Sistemas Distribuidos I (75.74)

Facultad de Ingeniería
Universidad de Buenos Aires

Trabajo Práctico 2 Tolerancia a Fallos - Flights Optimizer

Nombre y Apellido	Padrón	Email
Amezqueta, Alfonso Javier	94732	alfonso71292@gmail.com
Pareja, Facundo Jose	99719	fpareja@fi.uba.ar
Pérez Andrade, Violeta	101456	viperez@fi.uba.ar

Introducción

El presente modelo 4+1 pretende presentar la arquitectura elegida para el TP2 de la materia Sistemas Distribuidos. El objetivo del mismo es responder una serie de consultas a varios clientes a partir de una gran cantidad de datos streameada de Internet, con especial atención a una arquitectura distribuida, escalable y flexible, y con tolerancia a posibles fallos repentinos de los nodos. En este informe se hace especial énfasis en las diferencias de implementación respecto del TP1. Cuando un aspecto es igual entre ambos, se lo menciona así para que pueda ser consultado en el informe anterior.

Vista física

Introducción a la arquitectura

El diagrama ilustra la representación gráfica de los diversos nodos del sistema y la forma en que interactúan entre sí. A excepción del cliente, se sigue la pauta de que cada nodo se encarga de procesar la información de entrada y posteriormente registra los resultados en

una cola. Algunos nodos son procesos en vez de sistemas individuales en sí mismos, y los mismos son representados agrupados junto con los otros procesos del mismo sistema. La tolerancia a fallos se explica en detalle en la sección siguiente.
A continuación se procede a explicar la responsabilidad de cada proceso (hay nodos que pueden tener dos o más de los procesos mencionados, en los diagrama de despliegue y robustez se explica cómo se distribuyen estos procesos entre los nodo):

- Client: envía los registros de los archivos de vuelos y de aeropuertos de a uno al servidor además de recibir los resultados vía sockets TCP. Para esto, existen internamente unas clases *listener_client* y un *sender_client* que encapsulan este comportamiento. Los clientes se autoetiquetan según una variable de entorno ID en el campo *client_id*.
- Server: se encarga de recibir a los clientes a través de un socket TCP, y luego despacha el manejo del mismo a otro proceso, encapsulado mediante una clase ClientHandler. La misma maneja el resto de la comunicación con el cliente, o sea propagación de los mensajes enviados por el cliente a una cola para ser consumidos por los nodos. Cada proceso etiqueta individualmente los mensajes con un *message_id*. Esta numeración se reinicia con el fin del archivo de registros de aeropuertos.
- initial_column_cleaner: filtra todas las columnas superfluas (se mantienen *legID, client_id, message_id, startingAirport, destinationAirport, totalFare, totalTravelDistance, baseFare, travelDuration, segmentsArrivalAirportCode*).
- filter_by_three_stopovers: se encarga de filtrar los vuelos recibidos por una cola de entrada de acuerdo a si tienen tres escalas o más. Además filtra las columnas innecesarias para el resto de la consulta: (se mantienen *legID, client_id, message_id, startingAirport, destinationAirport, totalFare)*.
- query_2_column_filter: filtra todas las columnas superfluas para el resto de la consulta (se mantienen op_code, *legID*, *client_id*, *message_id*, *startingAirport*, *destinationAirport*, *totalTravelDistance*, *segmentsArrivalAirportCode*). Es el único nodo luego de initial_column_cleaner que no rechaza los registros de aeropuertos, y mantiene las columnas *opcode*, *message_id*, *client_id*, *Airport Code*, *Latitude* y *Longitude*.
- query_5_column_filter: filtra las columnas superfluas para el resto de la consulta (se mantienen (*op_code*, *client_id*, *startingAirport*, *baseFare*, *message_id*)
- group_by: se encargan de hashear elementos recibidos por una cola de entrada de acuerdo a un campo configurable y enviarlos al reducer. A diferencia del TP1, se presenta aquí la necesidad (ver sección *Recuperación de fallas*) de agregar un nodo que separe los elementos utilizando el campo *message_id*. Estos se llaman group_by_message_id.
- reducer_group_by: recibe una serie de campos y se encarga de agruparlos de acuerdo a un campo configurable y reducirlos por una función definida.
- dictionary_creator: recibe únicamente registros de aeropuertos y crea un diccionario con el formato {aeropuerto: (coordenadas)}. Una vez recibidos todos los registros, envía el mismo al distance calculator a través de una pipe.
- distance_calculator: a partir del diccionario recibido del dictionary creator, verifica los registros que cumplen con la condición pedida y los propaga. Además, remueve las columnas innecesarias (solo quedan *client_id, message_id, legID, startingAirport, destinationAirport y totalTravelDistance*) antes de enviar la respuesta

- avg_calculator: calcula la sumatoria del precio de todos los vuelos que recibe así como la cantidad de vuelos recibidos. Con estos dos resultados arma un mensaje con un resultado parcial que propaga para que se pueda generar el resultado final.
- final_avg_calculator: a partir de los resultados parciales provenientes de cada avg calculator obtiene el resultado final y lo comunica a *filter_by_average*.
- filter_by_average: baja los registros a disco mientras aguarda el cálculo final del promedio proveniente del *avg_final_calculator*. Una vez este está disponible propaga cada línea que supera el promedio.
- query_handler: consume mensajes de las distintas colas donde los nodos dejaron los resultados y propagarlos en una única cola
- result_handler: recibe los resultados mediante una cola de entrada, los encodea y los devuelve al cliente correspondiente. También permite el filtrado de mensajes repetidos mediante dos tipos de etiquetas identificatorias distintas (ver *Recuperación de fallas*).
- heartbeat_listener: se encarga de escuchar heartbeats que le llegan. Si descubre a un nodo que no le envía un heartbeat entonces procede a levantarlo de vuelta.
- healrtbeat_sender: envía heartbearts al healtchecker que es líder. Si la conexión falla tratará de establecer una conexión con otro healthchecker de los posibles.

Algunas aclaraciones de interés:

- Propagación de SIGTERM:
    - Para los nodos, excepto para el cliente, el sigterm se maneja con la librería signal. Cuando se recibe la señal de sigterm, es invocada la función `handle_sigterm` implementada en el middleware, la cual se encarga de cerrar la conexión. Por dentro del mismo, se ejecuta `channel.stop_consuming` en el canal asociado.
    - En el caso de los nodos con más de un proceso, el proceso principal propaga la señal a los subprocesos y luego aguarda su salida y realiza el *join* correspondiente.
    - Para el cliente, en caso de recibir un SIGTERM, el sender_client dará aviso al server mientras que el listener_client será informado por el result handler de que hubo un SIGTERM.

Recuperación de fallas

Es requisito que el tp sea tolerante a fallas, esto es, que ante caídas de los nodos, el funcionamiento sea correcto, más allá de alguna demora acotada. A continuación se evalúan las distintas estrategias que se utilizaron para enfrentar esta dificultad, agrupando los nodos por query. Durante toda esta lectura, recordar las siguientes características del proveedor de MoM utilizado (RabbitMQ):

- Los mensajes que no reciben un acknowledge (de ahora en más ACK) reaparecen en la cola, y si la misma es escuchada por varios nodos, pueden ser consumidos por un nodo distinto del original.

- En caso de darse la situación de arriba pero con un único nodo consumiendo de la cola, el primer mensaje presentado al nodo luego de la caída será el que estaba procesando antes de la misma.

En primer lugar se presentan algunos aspectos en común de todas las queries

1. **Column cleaners**: dentro de esta categoría caen el initial_column_cleaner, el query_2_column_filter y el query_5_column_filter. Como su función es únicamente la de quitar columnas y propagar los mensajes para adelante, no tienen estado alguno, y por lo tanto no requieren recuperación. Se presentan las siguientes alternativas relevantes:

Caída antes del envío	Caída luego del envío

`<filtrado de columnas>`
`<caída>`
`<propagación>`
`<ack>`
	`<filtrado de columnas>`
	`<propagación>`
	`<caída>`
	`<ack>`

En el primer caso el mensaje reaparecerá en la cola y será procesado ya por el mismo nodo al levantarse o por una de sus réplicas. Observemos que en el segundo caso hay una duplicación del mensaje, puesto que habrá sido propagado luego de su procesamiento pero igualmente se lo volverá a procesar. Esto implica por supuesto, la necesidad de una corrección de posibles duplicaciones más tarde.

