.UBAfiuba
FACULTAD DE INGENIERÍA

Flights Optimizer

Escalabilidad, Middleware y Coordinación de Procesos

Segundo cuatrimestre 2023

Apellido y Nombre	Padrón	Email
Buzzone, Mauricio	103783	mbuzzone@fi.uba.ar
De Angelis Riva, Lukas Nahuel	103784	ldeangelis@fi.uba.ar

Índice

1. Alcance

Se pide crear un sistema distribuido que analice 6 meses de registros de precios de vuelos de avión que permita obtener el resultado de las siguientes cuatro 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.

2. Vista de Escenarios

2.1. Diagrama de casos de uso

El sistema contempla tres casos de uso principales, marcados en el siguiente diagrama.

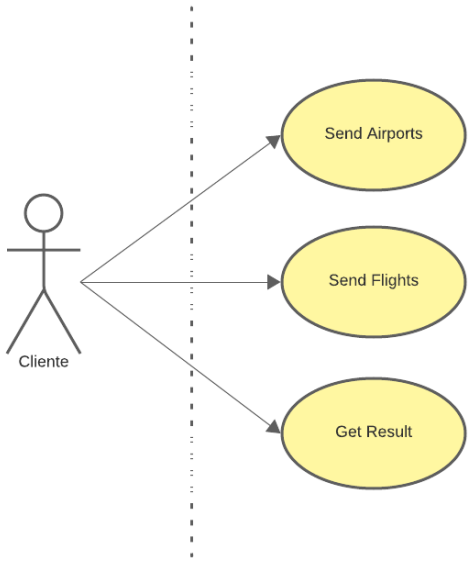

Figura 1: Diagrama de casos de uso

- **Send Airports**: Determina como el **Cliente**, envía los registros de aeropuertos al sistema.
- **Send Flights**: Determina como el **Cliente**, envía los registros de vuelos al sistema. Este caso de uso debe ser posterior a **Send Airports**.
- **Get Result**: Determina como el **Cliente**, solicita los resultados de las cuatro consultas.

3. Vista Lógica

3.1. DAG

Se presenta el siguiente diagrama DAG mostrando las relaciones entre las distintas actividades. Se puede observar como el procesamiento de las consultas están divididas en etapas con tareas simples e individualmente escalables.

Figura 2: DAG- Flights Optimizer

3.2. Diagramas de Clases

Mas cercano a la implementación se muestran los diagramas de clases de las partes mas importantes del sistema.

3.2.1. Middleware

La clase **Middleware** presenta una interfaz mínima para separar Pika de nuestro sistema. Luego las distintas clases hijas se encargan de declarar de donde y hacia donde se enviarán los mensajes.

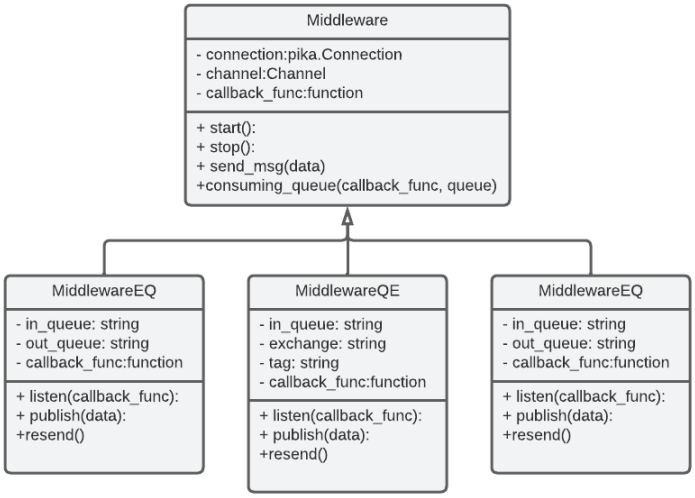

Figura 3: Diagrama de clases Middleware

Donde:

- **MiddlewareQQ**: se trata de un middleware que comunica, tanto por input como por output, con el patrón producer/consumer. **QQ** hace referencia a Queue-to-Queue.
- **MiddlewareEQ**: se trata de un middleware que se comunica, por input utilizando el patrón publisher/subscriber y por output utilizando producer/consumer. En este sentido **EQ** hace referencia a Exchange-to-Queue.
- **MiddlewareQE**: se trata de un middleware que se comunica, por input utilizando el patrón producer/consumer y por output utilizando el patrón publisher/subscriber. En este sentido **QE** hace referencia a Queue-to-Exchange.

