Bike Rides Analyzer

TP Individual: Middleware y Coordinación de Procesos

Agustín Miguel Flouret

Alcance

Se solicita un sistema distribuido que analice los registros de viajes realizados con bicicletas de la red pública provista por grandes ciudades.
Los registros cuentan con el tiempo de duración del viaje, estación de inicio y de fin. Se posee también latitud, longitud y nombre de las estaciones así como la cantidad de precipitaciones del día del viaje. Los registros se ingresan progresivamente, al recibirse de cada ciudad.

Se debe obtener los siguientes resultados:

- La duración promedio de viajes que iniciaron en días con precipitaciones >30mm.
- Los nombres de estaciones que al menos duplicaron la cantidad de viajes iniciados en ellas entre 2016 y el 2017.
- Los nombres de estaciones de Montreal para la que el promedio de los ciclistas recorren más de 6km en llegar a ellas.

Escenarios

El siguiente diagrama muestra los tres casos de uso que contempla este proyecto, los cuales corresponden a cada uno de los resultados que se desea obtener del sistema.

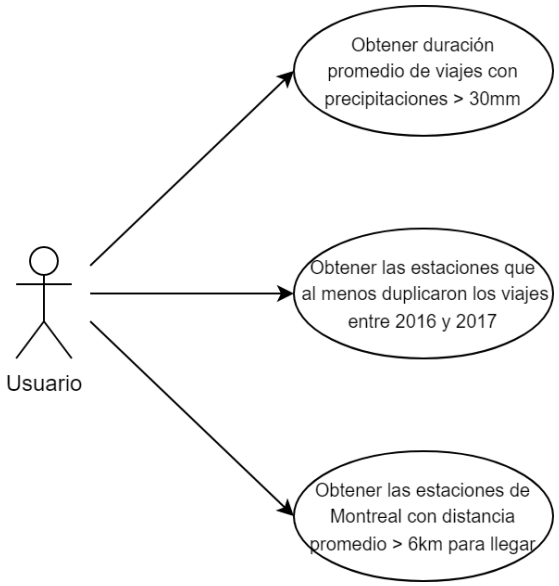

Figura 1: Diagrama de casos de uso

Arquitectura

El sistema desarrollado es un pipeline distribuido de procesamiento de datos compuesto por workers o procesos independientes, cada uno de los cuales tiene una responsabilidad bien definida y se encarga de filtrar o transformar los datos o realizar algún cálculo con el objetivo de obtener los resultados de la query.

El sistema está pensado para funcionar en un entorno multicomputadora, y permite configurar la cantidad de instancias de cada worker para que el sistema sea escalable.
Para la comunicación entre los workers se definió un middleware que utiliza RabbitMQ como broker de mensajes.

Además se implementó una aplicación cliente que permite enviar los datos al sistema a través de una conexión TCP, utilizando un protocolo de mensajes definido para este caso.

Vista lógica

A continuación se muestra el flujo de los datos en el diagrama DAG, que incluye los tres casos de uso. Cada nodo del DAG representa una etapa lógica del pipeline, el cual se ramifica en tres pipelines independientes que permiten obtener los tres resultados finales.

Figura 2: Diagrama DAG

Vista física

En el siguiente diagrama de robustez se muestran todos los componentes del sistema y la comunicación entre ellos:

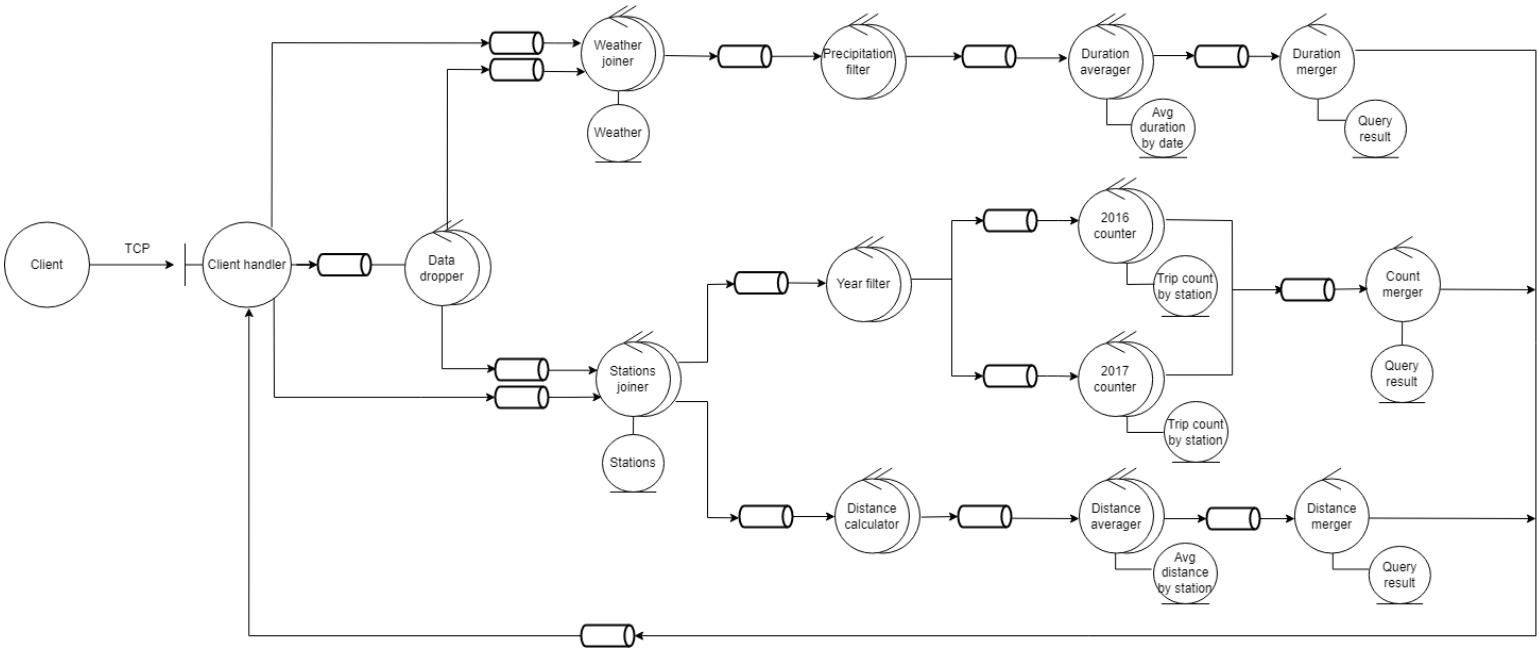

Figura 3: Diagrama de robustez

- **Client:** Envía los datos de los archivos de *trips*, *stations* y *weather* al sistema.
- **Client Handler:** Es la interfaz del sistema, se encarga de recibir los datos del cliente y enviarlo a las colas correspondientes. Al finalizar la ejecución del sistema, lee de una cola los resultados finales del procesamiento para enviarlos al cliente.
- **Data Dropper:** Recibe los *trips* del Client Handler y envía solamente los datos necesarios al Stations Joiner y al Weather Joiner.
- **Weather Joiner:**
    - Recibe los registros de *weather*, de los cuales almacena en memoria los datos de precipitaciones.
    - Cuando termina de recibir los registros de *weather* comienza a leer los *trips* provenientes del Data Dropper, y realiza un join por fecha de ambos datos. El resultado es enviado a la siguiente etapa.
- **Stations Joiner:** idem Weather Joiner, con los registros de *stations* en lugar de *weather*.
- **Precipitation Filter:** Envía a la siguiente etapa solamente los *trips* que ocurrieron en días con precipitaciones mayores a 30.
- **Duration Averager:** Se encarga de agrupar los datos por fecha, calculando el promedio de la duración de los *trips* para cada fecha.