2. **Logs de estado y vuelos**: la mayor parte de los nodos tienen uno o dos de los siguientes logs:

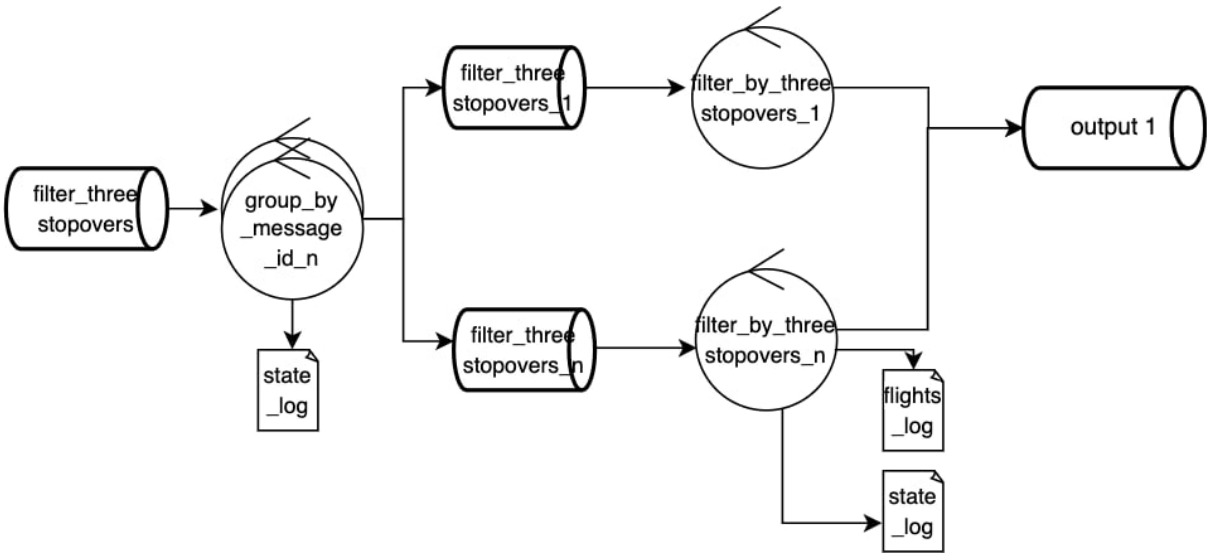
- Un log de estado (*state_log.txt*) que indica el estado de un cliente. Por lo general hay sólo dos:
  `BEGIN_EOF` que indica que se recibió el EOF.
  `EOF_SENT` que indica que se propagó el EOF con la información necesaria para que los siguientes nodos del sistema continúen sus tareas. Esto indica también la posibilidad de borrar los archivos de cliente.

- Un log de registros de vuelos (*flights_log.txt*) que es usado en los nodos en que se desea evitar repetir procesamiento de algún tipo. Su contenido exacto varía de nodo a nodo.

Algunos nodos tienen otros códigos o logs extra, y se especificará el mismo en sus respectivas secciones.

Finalmente se presenta el flow de cada query y las medidas que se tomaron para lograr resultados correctos, incluyendo los protocolos de manejo de mensajes y de recuperación ante una caída.

Query 1



En primera instancia, como los nodos *filter_by_three_stopovers* limitan su función a limpieza de filas, se intentó mantenerlos sin recuperación. Sin embargo, su interacción con la query 3 (ver query 3) generó la necesidad de que estos guarden en su estado la cantidad de registros de distinto *message_id* que pasaron por ellos, y de los mismos, la cantidad que pasó el filtro. Como solución a esto se planteó la existencia de un *group_by_message_id*:
un nodo similar a aquellos presentes en las queries 3, 4 y 5 pero con una diferencia importante: usa una función de hash predecible y conocida por los filtro con información mínima: en este caso el message_id del EOF.

La misma consiste en repartir un mensaje a cada uno, comenzando con el filter_by_three_stopovers_1. De esta forma, al recibir el EOF, los filtros pueden inmediatamente conocer la cantidad de registros distintos que deberían haber recibido antes de considerar su tarea terminada respecto de ese cliente.

El group_by_message_id únicamente guarda un *state_log* básico con dos tipos de líneas distintas:

```
BEGIN_EOF, message_id, client_id
EOF_SENT, client_id
```

De esta forma se asegura de haber enviado el EOF aun en caso de caídas antes de poder propagarlo.

El nodo *filter_by_three_stopovers* requiere una lógica más compleja: se basa en construir un set con los ids de mensajes faltantes cuando se consigue el EOF. Para ello utiliza no solo un *state_log* sino un *flights_log* donde guarda las líneas de la siguiente forma:

```
message_id, client_id, filtering_result
```

donde filtering resulta 1 si el vuelo no pasa, y 0 si pasa. Esta información, sumada al conocimiento que el nodo tiene sobre la distribución de mensajes permite reconstruir fácilmente la lista de mensajes faltantes y número de aceptados.

<u>Manejo de mensajes</u>:

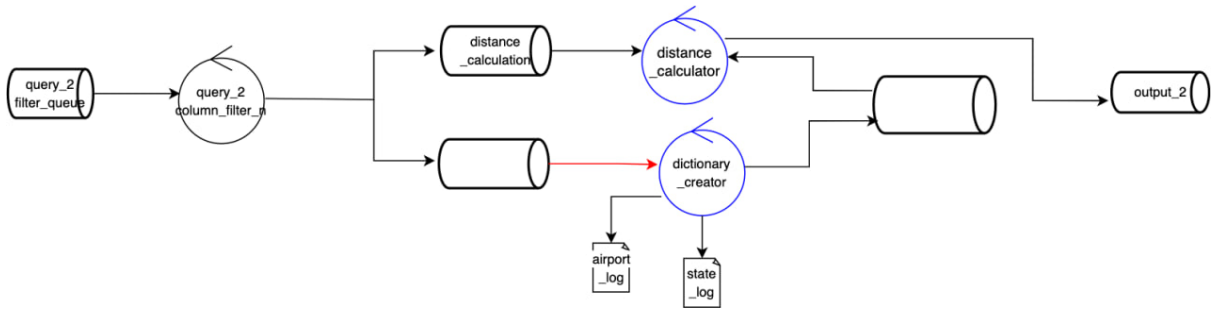
```
para cada registro de vuelos recibido:
            no procesarlo si es de un cliente ya procesado
            si es un EOF:
                  logear BEGIN_EOF,message_id,client_id
                  marcar estado_eof = True
                  obtener mensajes aceptados y faltantes para el cliente
                  enviar eof y logear EOF_SENT,client_id si no faltan mensajes
                  ack y salida
            <filtrado y envío de mensajes>
            logear vuelo recibido con resultado de filtrado
            si estado_eof = True:
                  si aún faltan vuelos:
                        si el vuelo no se filtró aumentar aceptados en 1
                        quitar el id de la lista de faltantes
                  enviar eof y logear EOF_SENT,client_id si no faltan mensajes
            ack
```

Recuperación:

```
para cada línea del state_log:
      si hay un BEGIN_EOF sin su correspondiente EOF_SENT:
            marcar estado_eof = True
            reconstruir mensajes aceptados y faltantes para el cliente
            enviar eof y logear EOF_SENT,client_id si no faltan mensajes
```

Al completar el procesamiento de todos los mensajes, el nodo ya completo su rol en la query 1, y por otro lado, puede construir un mensaje EOF con la información que el *group_by_route* requiere (esto es, la cantidad de mensajes que pasaron el filtro).

Query 2



Esta consta de un *column cleaner* y dos procesos ejecutándose en un mismo nodo, conectados por una *pipe* (el nodo en sí se llama *distance_calculator*). La principal diferencia con el TP1 radica en que ya no es posible abandonar una cola por completo al recibir los registros de vuelos necesarios, pues cada cliente tiene un archivo propio. Esto genera la necesidad de utilizar un proceso secundario para encargarse exclusivamente de la generación del diccionario `{airport_code: coordinates}` utilizado posteriormente para los cálculos en el filtro.

1. El *dictionary creator* escucha en un *exchange* de tipo *fanout* para recibir todos los registros de aeropuertos, almacenando los valores necesarios en un *airports_log*, y realiza un envío por pipe del diccionario de coordenadas de aeropuertos cuando se completa (es posible conocer esto gracias al `EOF_AIRPORTS_FILE` que es propagado a todos los dictionary_creator por igual). Los protocolos son los siguientes:

Manejo de mensajes:

```
para cada registro de aeropuertos recibido:
            si es un EOF:
                  logear BEGIN_EOF, message_id, client_id
                  enviar el diccionario por cola si esta completo
                  ack
            logear airport_code, latitud, longitud a su archivo correspondiente
            ack
```

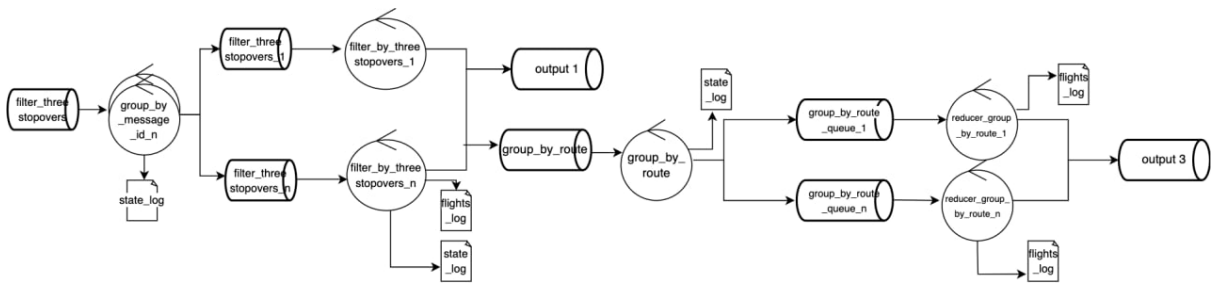
Recuperación:

```
para los clientes activos:
      recrear el diccionario a partir de su log
para cada línea del archivo de estado:
      recuperar el tamaño esperado del diccionario
      enviarlo si esta completo
```

2. El distance calculator simplemente filtra los resultados uno a uno, según la condición pedida. Para evitar demorarse esperando demasiado el diccionario, recién se espera el mismo de la pipe cuando se recibe un mensaje que lo requiere para realizar el filtrado. Esto asume que
    a. Se envían siempre primero los registros de aeropuertos.
    b. La cantidad de registros de vuelo es pequeña respecto de la cantidad de vuelos.

De esta forma se acotan los tiempos de espera de cada cliente. Este proceso no mantiene estado y por lo tanto no precisa recuperación. El uso de pipes permite que al caerse el nodo se vuelva a dar un intercambio de diccionarios (pues el dictionary creator recreará el diccionario necesario y lo enviará). De esta manera se evitan problemas de sincronización a la hora de compartir información por las pipes.