Unificando la interfaz de comunicación de los 3 posibles Middlewares en las siguientes sencillas funciones:

- `listen`: escuchar mensajes.
- `publish`: publicar mensajes.
- `resend`: enviar mensajes al input.

3.2.2. Workers y Handlers

Todos los handlers tiene una misma lógica principal donde lo que realmente cambia es lo que hacen con los mensajes que reciben a través del middleware y como se envían hacia el siguiente paso del pipeline, de modo que existe una clase **Worker** que se encarga de encapsular toda la lógica. Luego, los distintos handlers sólo deben implementar una función para trabajar los datos para cada caso particular.

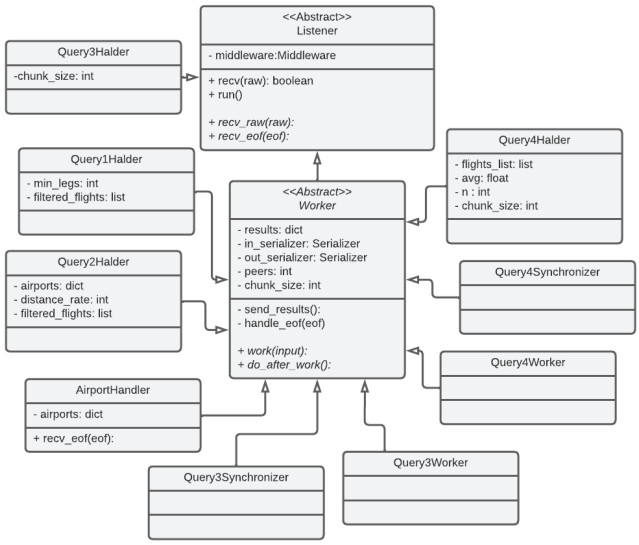

Figura 4: Diagrama de clases Worker

En el diagrama entonces podemos observar dos clases abstractas:

- **Listener**: abstrae la interfaz de comunicación contar el middleware que recibe al instanciarse. Es necesario definir las siguientes funciones:
  - `recv_raw`: recepción de un chunk de datos crudos.
  - `recv_eof`: recepción de un mensaje de finalización
- **Worker**: abstrae la lógica de recepción de un chunk de datos y envío de resultados. Los datos recibidos son deserializados utilizando `in_serializer` y los resultados a enviar son serializados utilizando `out_serializer`. Es necesario definir las siguientes funciones:
  - `work`: el trabajo a realizar sobre un único dato del chunk
  - `do_after_work`: define qué se hace luego de haber realizado `work` sobre todo un chunk

4. Vista de Procesos

A continuación se presentan distintos escenarios y como se maneja el flujo de mensajes a través del sistema.

4.1. Flujo de los vuelos para la Consulta 1 y 3

En el siguiente diagrama se presenta como un vuelo viaja a través de los filtros de la consulta 1 y posteriormente de la consulta 3. Se puede observar en el diagrama principalmente el cómo todos los vuelos seleccionados por tener tres escalas o más son enviados tanto al **ResultHandler** como a el pipeline de la consulta 3.