- **Duration Merger:** Calcula el promedio final a partir de los promedios enviados por las instancias del Duration Averager al finalizar el flujo de los datos, y envía el resultado final a la cola de resultados.
- **Year Filter:** Envía a la siguiente etapa los *trips* que se realizaron en los años 2016 o 2017
- **Trip Counter (2016 y 2017):** Almacenan la cuenta de la cantidad trips que se realizaron en un determinado año por cada estación. Hay un trip counter para 2016 y otro para 2017 (a su vez puede haber múltiples instancias de los mismos).
- **Count Merger:** Calcula el total de *trips* por cada año a partir de los resultados de los trip counters. Para el resultado final selecciona únicamente las estaciones que duplicaron la cantidad de *trips* entre 2016 y 2017, y envía el resultado a la cola de resultados.
- **Distance Calculator:** Calcula la distancia recorrida por cada *trip* utilizando la función *haversine*, a partir de las coordenadas de las estaciones origen y destino.
- **Distance Averager:** Se encarga de agrupar los datos por *end_station*, almacenando el promedio de las distancias.
- **Distance Merger:** Calcula el promedio final a partir de los promedios enviados por las instancias del Distance Averager. Luego selecciona las estaciones cuya distancia promedio es mayor a 6km, y envía el resultado a la cola de resultados.

En el diagrama de despliegue se puede observar que cada componente se despliega independientemente, y que toda la comunicación se realiza a través de RabbitMQ, excepto el cliente que se comunica directamente con el Client Handler:

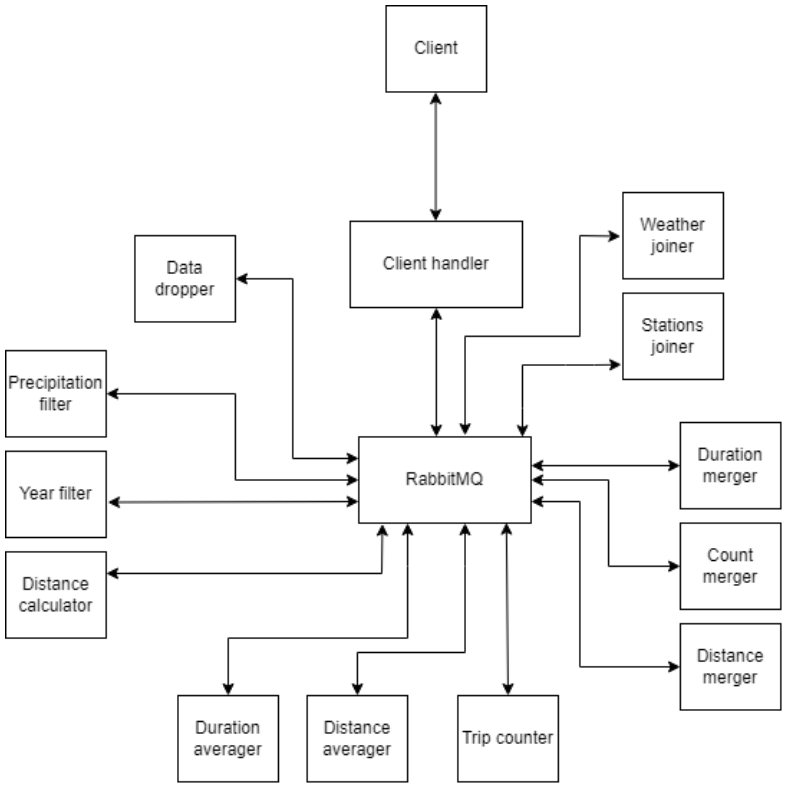

Figura 4: Diagrama de despliegue

Vista de procesos

Como se mencionó anteriormente, el cliente envía todos los *trips*, *stations* y *weather* al sistema. Esto se realiza en un orden secuencial: primero se envían los datos de *stations* y *weather* y luego los *trips*.

En el siguiente diagrama de secuencia se muestra el envío de los *trips* de cada ciudad y la propagación de la información por el pipeline, desde el Client Handler hasta el Duration Averager, el cual almacena el resultado parcial de una de las consultas (obtener la duración promedio de viajes que iniciaron en días con precipitaciones >30mm).