Query 3



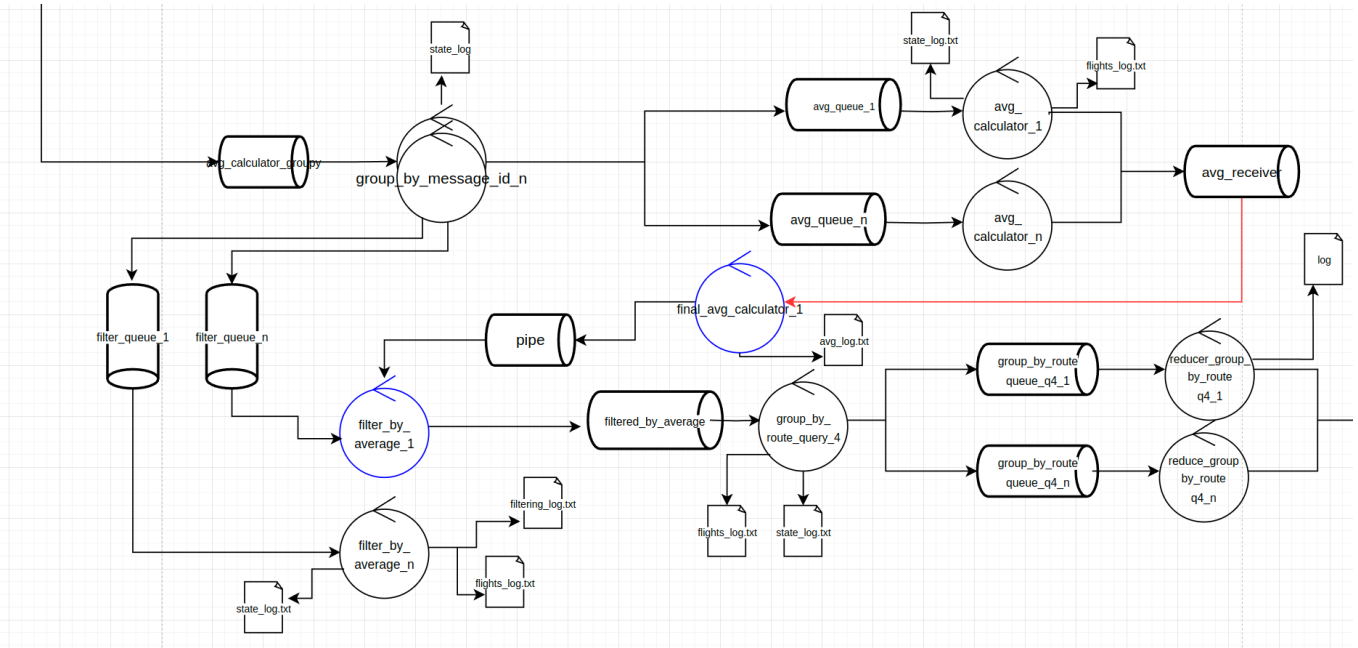
Esta *query* continua a partir del flujo mostrado para la query 1 (justo a continuación de los *filter_by_three_stopovers*. La dificultad inicial radica en que para el correcto funcionamiento de los *reducers*, es **necesario que el EOF llegue ordenado** (esto es, justo atrás de todos los mensajes del cliente, más allá de que después pueda haber duplicados). Esto resultó demasiado difícil de garantizar replicando los group_by_route, por lo tanto se optó por sacrificar disponibilidad y adoptar estos (y los de las queries 4 y 5) como **punto único de falla**.

Aun con esta consideración, hay otro aspecto para tener en cuenta: el momento exacto de envío de propagación del EOF para un cliente cualquiera. Ya no es suficiente con el *message_id*, pues se realiza un filtrado de filas en el paso anterior, y además puede haber duplicados. Para evitar esto se utiliza un *state_log* donde se logean los varios EOF recibidos, uno por cada *filter_by_three_stopover*, que además informa los registros que pasaron el filtro (ver query 1). Finalmente, es cuestión de propagar esa cantidad de registros a los *reducers* (no a cada uno sino la suma en general) y con eso se puede garantizar que el EOF llegará en orden.

Nótese que el group_by_route no tiene state_log, esto es, no filtra vuelos repetidos, y esto es posible debido a que la query pide los 2 vuelos más rápidos, o sea que vuelos duplicados no cambian el resultado. Esta facilidad se limita a esta query en particular.

Query 4

En esta sección se presenta una porción específica de la consulta: hasta el *group_by_route_query_4*, el resto es casi idéntico a las queries 3 y 5 y se presenta allí. La particularidad de esta consulta radica en que todo su comportamiento está fuertemente condicionado por la recepción del EOF. De esta forma, se vuelve especialmente necesaria alguna estrategia para que el mismo llegue a cada nodo en forma ordenada.



El *group_by_message_id* tiene la misma funcionalidad que el presentado en la query 1, pero encolando el mensaje en dos colas distintas. Los nodos *avg_calculator* por su parte, utilizan casi el mismo esquema que el filter_by_three_stopovers, con una diferencia en el *flights_log*, que guarda líneas del estilo `message_id, baseFare.` La recuperación es idéntica, utilizando el log para reconstruir los vuelos faltantes y los valores de *sum* y *count* necesarios.

Con este esquema se vuelve necesario realizar una agregación con todos los resultados parciales de cada *avg_calculator*. Siguiendo con las similitudes entre los nodos, esto lo realiza un proceso lanzado por el *filter_by_average*, que aguarda cada resultado parcial en un intercambio (exchange) de tipo *fanout*, y luego construye el resultado final y lo envía por una pipe. Esta es la misma idea que el dictionary_creator de la *query 2*, pero a diferencia del mismo, como ambos trabajan con el archivo entero: ya no es posible asumir que uno de los procesos va a ser significativamente más breve que el otro. Entonces:

- Se requiere entonces alguna forma de distinguir cuando la probabilidad de que el promedio está lista es máxima.
- Esta idea tiene que permitir recuperación

Para solucionar este problema se utiliza el EOF como indicador aproximado de que el promedio estará en la pipe en breve (se asume claro está, velocidades similares de procesamiento). Por otro lado, la pipe de por sí garantiza la sincronización, pues ante caídas de los nodos, el filter_by_average precisará nuevamente el promedio para filtrar y se repetirá el proceso.

El *filter_by_average* mantiene la particularidad del TP1 de bajar las líneas enteras a disco, a la espera del promedio para poder comenzar su procesamiento. Usa tres logs distintos: el ya conocido *state_log*, *flights_log* que ahora contiene cada registro entero, y finalmente un *filtering_log* con líneas del estilo `lines_processed,lines_accepted.` Los protocolos son:

<u>Manejo de mensajes</u>:

```
si el message_id == EOF:
      obtener promedio general para ese cliente si no se lo tiene
      marcar que el cliente recibió EOF
      logear(BEGIN_EOF, message_id, client_id)
      leer el archivo y procesarlo (*)
      si no hay mensajes restantes:
            propagar EOF con cantidad de mensajes aceptados si corresponde
      ack y salida
logear(message_id, client_id, resultado_de_filtrado)
si se recibió EOF:
      si faltan mensajes:
            <realizar filtrado y envío>
            sumar 1 a los mensajes aceptados si corresponde
            quitar message_id de los mensajes faltantes
      sumar 1 a líneas procesadas
      logear(lineas_procesadas, lineas_aceptadas)
      propagar EOF con cantidad de mensajes aceptados si corresponde
```

```
ack y salida
```

(*) Lectura del archivo

```
construir la lista de ids de mensajes faltantes
para cada línea en el archivo, partiendo de la última línea procesada:
	sumar 1 a las líneas procesadas
	quitar id de mensajes faltantes de la lista
	<realizar filtrado y envío>
	sumar 1 a los mensajes aceptados si corresponde
	cada READ_SIZE lineas logear en filtering log (lineas_procesadas, lineas_aceptadas)
```

Y el protocolo de recuperación:

```
obtener lineas_procesadas, lineas_aceptadas del filtering log
para cada línea válida del state log:
	si existe un BEGIN_EOF sin correspondiente EOF_SENT:
		obtener promedio general para ese cliente
		marcar EOF como recibido
		leer el archivo y procesarlo (*)
		propagar EOF con cantidad de mensajes aceptados si corresponde
```

Existe una **limitación importante** del sistema relacionada con la implementación de la lectura del archivo una vez recibido el promedio: el mismo se despacha sin pausas hasta llegar al final (no necesariamente terminó el procesamiento de cliente en ese punto), y luego se envían los mensajes a medida que se reciben. Esto claramente tiene el potencial de retrasar considerablemente a los otros clientes. El mejor rendimiento se alcanza entonces cuanto más antes llegue el EOF (y por lo tanto el promedio).