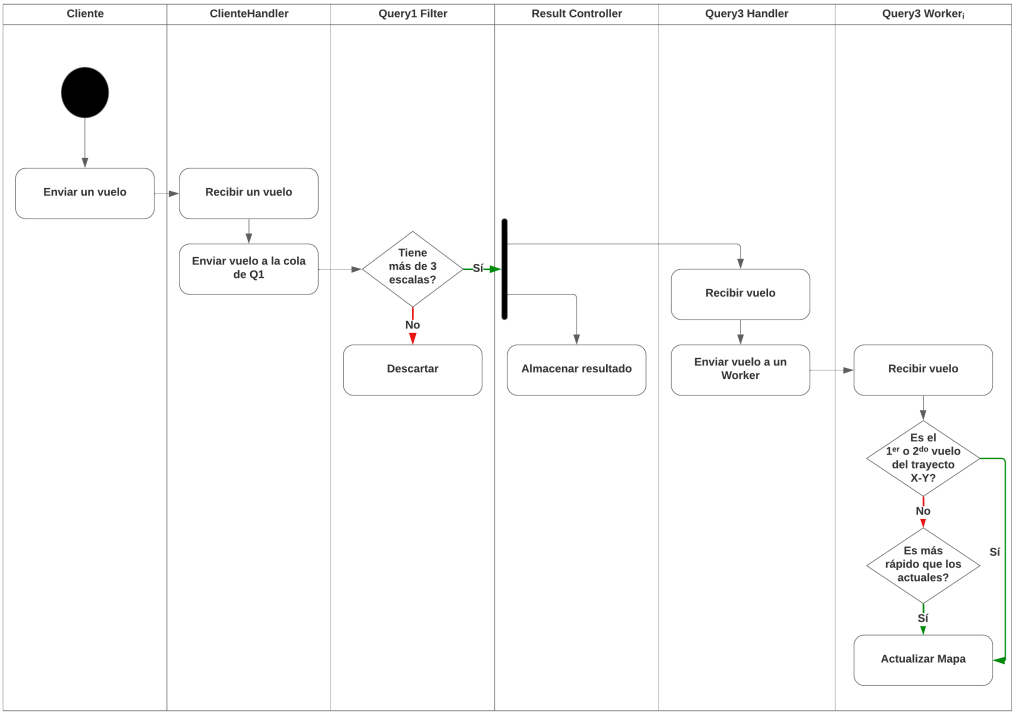

Figura 5: Flujo de mensajes con vuelos para la Consulta 1 y 3

4.2. Flujo de los vuelos para la Consulta 2

En el siguiente diagrama se ve como como funciona el flujo de mensajes para la consulta 2. Donde primero se envían todos los aeropuertos con sus coordenadas y luego se envían todos los vuelos.

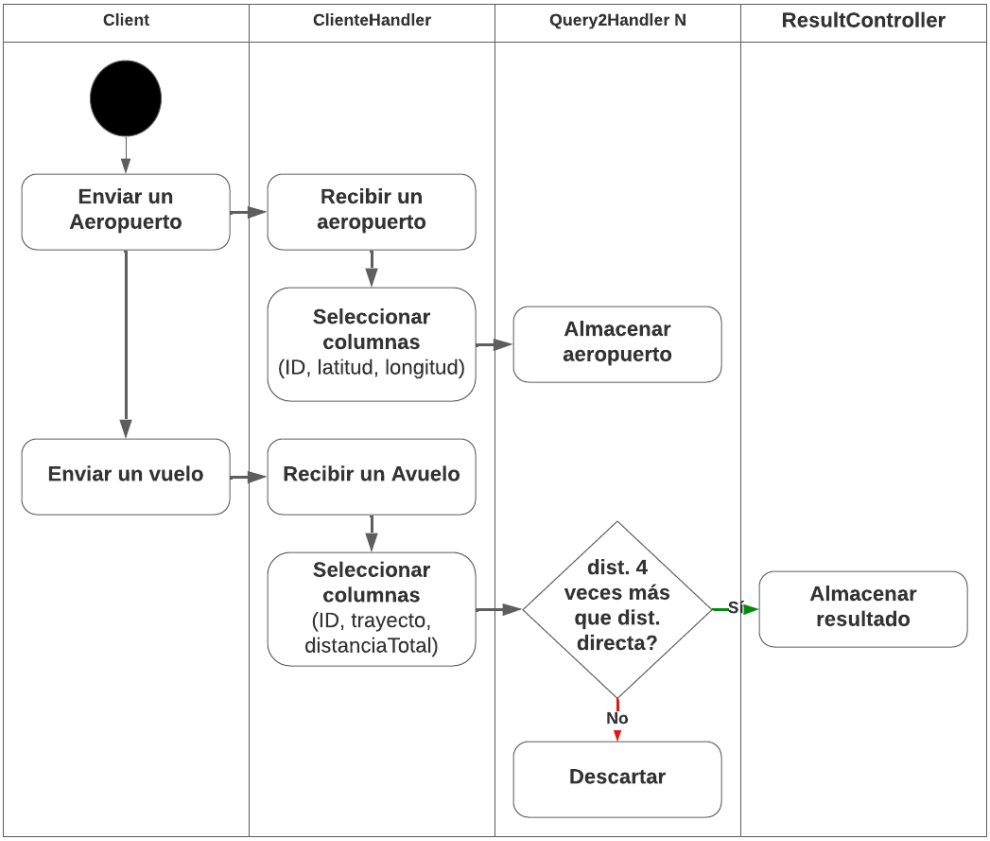

Figura 6: Flujo de mensajes con vuelos para la Consulta 2

4.3. Flujo de mensaje EOF en consulta 4

En el siguiente diagrama se muestra la cadena de mensajes que se envían por el pipeline de la consulta 4. En este escenario sólo existen 2 vuelos que pasan el filtro de la consulta 4. Por lo que luego de recibir un EOF (1) se distribuyen los dos vuelos a los diferentes trabajadores y (2) se envía el EOF a los trabajadores para indicar que ya no habrán más vuelos a procesar. El EOF es enviado por **Query4Handler** con un parámetro especial (inicialmente en cero) el cual indica la cantidad de trabajadores (peers) que leyeron dicho EOF.

