.UBAfiuba
FACULTAD DE INGENIERÍA

Sistemas Distribuidos TP N° 1: Middleware y Coordinación de Procesos

1° Cuatrimestre, 2023

Apellido y Nombre	Padrón	Email
Mateo Capón Blanquer	104258	mcapon@fi.uba.ar

Índice

1. Alcance

El presente sistema distribuido resuelve tres consultas en paralelo sobre los viajes en bicicleta que se realizan en la red pública de determinadas ciudades. Permite obtener el resultado de las siguientes tres consultas.

- La duración promedio de viajes que iniciaron en días con precipitaciones mayores a 30mm.
- Los nombres de las 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.

En este sentido, el sistema necesita de la ingesta de registros que caracterizan a las estaciones de bicicleta, la cantidad de precipitaciones, y a los viajes realizados en las ciudades que se desean analizar.

2. Escenarios

El sistema contempla cuatro casos de uso principales, marcados en el diagrama a continuación.

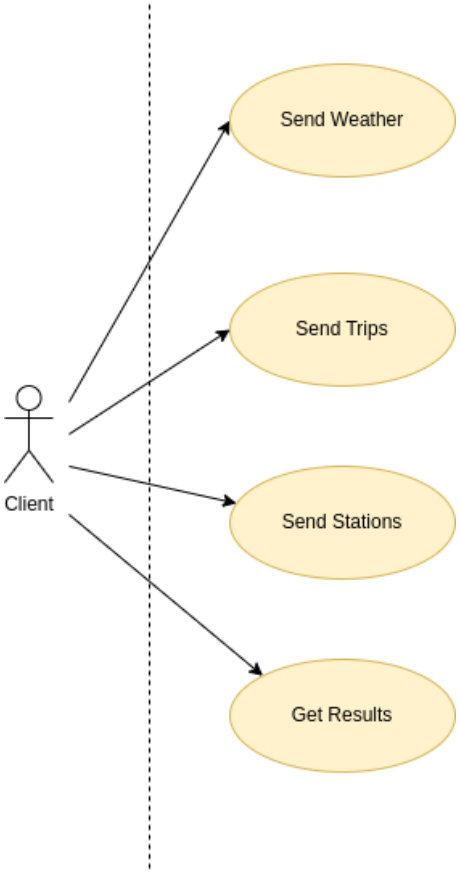

Figura 1: Casos De Uso

3. Arquitectura de Software

En esta sección se presenta una descripción genérica de la arquitectura. Tal como se muestra en la figura 2, el sistema está compuesto por un único cliente, un servidor con el que éste se comunica, y un conjunto de microservicios.

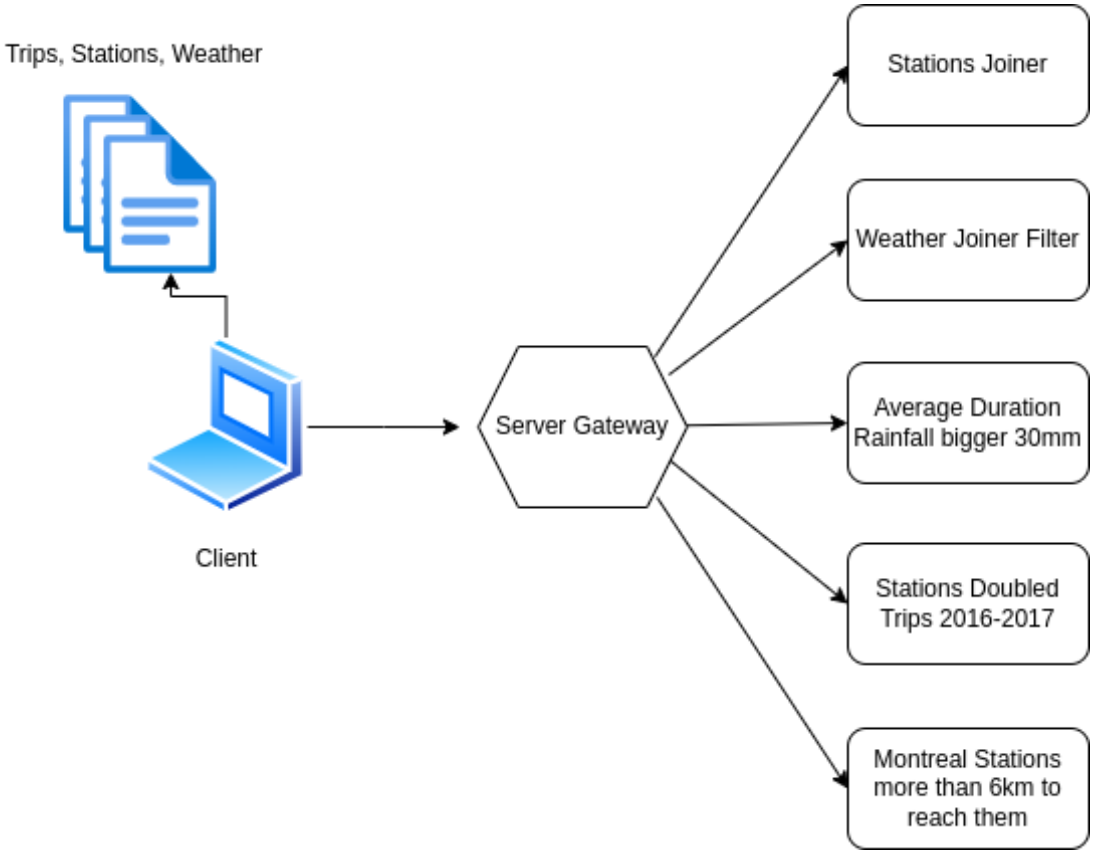

Figura 2: Arquitectura Genérica

El cliente envía la información necesaria para que el servidor pueda procesar las consultas: los archivos con los datos de las estaciones, del clima y de los viajes en las distintas ciudades.