El resto del flujo de la query 4 es muy parecido al de la 3 ya presentada, con la diferencia de que se requiere un log de vuelos (no aplica idempotencia respecto de duplicados). Se presenta a continuación

Tipos de archivos logs usados:

- **log de estado:** guarda información del estado del reducer (cada reducer tiene uno)
- **log de ruta:** cada línea guarda información sobre los mensajes de vuelos recibido correspondientes a una ruta por cliente (cada cliente va a tener un archivo de estos por ruta por el cual agrupa el reducer)

Protocolo:

1. El reducer recibe un mensaje con un vuelo y se identifica el número de cliente y la ruta perteneciente al mismo
2. Se graba en el **log de ruta** (si no existe se lo crea) para la ruta del vuelo recibido del cliente correspondiente la siguiente línea al final del archivo:
   <id_mensaje_vuelo>,**<sumatoria>,<máximo>, <cantidad>\n**
   donde:
   - **sumatoria**: sumatoria del **totalFare** de los vuelos recibidos para esa ruta
   - **máximo:** máximo **totalFare** recibido para esa ruta

- **cantidad:** número de vuelos recibidos para esa ruta

3. En algún momento se va a recibir el EOF correspondiente al cliente.
4. Se construye el mensaje del resultado para la ruta y se envía
5. Por cada ruta procesada se escribe en el **log de estado** del reducer la siguiente línea antes de mandar el resultado del aeropuerto:

   <id_cliente>,**<codigo_ruta>\n**
6. Cuando procesaron todos los **log de ruta** del cliente se escribe la siguiente línea en el **log de estado:**

   <id_cliente>\n

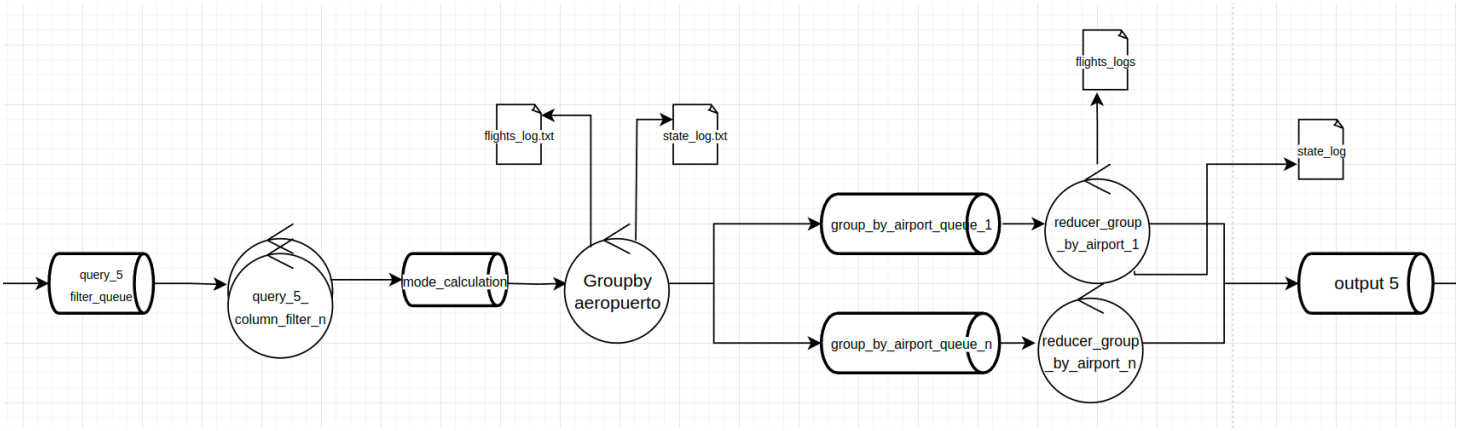
Recuperación:

Al momento de la recuperación pueden ocurrir los siguientes escenarios:

1. **log de estado** vacío => no se recibió ningún EOF
   a. Se corrigen todos los **log de ruta** de cada cliente y se recorre cada uno de dichos logs.
   b. Se toma el **id_mensaje_vuelo** procesado por cada cliente. La última línea del archivo contiene la sumatoria, máximo y cantidad de vuelos procesados hasta el momento.
   c. Ahora se conocen todos los vuelos procesados para cada cliente entonces si llega un vuelo con un message_id que está repetido para un cliente dicho vuelo será descartado.

2. **log de estado no** vacío => se recibió al menos un EOF:
   a. Se lee el **log de estado** de arriba hacia abajo
   b. Si para un cliente la primera línea que se encuentra es **<id_cliente>\n** entonces el cliente se considera procesado y se revisa si se quedan archivos del cliente por borrar.
   c. Si para un cliente la primera línea que se encuentra es **<id_cliente>**,**<codigo_ruta>\n** entonces el cliente recibió el eof pero tiene **log de ruta** por cerrar, se identifican los archivos faltantes. Se va leyendo la última línea de cada archivo la cual contiene el último valor registrado de la sumatoria. máximo y cantidad y se construye el mensaje de la ruta con esa información.
   d. Para aquellos clientes que no están en **log de estado** se recupera igual a como si no hubiesen recibido EOF (paso 1 de Recuperación)

Query 5



La estructura de la query 5 se asemeja fuertemente a la de la query 1, con la excepción marcada de que el *query_5_column_filter* es un nodo de limpieza de columnas, y como tal, el *message_id* del EOF sigue valiendo como indicador de la cantidad de líneas diferentes que es necesario procesar (o sea no es necesario recalcularlo como en el caso de *group_by_route*). Por otro lado, siendo el mismo un filtro de columnas no es necesario implementar recuperación de estado de ningún tipo.

Aprovechando el comportamiento de RabbitMQ establecido al principio de esta sección y que solo se utiliza un único *group by* se puede afirmar que bastará con enviar un EOF luego de procesados todos los mensajes para que el reducer realice correctamente el procesamiento.

Tanto el *group_by_airport* como cada *reducer_group_by_airport* posee un *state_log* que guarda las líneas ya conocidas (en el caso del reducer tan solo se anota el número del cliente al finalizar su procesamiento) y varios *flights_log* (por aeropuerto por cliente).

A continuación se presentan los protocolos relevantes:

<u>Manejo de mensajes</u>:

1. El reducer recibe un mensaje con un vuelo y se identifica el número de cliente y el aeropuerto perteneciente al mismo
2. Se graba en el **log de aeropuerto** (si no existe se lo crea) para el aeropuerto del vuelo recibido del cliente correspondiente la siguiente línea al final del archivo:
   <id_mensaje_vuelo>,<base_fare_vuelo>\n
3. En algún momento se va a recibir el EOF correspondiente al cliente.
4. Se recorren todos los **log de aeropuerto** del cliente correspondiente. Para cada archivo se va a leer cada línea y se va a obtener todos los **baseFare** correspondientes al aeropuerto. Para cada aeropuerto se calcula la moda y la cantidad de repeticiones, si hay dos o más **baseFare** con misma moda se los incluye en el resultado final del aeropuerto.
5. Por cada aeropuerto procesado se escribe en el **log de estado** del reducer la siguiente línea antes de mandar el resultado del aeropuerto:
   <id_cliente>,<codigo_aeropuerto>\n

6. Cuando procesaron todos los **log de aeropuerto** del cliente se escribe la siguiente línea en el **log de estado:**
   <id_cliente>\n

Recuperación:

Al momento de la recuperación pueden ocurrir los siguientes escenarios:

1. **log de estado** vacío => no se recibió ningún EOF
   a. Se corrigen todos los **log de aeropuerto** de cada cliente y se recorre cada uno de dichos logs.
   b. Se toma el **id_mensaje_vuelo** procesado por cada cliente.
   c. Ahora se conocen todos los vuelos procesados para cada cliente entonces si llega un vuelo con un message_id que está repetido para un cliente dicho vuelo será descartado.
2. **log de estado no** vacío => se recibió al menos un EOF:
   a. Se lee el **log de estado** de arriba hacia abajo
   b. Si para un cliente la primera línea que se encuentra es **<id_cliente>\n** entonces el cliente se considera procesado y se revisa si se quedan archivos del cliente por borrar.
   c. Si para un cliente la primera línea que se encuentra es **<id_cliente>**, **<codigo_aeropuerto>\n** entonces el cliente recibió el eof pero tiene **log de aeropuerto** por cerrar, se identifican los archivos faltantes y se procede a procesarlos como se describió en el protocolo (paso 4 en adelante)
   d. Para aquellos clientes que no están en **log de estado** se recupera igual a como si no hubiesen recibido EOF (paso 1 de Recuperación)

Healthchecker

Este módulo se encarga de la recuperación de los nodos caídos. Hay tres réplicas de *healthcheckers* donde una se comporta como líder y cumple la función de levantar nodos caídos. A partir del siguiente esquema:

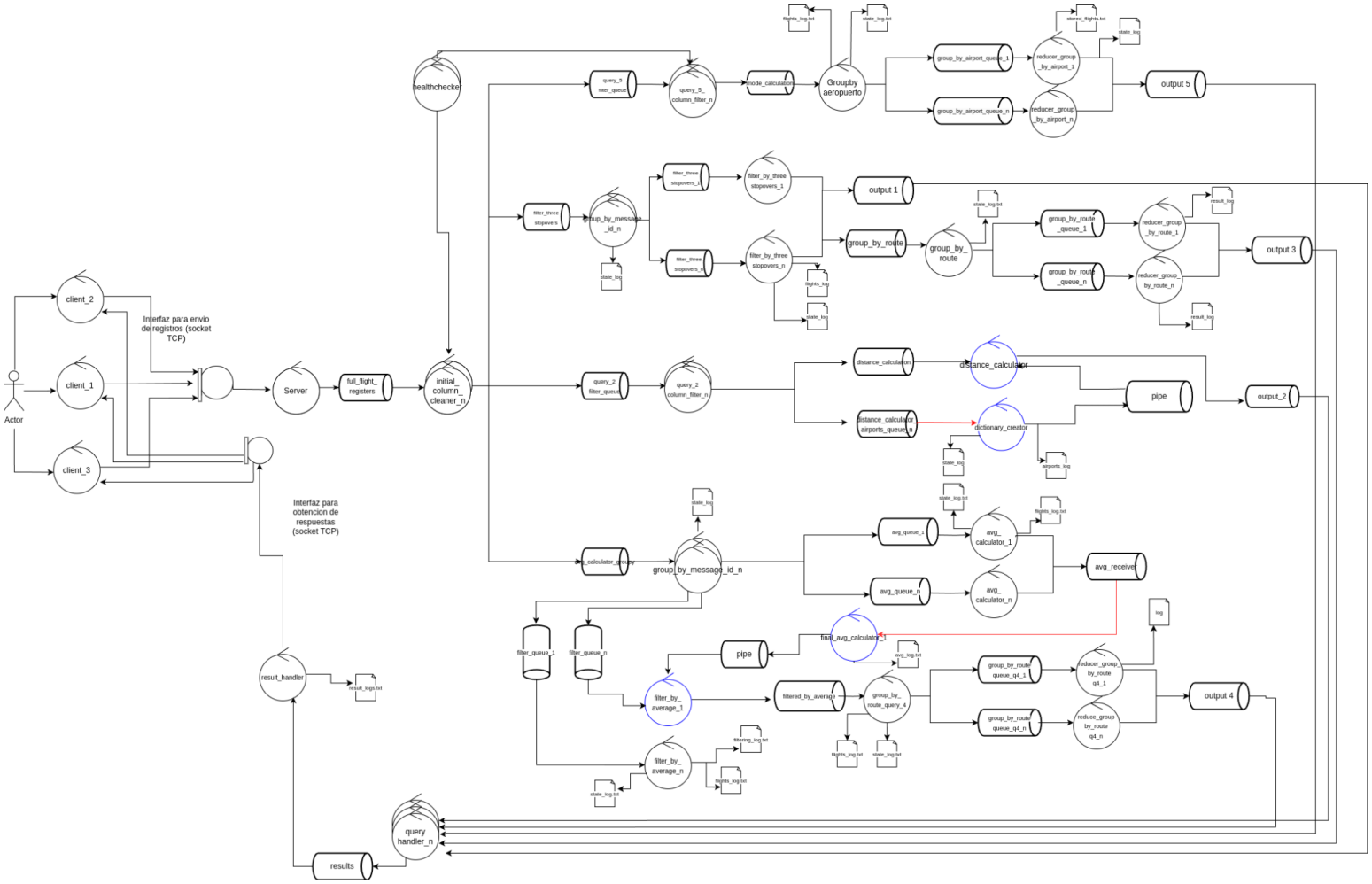
1. Nodos/*healthchecker* no líder: Estos nodos tienen como atributo un objeto *heartbeat_sender* que se encarga de con una frecuencia dada enviar mensajes mostrando que vive. En caso que un *healthchecker* no conteste, intenta enviarle al resto y queda en comunicación con el que haya contestado. Por otra parte, los healthcheckers sí saben quién es el líder y se comunican, también mediante un *heartbeat_sender*, con el líder.
2. El healtchchecker, por otra parte, escucha *heartbeats* de los otros nodos del sistema, exceptuando aquellos con conexiones TCP (ver diagrama de paquetes). Ante un timeout o caída del socket, se utiliza *docker-in-docker* para reiniciar el nodo caído (mediante `docker start`)
3. Teniendo varios posibles nodos que quieren ejecutar comandos, se requiere utilizar un algoritmo de elección de líder para determinar quién es el líder actual (o sea el único con acceso a la funcionalidad principal en ese momento). Se decidió utilizar el algoritmo Bully.

Fallas de lectura

Para posibles fallas de lectura se decidió implementar un algoritmo sencillo que no requiriese borrado de líneas constante. Se realiza en dos partes:

1. Antes de abrir cualquier archivo al levantarse, se invoca una función *correct_last_line(file)* que analiza la última línea del archivo (la única que puede estar rota) y en caso de estarlo, o sea de no cumplir con que el carácter newline sea el último, se la señaliza mediante un caracter numeral (#) al final y se inserta luego un newline para dejar el archivo “corregido” y listo para escritura.
2. Para cada lectura posterior del archivo se ignoran las líneas que terminan con con ‘#/n’ pues no contienen información válida.

Diagrama de robustez



Algunas consideraciones importantes sobre el diagrama:

- Las líneas rojas representan intercambios (exchanges) de tipo fanout
- Los nodos azules representan dos procesos de un mismo sistema individual
- Los healthcheckers tiene una conexión a cada nodo exceptuando aquellos con conexiones TCP (clientes, *server* y *result handler*). Las conexiones ilustradas son a mero efecto representativo.

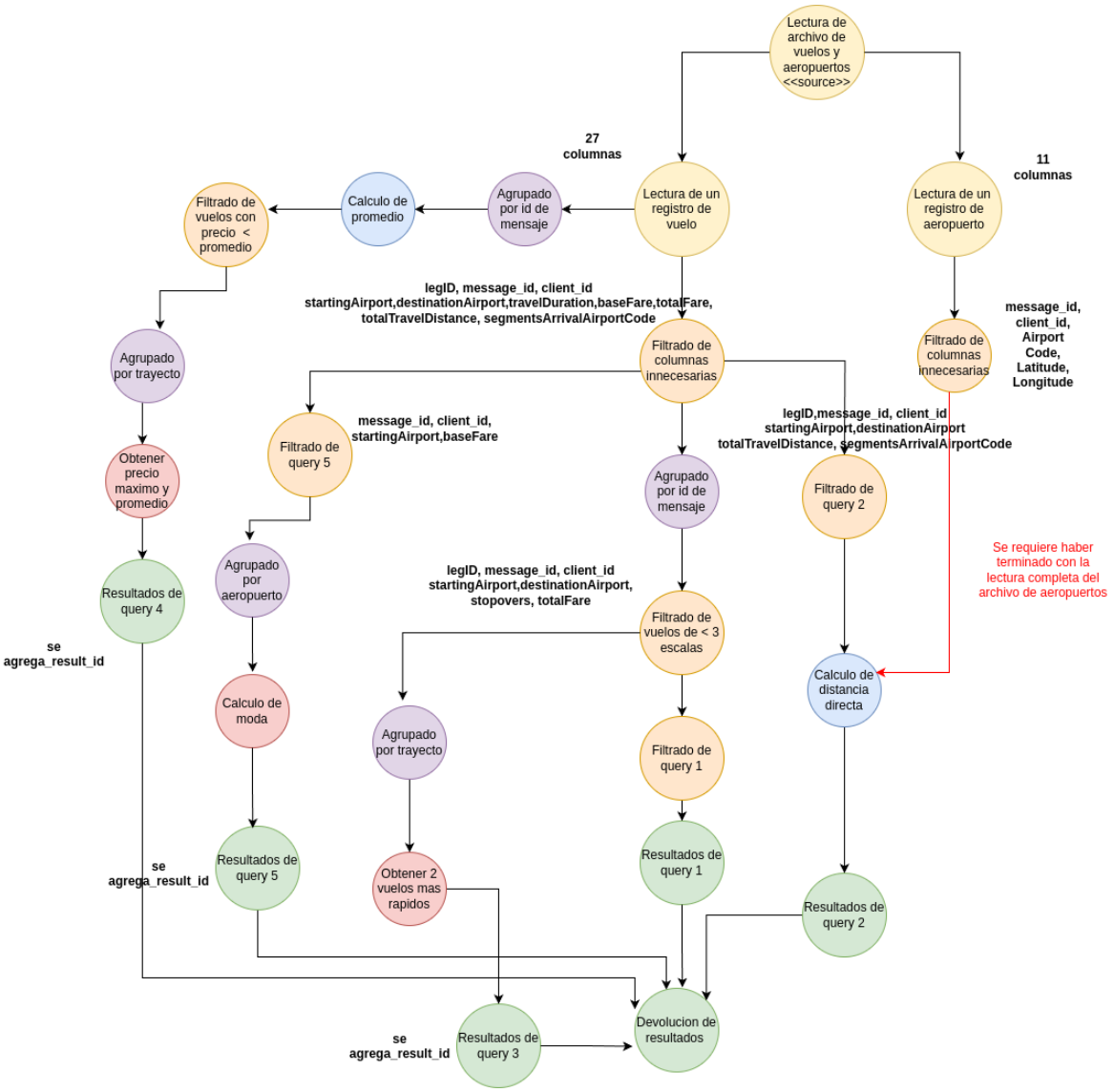
Vista de procesos

Se muestra el grafo acíclico dirigido que se utilizó para modelar las consultas del sistema. Se indican las tareas aplicadas a los datos de entrada (registros de vuelos). Las tareas están coloreadas según su rol en forma general:

- Amarillo para la fuente de los datos.
- Naranja para los filtros.
- Azul para cálculos.
- Violeta para agrupamiento de datos.
- Rojo para procesamiento de datos agrupados.
- Verde para la salida de los datos.

Como se puede observar el DAG es casi idéntico al TP1, con las salvedades de que:

- Se agregan los campos message_id y client_id como restantes luego de cada filtrado
- Se consideró el agrupado por message_id (realizado en los *group by* de las queries 1 y 4), aunque no sea parte del procesamiento en sí de los datos.
- Se agrega un result_id para etiquetar los resultados y permitir así que sean filtrados en el último nodo (result handler) antes de enviarlos al cliente.