Cuando un **Query4Worker** recibe un mensaje de EOF pueden ocurrir dos cosas:

1. Si aún quedan peers a ser informados, envía el EOF sumando uno al contador.
2. Si él es el último, entonces informa a **Query4Synchronizer** que ya no se habrán más resultados a juntar.

En este caso **Query4Worker 1** toma el (EOF, 0), y como tiene información parcial de los resultados primero envía dicha información al sincronizador y luego envía (EOF, 1) a **Query4Worker 2** quien también tiene información parcial que envía al sincronizador. Como **Query4Worker 2** es el último worker, envía (EOF, 0) al sincronizador para informar que ya no habrán más resultados parciales a juntar.

Por último, como se ve en la figura, **Synchronizer** cuando obtiene resultados parciales hace "merge" de dichos resultados con los que ya almacenó y cuando recibe un EOF publica dichos resultados con la tag de 'Q4'.

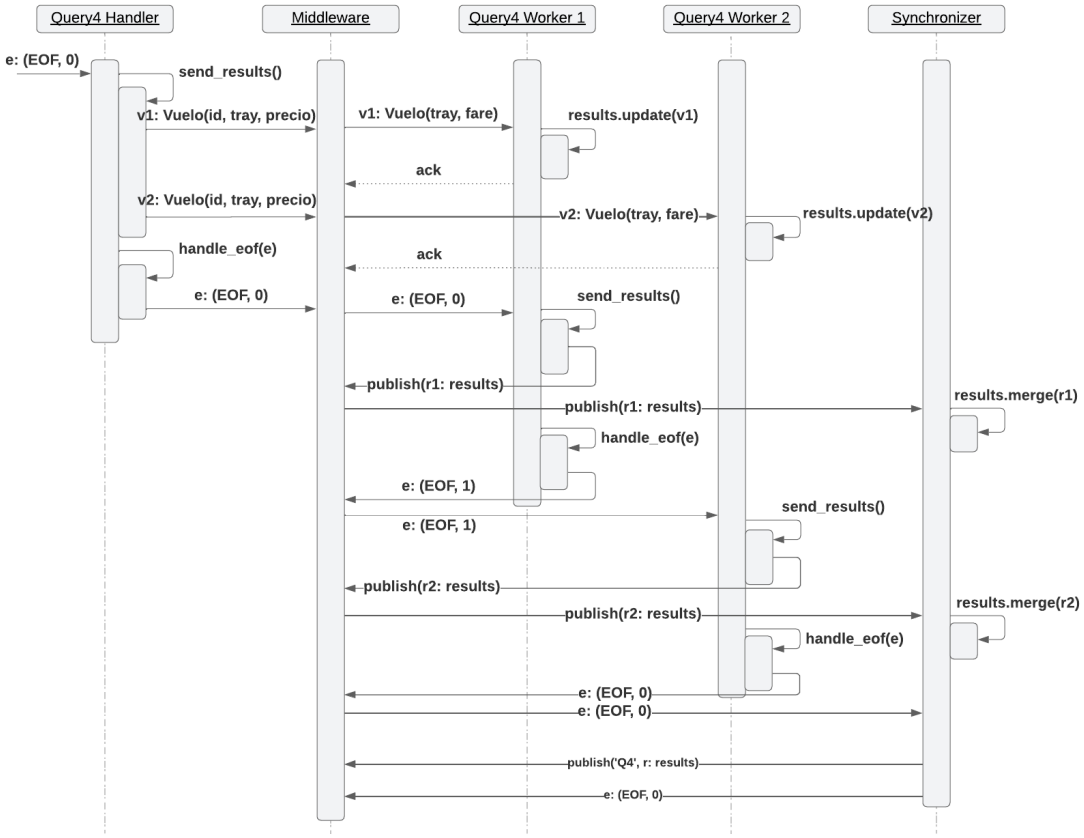

Figura 7: Diagrama de secuencia EOF consulta 4

5. Vista de Desarrollo

5.1. Diagrama de paquetes

Inicialmente se presenta un diagrama de paquetes para observar la estructura del repositorio.