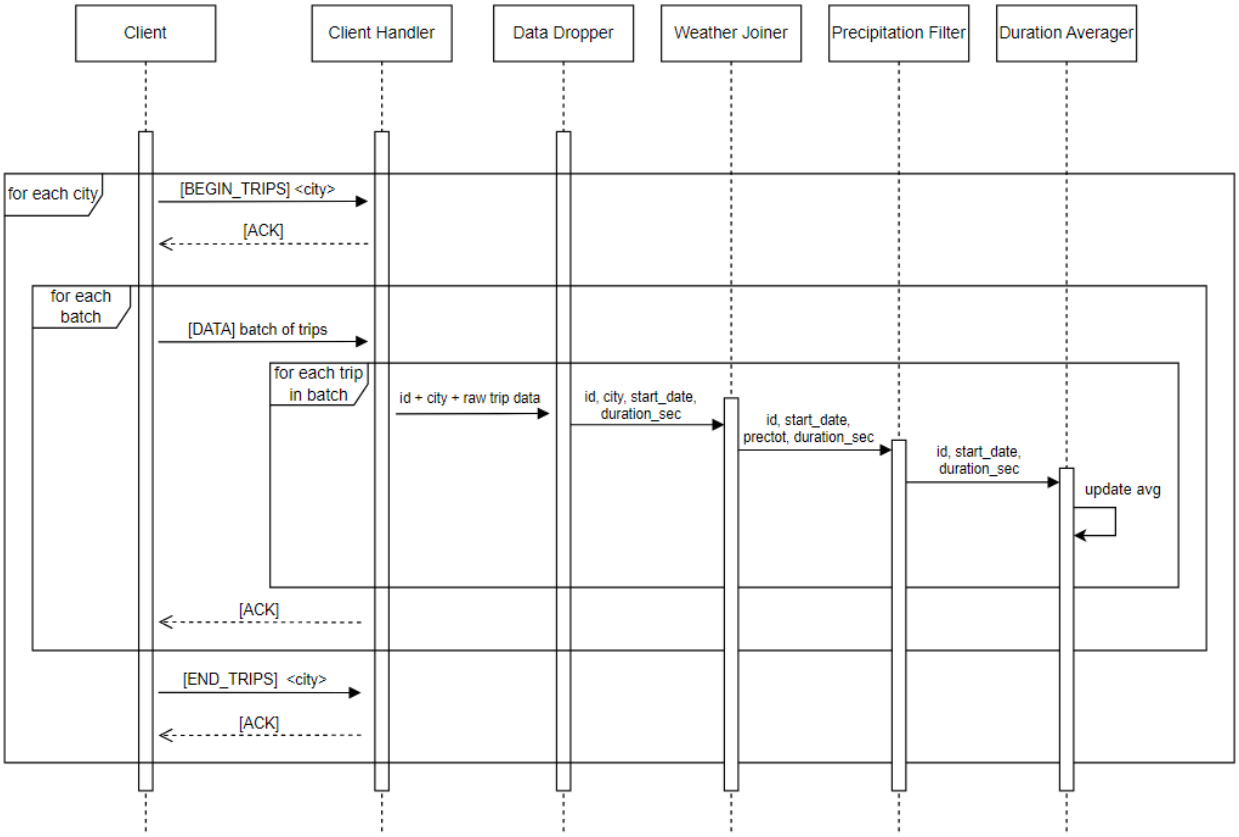

Figura 5: Diagrama de secuencia del envío de los *trips*

Una vez que el cliente termina de enviar los *trips*, envía el mensaje GET_RESULTS. Cuando el Client Handler recibe este mensaje, envía el mensaje EOF a las siguientes etapas, y este mensaje se propaga por el pipeline hasta llegar al Duration Averager. Esto se puede observar a continuación en otro diagrama de secuencia:

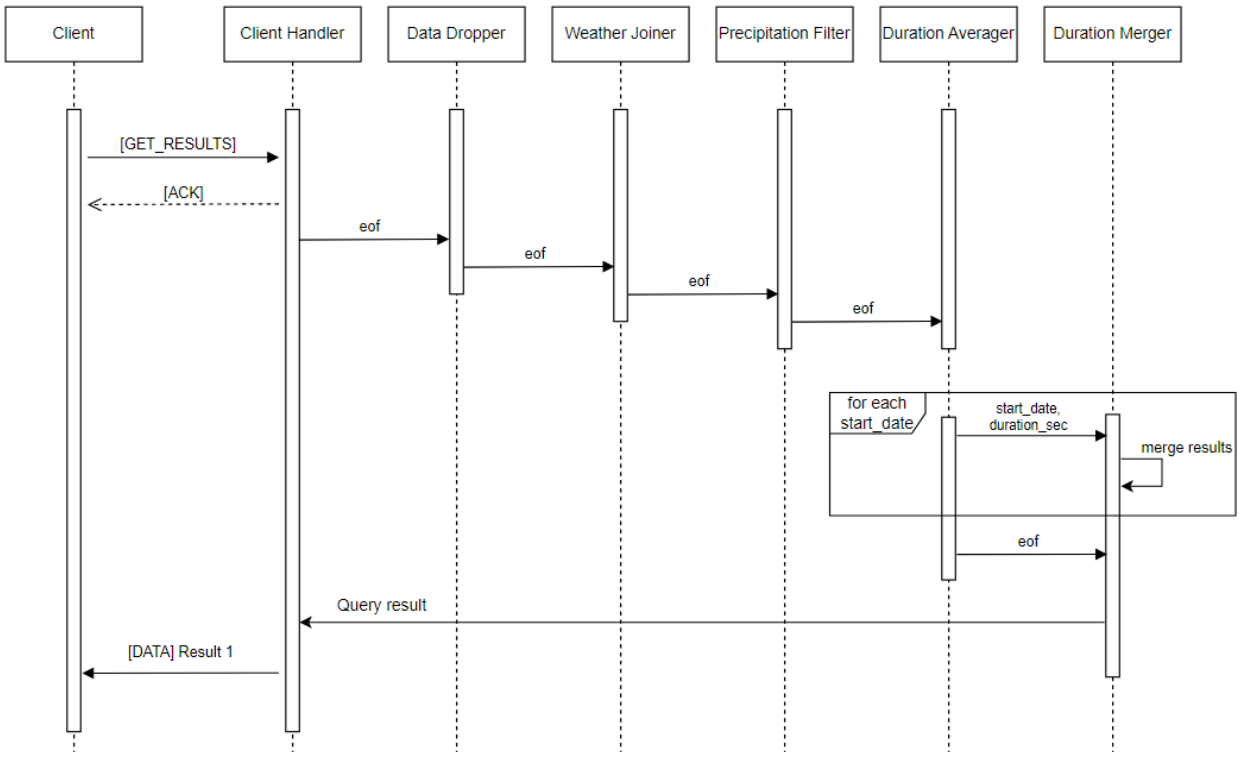

Figura 6: Diagrama de secuencia de la obtención de resultados

Por último, en el siguiente diagrama de actividad se muestra con más detalle el funcionamiento del Weather Joiner y del Precipitation Filter para ejemplificar cómo se sincronizan las diferentes etapas en el pipeline.

Lo primero que se puede destacar es que el Weather Joiner primero lee todos los mensajes de *weather*, y cuando recibe el mensaje EOF comienza a leer los mensajes de *trips* que recibe en otra cola. De esta manera se asegura que el procesamiento de los *trips* comenzará una vez que tenga todos los registros de *weather* almacenados.

Otro punto importante es que cuando un proceso recibe un mensaje EOF durante el procesamiento de *trips*, debe verificar que la cantidad de EOFs recibidos hasta el momento sea igual a la cantidad de instancias de la etapa anterior del pipeline. En caso de que no lo sea, seguirá escuchando en la cola hasta recibir todos los EOFs. Esto permite asegurar de que no quedarán mensajes sin procesar.