Vista de desarrollo

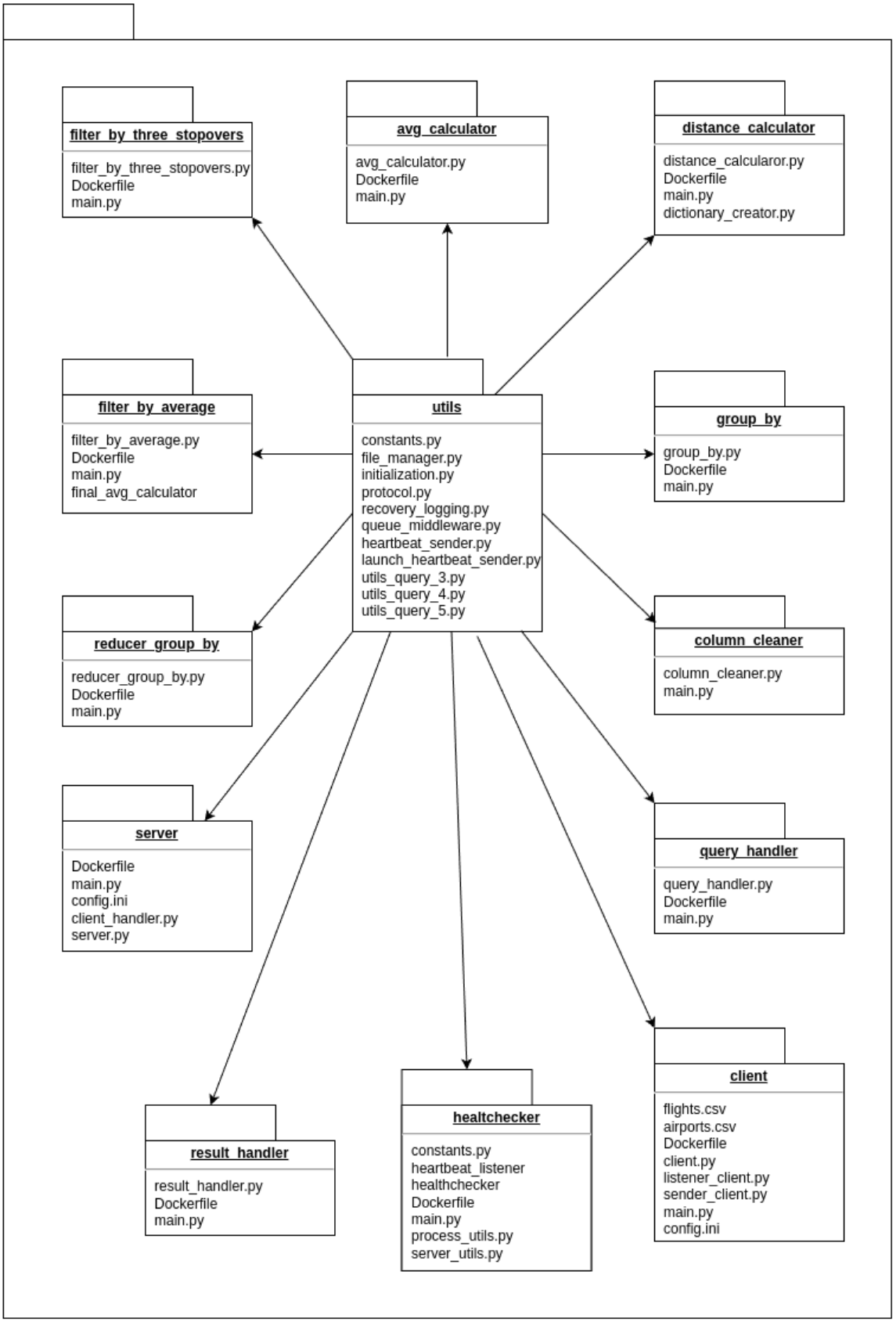
Diagrama de paquetes

En el diagrama de paquetes se pueden ver los archivos que componen el sistema así como los paquetes donde están distribuidos cada uno. Se observa que la gran mayoría usan las funciones definidas en los archivos de la carpeta utils. Esto se debe al uso de middleware propio para encapsular el uso de *pika*, así como funciones con funcionamiento general que puede ser usadas por distintos nodos y un archivo para constantes. Respecto del TP1 se notan los siguientes cambios:

- Un *distance_calculator.py* que encapsula el comportamiento de recepción de registros de aeropuertos y la creación del diccionario de coordenadas indexadas por código de aeropuerto.
- Un *client_handler.py* que cumple la función vieja del servidor, esto es, el manejo de un cliente, exceptuando la recepción de la conexión por el welcoming socket (que pasa a manejarla el server antes de delegar el resto de la interacción con el cliente).
- Un *final_avg_calculator.py* que maneja la recepción de los resultados de promedio parcial de cada *avg_calculator* y la construcción del promedio final para filtrado con los mismos.
- En *utils* se observan *launch_heartbeat_sender.py* y *heartbeat_sender.py* para encapsular el comportamiento de pingeo al healthchecker y facilitar su importación por cada clase de nodo del sistema.
- Finalmente, un nuevo paquete *healthchecker* que permite recibir *heartbeats* constantes de parte de los nodos y revivirlos en casos de que estos se detengan mediante *docker-in-docker*. También admite replicación y utiliza el algoritmo de elección Bully para decidir un líder y evitar superposición a la hora de lanzar los comandos para revivir nodos caídos.

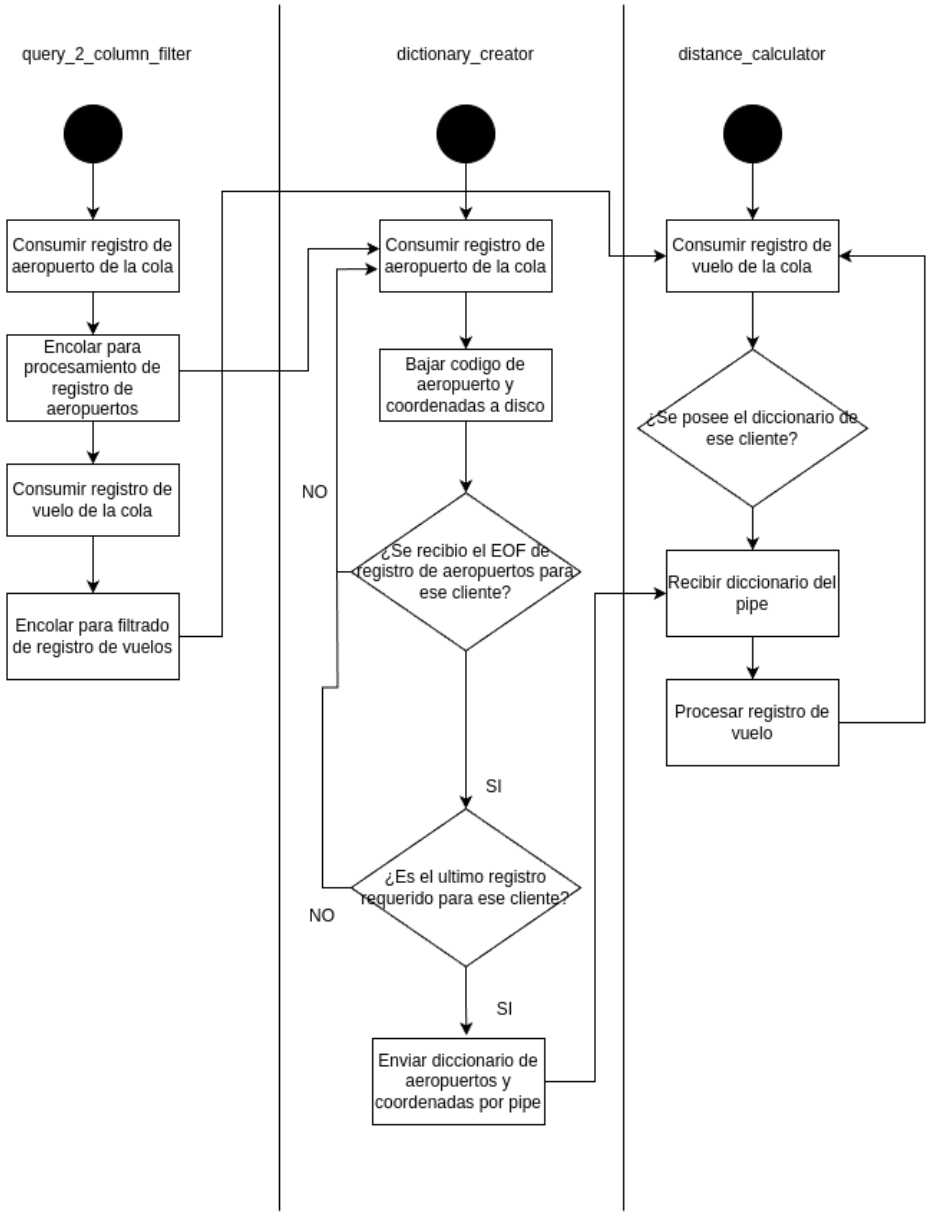
Por otro lado (no presente en el diagrama por simplicidad) se crean archivos para cada cada nodo (llamados *logs* durante todo el informe) para guardar información necesaria para efectuar correctamente la recuperación luego de fallos. Los mismos siguen una estructura común para facilitar el posterior borrado al finalizar con el cliente:

- **{paquete}/{nombre de contenedor}/{cliente}/<archivos de cada cliente>**
- **{paquete}/{nombre de contenedor}/<state_log.txt>**


filter_by_three_stopovers
filter_by_three_stopovers.py
Dockerfile
main.py
avg_calculator
avg_calculator.py
Dockerfile
main.py
distance_calculator
distance_calcularor.py
Dockerfile
main.py
dictionary_creator.py
filter_by_average
filter_by_average.py
Dockerfile
main.py
final_avg_calculator
utils
constants.py
file_manager.py
initialization.py
protocol.py
recovery_logging.py
queue_middleware.py
heartbeat_sender.py
launch_heartbeat_sender.py
utils_query_3.py
utils_query_4.py
utils_query_5.py
group_by
group_by.py
Dockerfile
main.py
reducer_group_by
reducer_group_by.py
Dockerfile
main.py
column_cleaner
column_cleaner.py
main.py
server
Dockerfile
main.py
config.ini
client_handler.py
server.py
query_handler
query_handler.py
Dockerfile
main.py
result_handler
result_handler.py
Dockerfile
main.py
healtchecker
constants.py
heartbeat_listener
healthchecker
Dockerfile
main.py
process_utils.py
server_utils.py
client
flights.csv
airports.csv
Dockerfile
client.py
listener_client.py
sender_client.py
main.py
config.ini