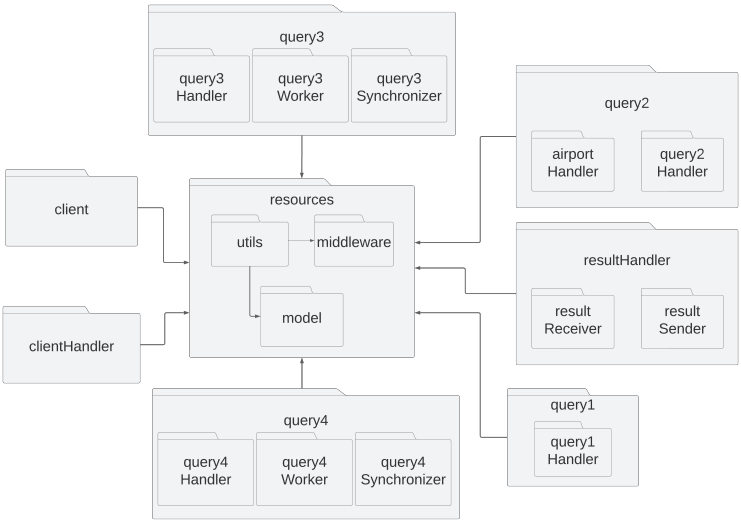

Figura 8: Diagrama de paquetes

5.2. Decisiones de diseño

5.2.1. Protocolo de comunicación

Tanto para la comunicación entre el sistema y el cliente, como para la comunicación dentro del sistema. Desarrollamos nuestro propio protocolo siguiendo una estructura TLV para estructurar los mensajes.

5.2.2. Patrones de diseño

Para implementar la consulta 3 y 4, aplicamos un mismo patrón. Donde los vuelos llegan a un único punto de entrada, y este se encargar de distribuir la carga entre los distintos trabajadores. Luego todos estos resultados parciales se juntan en un único punto, que sera el encargado de enviar los resultados a la parte del sistema que se encarga de guardarlos.

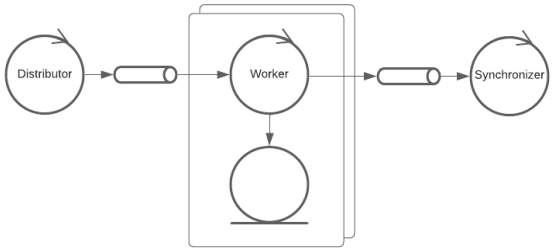

Figura 9: Patrón consulta 3 y 4

6. Vista de Física

6.1. Diagrama de Robustez

A continuación se muestra el diagrama de robustez divido para los distintos flujos dentro del sistema.

En primer lugar se presenta la sección del sistema que incumbe a las consulta 1 (vuelos con 3 escalas o más) y la consulta 3 (los 2 vuelos mas rápidos con 3 escalas o más por trayecto.) Como se puede observar, una consulta depende de los resultados de la otra, de modo que **Query3Handler** escucha los resultados, los vuelos que envían los **Query1Handler** para luego distribuirlos a los **Query3Worker** quienes se encargar de agrupar los distintos vuelo que reciben por su trayecto quedándose son los 2 mas rápidos. Por ultimo, cuando se recibe un mensaje de EOF los **Quey3Worker** envían sus resultados parciales a **Query3ynchronizer** para que sincronice los resultados parciales, agrupando y por último enviando los resultados finales al **ResultHandler**

