/20231207175510_distancias.txt
joaquin@DESKTOP-1UM22AF:/mnt/c/Github/tp-distribuidos$ wc results/client_2/20231207175510_dos_mas_rapidos.txt -l
245 results/client_2/20231207175510_dos_mas_rapidos.txt
joaquin@DESKTOP-1UM22AF:/mnt/c/Github/tp-distribuidos$ wc results/client_2/20231207175510_max_avg.txt -l
234 results/client_2/20231207175510_max_avg.txt
joaquin@DESKTOP-1UM22AF:/mnt/c/Github/tp-distribuidos$ wc results/client_2/20231207175510_tres_escalas.txt -l
199910 results/client_2/20231207175510_tres_escalas.txt
```

Figura 12.6: cantidad de líneas de cada archivo de resultados

13. Conclusiones y aprendizajes

- Pudimos poner en práctica los conceptos vistos en la materia, principalmente sobre escalabilidad y tolerancia a fallos a un problema que podría darse en un ambiente real. Aprendimos sobre los beneficios y desafíos que trae querer implementar una arquitectura de este estilo.
- El análisis previo y feedback del profesor antes de empezar a desarrollar la arquitectura ayudó mucho a la organización del proyecto y de la separación de tareas.
- La metodología de *pair programming* adoptada para casi la totalidad de las tareas que compusieron al desarrollo integral del proyecto fue muy beneficiosa para avanzar rápido y con una primera revisión de código inmediata por parte del otro compañero.
- Por último, notamos como una decisión de diseño que se tomó al principio, como es la del EOF, condiciona completamente cómo se encararon las soluciones para que el sistema sea tolerante a fallos y a la vez compatible con la arquitectura previamente diseñada.

Puntos a mejorar

- **Eliminado de clientes ya procesados:**
  En el sistema actual, luego de una ejecución, no se están eliminando los archivos de log pertenecientes a clientes ya procesados. Una mejora que se podría implementar

es la de un mensaje especial que se vuelva a propagar por todo el sistema avisando sobre la desconexión de un cliente y posibilidad de eliminar todo lo relacionado a él.

- **Mejoras en cómo se guardan los logs**:
  Actualmente los logs se guardan en texto plano. Para una optimización de performance y de espacio en disco se podría implementar una serialización especial que termine guardando los datos en binario.

- **Paralelizar el procesamiento de los mensajes en el Connection**:
  Los mensajes que recibe el Connection los pasa secuencialmente a cada Processor encargado de un cliente en particular, esto se podría pensar en paralelizar por cliente, aunque habría que modificar el acceso a los logs.