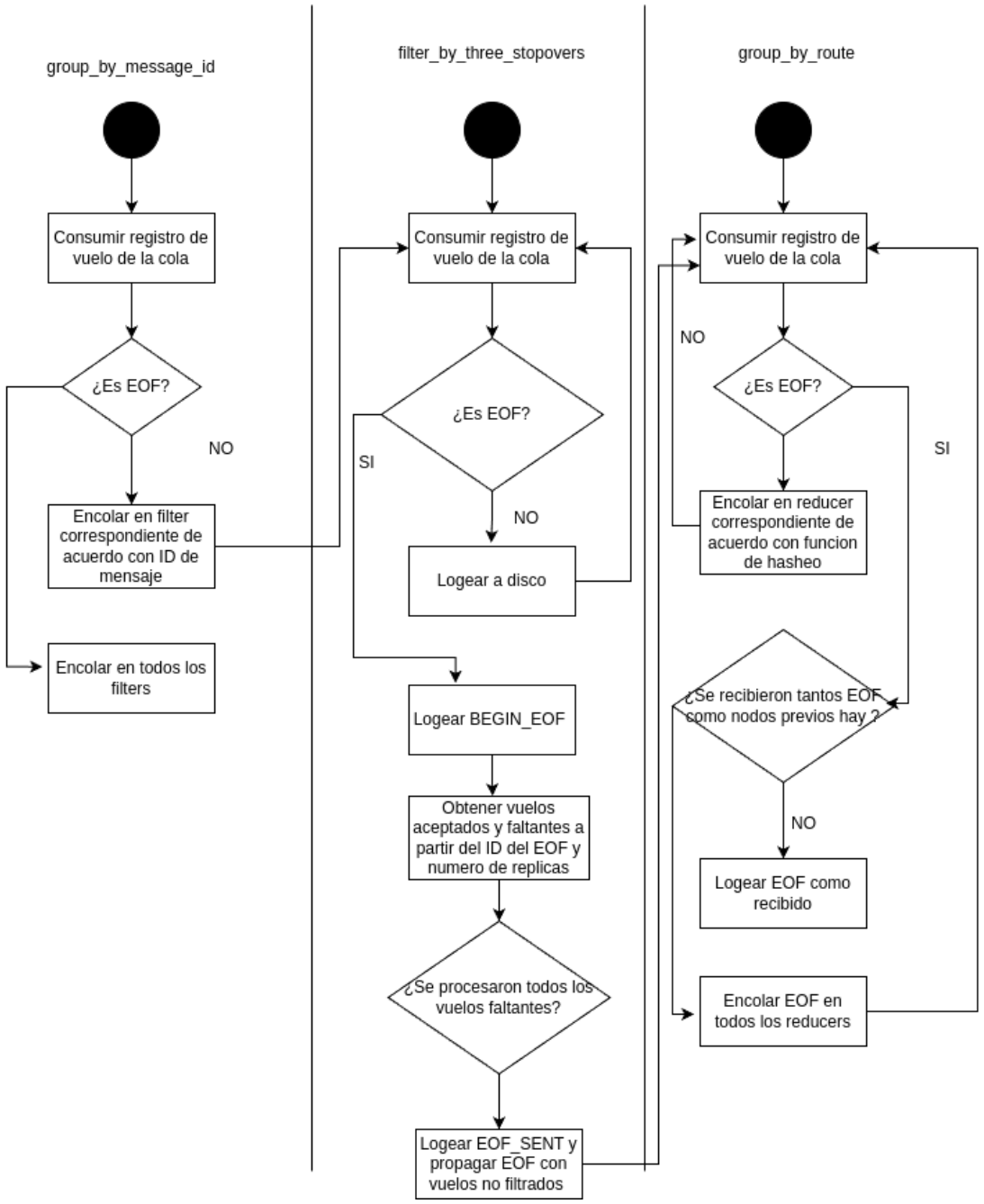
Vista lógica

Diagrama de actividad de la consulta 2



En el diagrama de arriba se muestra la interacción entre los dos procesos que componen el nodo *distance calculator*, y cómo se sincronizan en la comunicación mediante una pipe para que siempre se encuentre disponible el dato requerido.

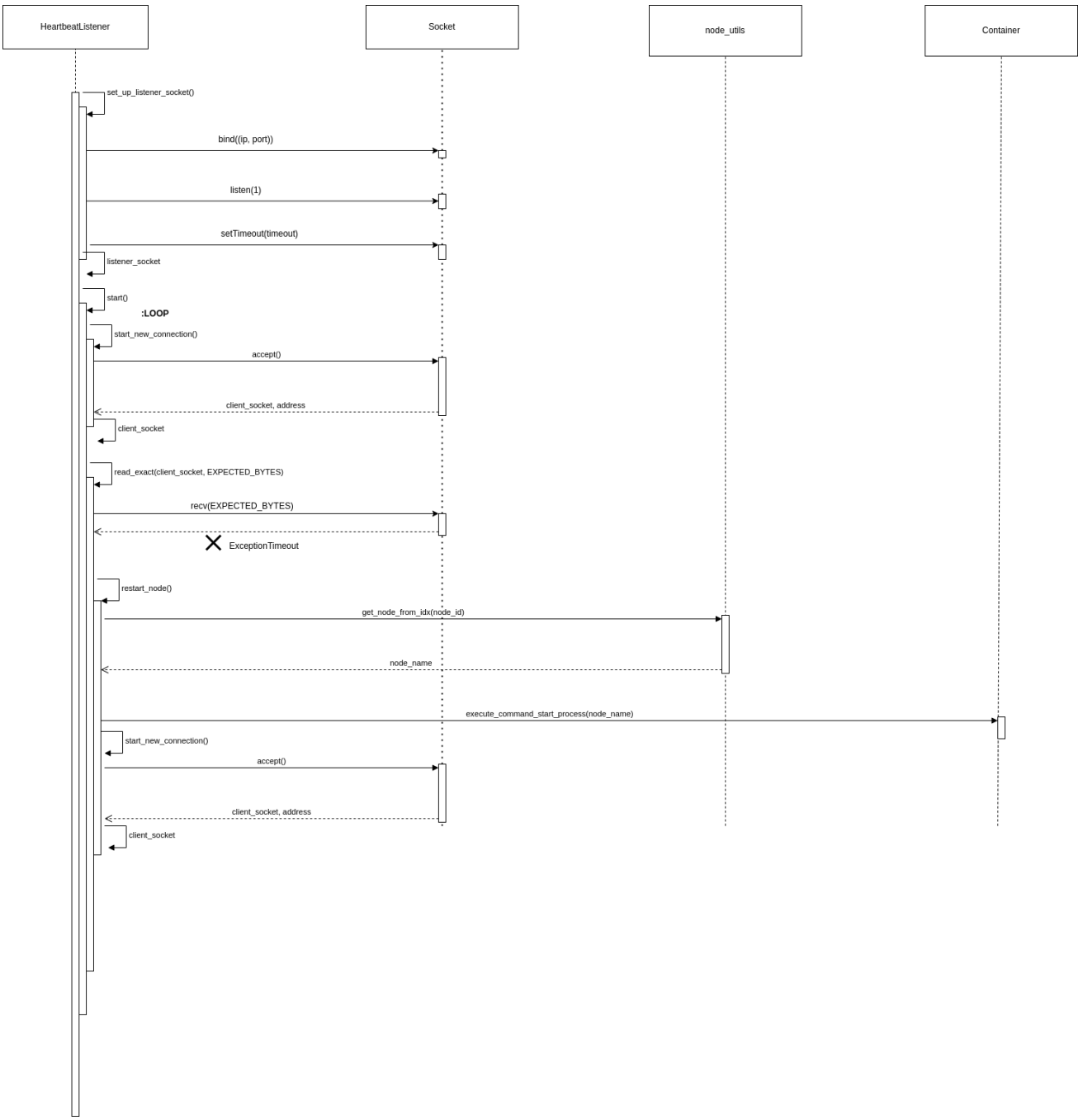
Diagrama de actividad de la interacción de la query 3



En el diagrama de arriba se muestra el protocolo de manejo de mensajes y el flujo de interacción básico entre el filter_by_three_stopovers y el group_by_route. Obsérvese que en ningún momento el group_by_route consulta un log para filtrar mensajes repetidos (a diferencia de los group by de las queries 4 y 5). Esto se debe a que la naturaleza de la query misma (los 2 vuelos más rápidos) permite que el procesamiento de mensajes repetidos resulte idempotente.