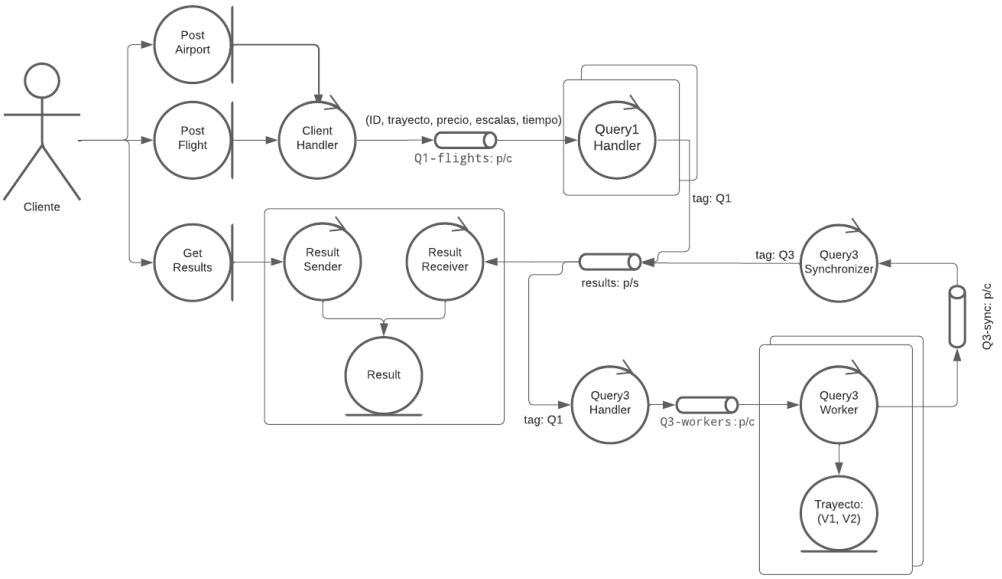

Figura 10: Diagrama de Robustez Q1 Q3.

El segundo caso que presentamos es el de la consulta 2 (distancia total mayor a cuatro veces la distancia directa entre origen/destino). En esta ocasión, podemos observar que el **ClientHandler** se comunica con los **Query2Handler** por medio de dos colas, una por donde los aeropuertos y otra por donde enviara los vuelos. La primera utiliza el modo publisher/subcriber de modo que todas las instancias de **Quey2Handler** obtienen la información de los aeropuertos, de este modo todos tiene una replica de dicha información, aumentando la redundancia en el sistema pero permitiendo manejar las consultas sin tener que acceder a un recurso compartido. Por otro lado la cola por la que se envían los vuelos es producer/consumer, de modo que todos compiten por "trabajar" con dichos vuelos. Por ultimo si un vuelo cumple con la condición pedida, este se envía por una cola de resultados al **ResultHandler**.

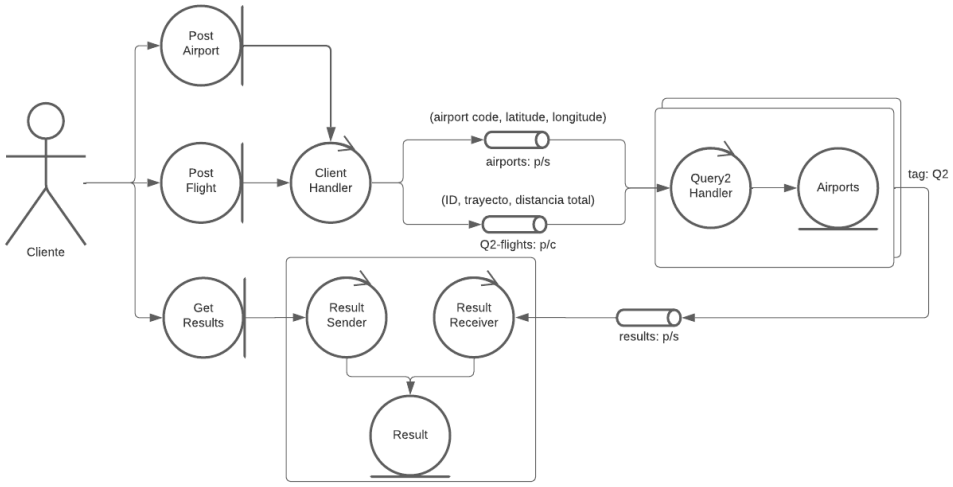

Figura 11: Diagrama de Robustez Q2.

Por ultimo, se presenta la parte del sistema que se encarga de realizar la consulta 4 (precio avg y max por trayecto de los vuelos con precio mayor a la media general). En esta ocasión **Query4Handler** se encarga de almacenar el trayecto y el precio de cada vuelo, para luego filtrar aquellos que poseen precio por encima de la media y los envía a los **QueryWorker**. Dicha media se obtiene una vez leídos todos los vuelos. Luego, el procedimiento continúa de manera similar al del proceso de la consulta 3, donde cada **Query4Worker** calcula sus resultados parciales y luego todos los envían a **Query4Synchronizer** para que este los junte y envíe los resultados finales al **ResultHandler**