Una vez que recibe todos los EOFs, el proceso envía un EOF a todas las instancias de la siguiente etapa y finaliza su ejecución.

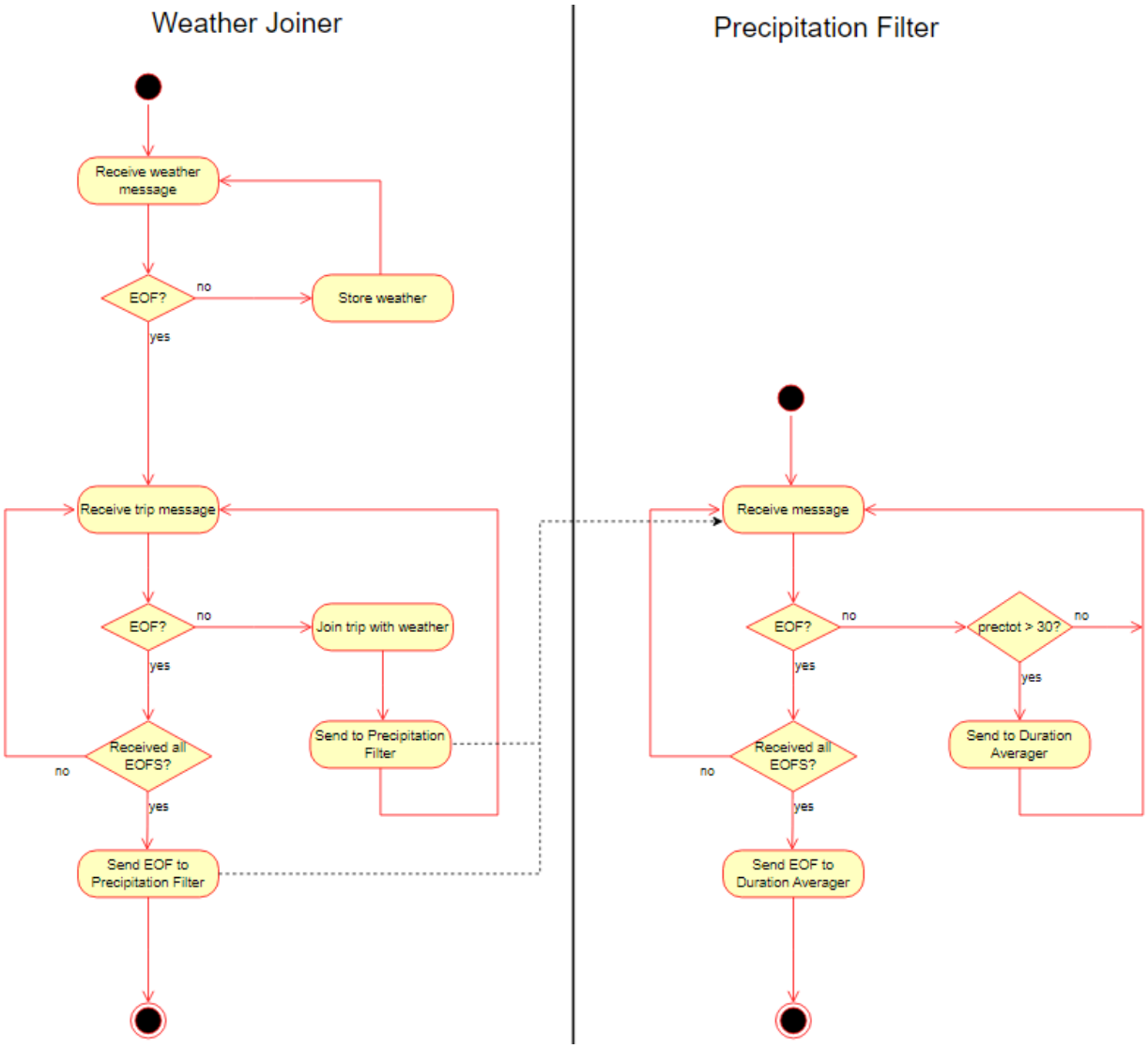

Figura 7: Diagrama de actividad de Weather Joiner y Precipitation Filter

Vista de desarrollo

El sistema está desarrollado en Go, y cada componente del sistema es un paquete. Existen dos paquetes comunes que son importados por el resto de los paquetes:

- **protocol**: implementa el protocolo de comunicación entre el Client y el Client Handler
- **middleware:** implementa los tipos Consumer y Producer, que permiten enviar y recibir mensajes mediante RabbitMQ. Toda la comunicación entre los componentes del sistema pasa por este middleware, excepto la comunicación entre el Client y el Client Handler.

En el diagrama de paquetes se muestran estas dependencias:

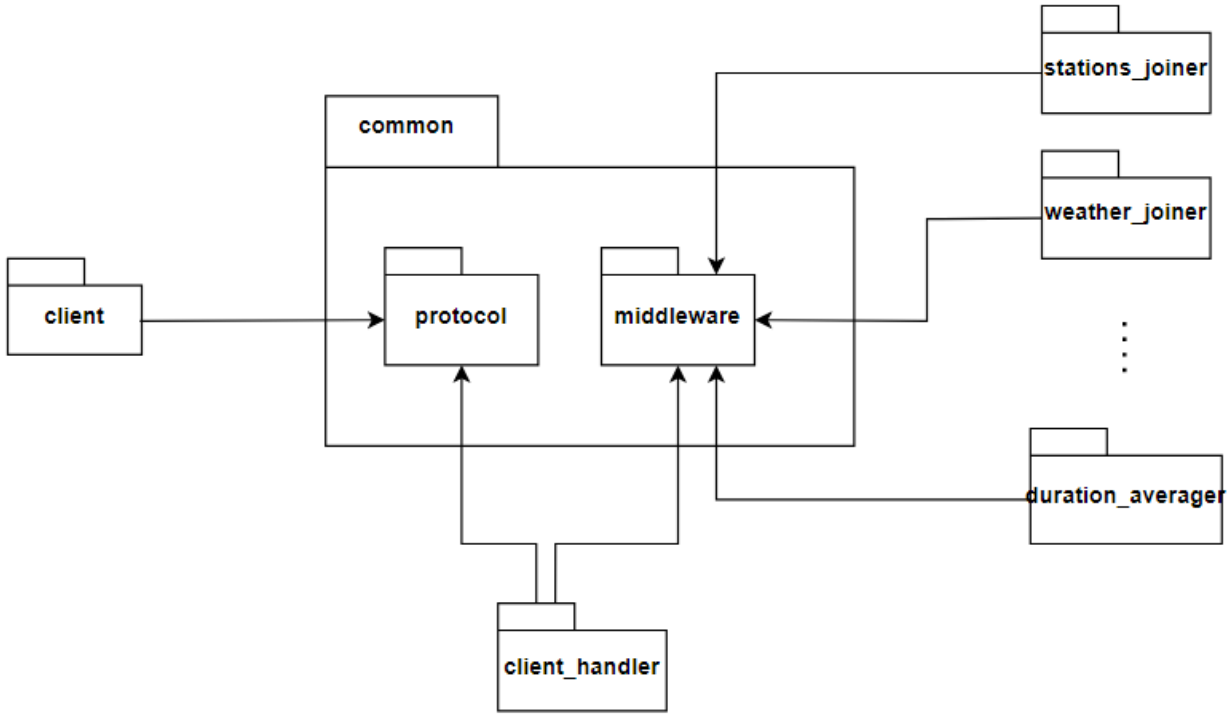

Figura 8: Diagrama de paquetes

Deuda técnica y posibles mejoras

- Enviar datos en batches para reducir la cantidad de mensajes que se envían en el middleware.
- Logging: algunos procesos no loggean su actividad, y los que lo hacen no usan el mismo criterio para loggear.
- Limpieza general del código y comentarios.