Diagrama de secuencia

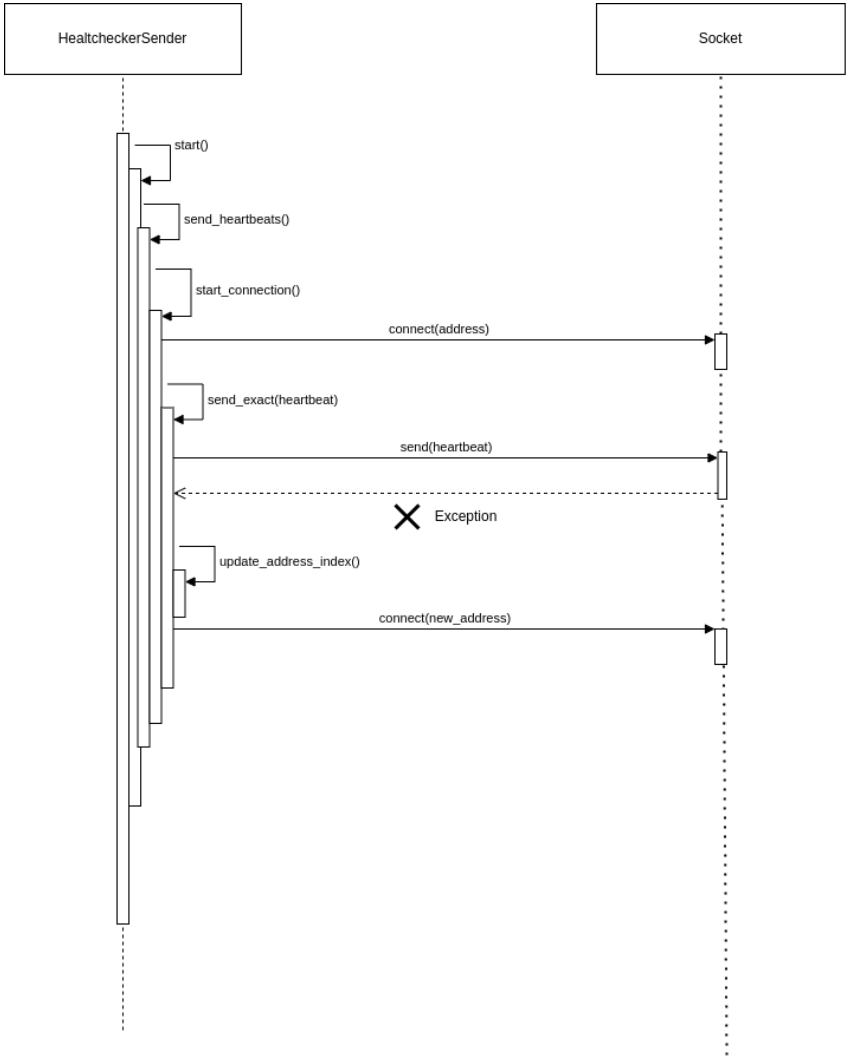
HeartbeatListener



Este gráfico describe el funcionamiento del Healtchecker para detectar que un nodo está caído y proceder a levantarlo devuelta:

1. En primer lugar el *healthcheker* hace un bind y un listen en los puertos indicados y luego aceptará una conexión TCP con el *HeartbeatSender*.
2. El *HeartbeatListener* setea un timeout y luego espera por un heartbeat. Si el *heartbeat* llega entonces el Heartbeat Sensor está activo y no necesita ser levantado.
3. Si aparece una excepción entonces la conexión con el *Sender* se cayó y él mismo necesita ser levantado.
4. El Listener identifica el nombre del nodo que está caído y procede a levantarlo.
5. El listener procede a esperar por una nueva conexión del *Sender* para continuar con su comunicación.

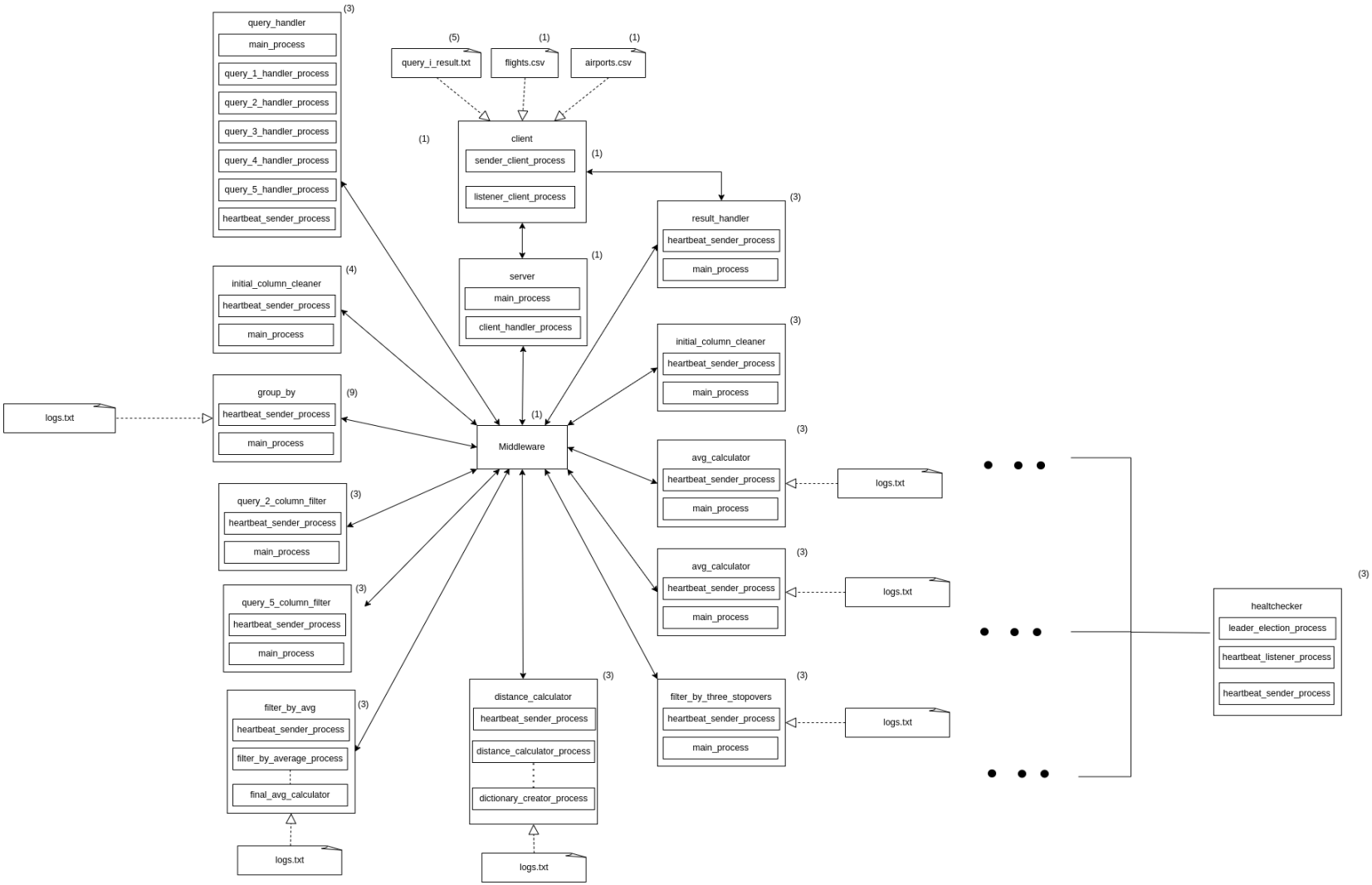
HeartbeatSender



En el caso que el Sender sea el que identifica que la conexión con el proceso Listener es el que está descrito a continuación:

1. El sender hace de cliente para el Heartbeat Listener y establece una conexión TCP con el mismo.
2. El Sender envía un Heartbeat al Listener.
3. Si la conexión con el listener falla entonces el Sender intentará establecer una conexión con otro Healtchecker para poder enviar el Heartbeat y que pueda avisar que se encuentra activo.

Diagrama de despliegue



Los cambios introducidos en el diagrama con respecto al TP1 son los siguientes:

- En primer lugar se suma un nuevo tipo de nodo: el **healthchecker**, cuyo rol consiste en volver a lanzar nodos caídos por cualquier motivo. El mismo está replicado tres veces y lanza tres tipos de procesos durante su ejecución:
    - **leader_election_process**: para poder realizar la elección de líder entre las réplicas.
    - **heartbeat_listener_process**: este proceso lo lanza únicamente el líder y su función es aguardar heartbeats de los nodos para identificar caídas.
    - **heartbeat_sender_process**: proceso ejecutado por las replicas que no son el líder y cuya función es enviarle al líder heartbeats para que el sepa que siguen levantadas.
- Los nodos **avg_calculator**, **filter_by_average**, **filter_by_three_stopovers, group_by** (de todo tipo) **y distance_calculator** contienen ahora archivos de logs (los tipos de logs utilizados varían según el tipo de nodo) para garantizar la tolerancia a fallas.
- Todos los nodos utilizados en la entrega anterior (excepto el cliente) ahora lanzan un proceso **heartbeat_sender_process** para poder comunicar al líder que permanecen activos.
- Los nodos de **filter_by_average** lanzan dos nuevos procesos:
    - **final_avg_calculator_process**: proceso que espera resultados parciales de cada **avg_calculator** y construye un resultado final para enviar por pipe al proceso encargado de filtrado.
    - **filter_by_average_process**: proceso que registra los vuelos que llegan y una vez llegan los promedios esperados empieza a despachar aquellos con total Fare mayor al promedio recibido.
- Los nodos de **distance_calculator** lanzan dos nuevos procesos:
    - **dictionary_creator_process**: recibe los aeropuertos por un intercambio de tipo fanout, crea un diccionario con las coordenadas de cada aeropuerto como valor y el código del aeropuerto como llave, y una vez completo el mismo, lo envía por pipe al proceso encargado del filtrado.
    - **distance_calculator_process**: usa el diccionario recibido para procesar los vuelos y obtener el resultado de su respectiva consulta.
- El nodo server ya no maneja clientes directamente sino que invoca un **client_handler_process** para manejar cada uno.