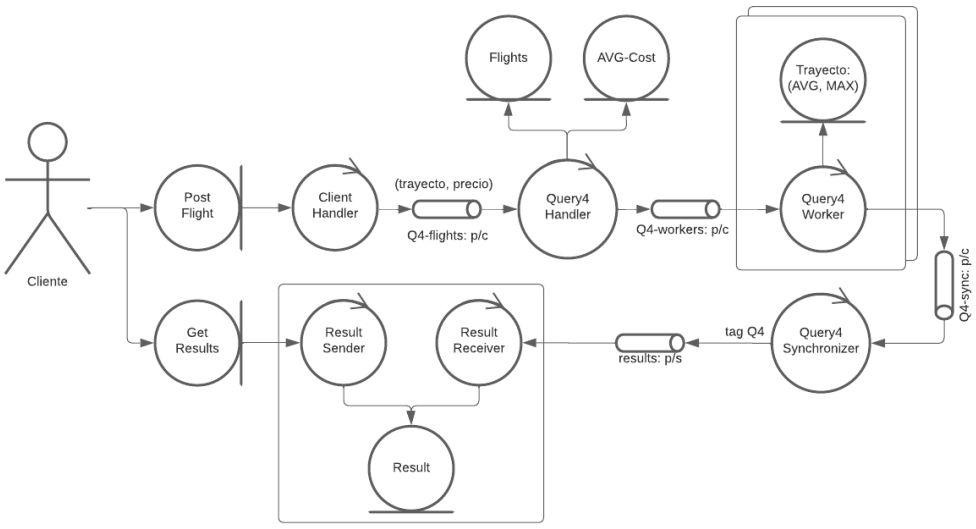

Figura 12: Diagrama de Robustez Q4.

6.2. Diagrama de Despliegue

A continuación se muestran las aplicaciones que se despliegan en el sistema. Como podemos observar, todas ellas se comunican a través del middlerware. Esto quiere decir que el middlerware es un punto único de falla, pero que los distintos componentes no depende entre si.

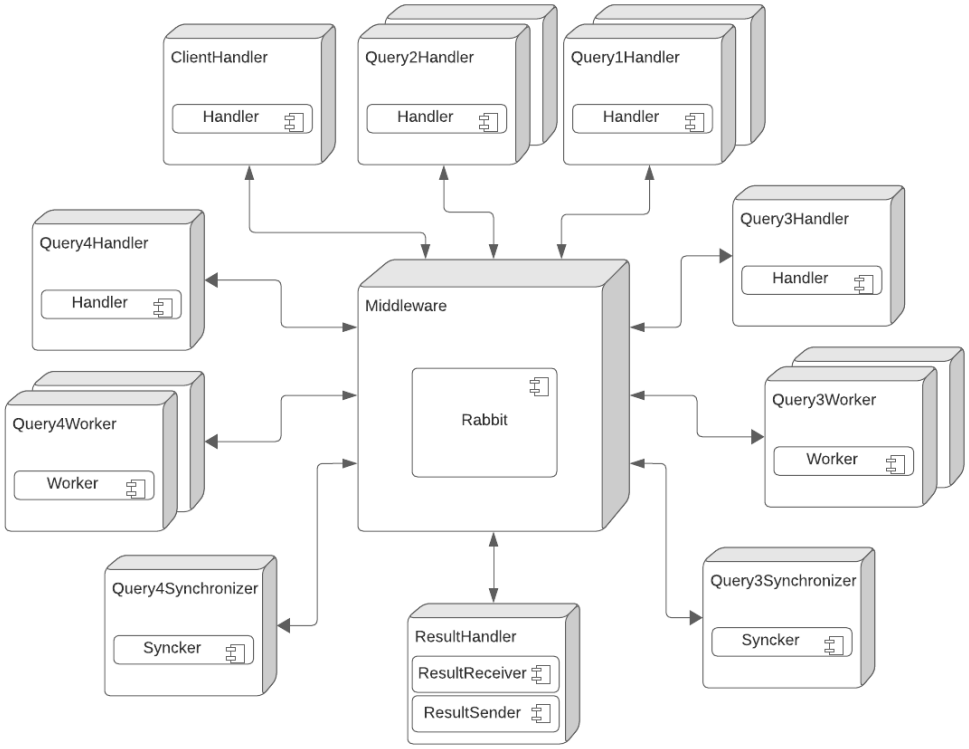

Figura 13: Diagrama de despliegue