Por su lado, el servidor procesa esta información en sus respectivos microservicios. Los mismos a su vez, están compuestos por múltiples servidores, permitiendo la resolución de las consultas en paralelo.

4. Objetivos de Arquitectura y Restricciones

Se decide crear un único cliente. En un sistema real, los datos estáticos como las estaciones, o semi-estáticos, como el clima, se ingresarían en el sistema por medio de otros clientes. Mientras que la ingesta dinámica, los viajes, se debería ingresar a través de otro(s) cliente(s). Dado la naturaleza del problema a resolver, en el que se necesitan primero los datos estáticos para luego poder resolver las consultas, se decide acoplar el envío de información en un único cliente.

El *Server Gateway* se utiliza para que el cliente tenga un único punto de entrada al sistema. Se busca así, que el cliente sepa únicamente enviar la información requerida, y luego esperar por los resultados.

Se diseñan los microservicios para que el sistema sea escalable a grandes volúmenes de datos,

permitiendo distribuir la carga en múltiples computadoras. En las próximas secciones se hará foco sobre este punto.

Vale aclarar que el presente sistema no es tolerante a fallos. Ante la caída de un servicio, o ante un error crítico detectado, se detiene el procesamiento.

5. Vista Lógica

Se presenta el siguiente diagrama DAG mostrando las relaciones entre los componentes del sistema en el caso de uso de procesar los viajes. Se puede observar como el procesamiento esta dividido en etapas con tareas fácilmente escalables.

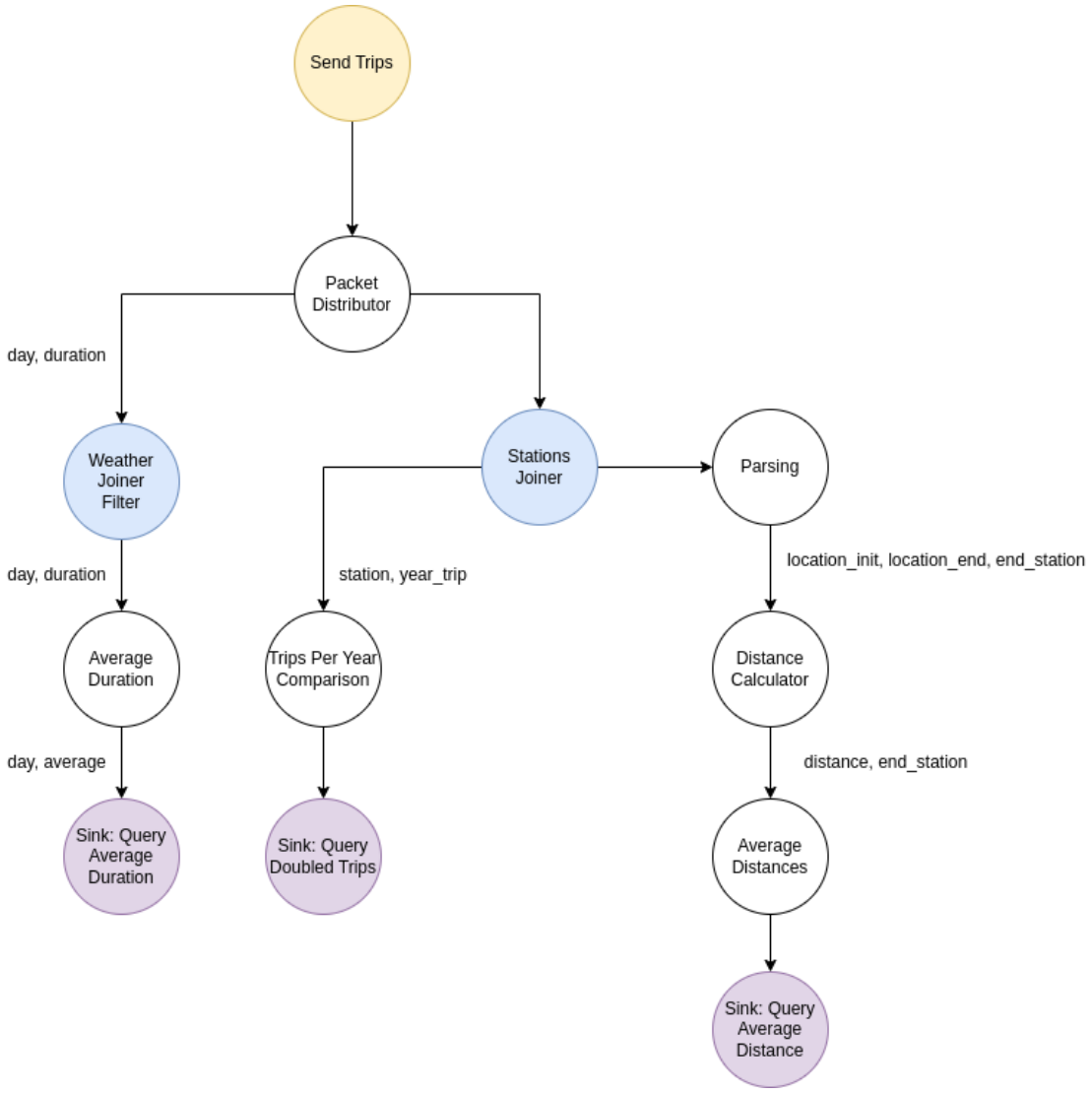

Figura 3: DAG - Envio de Viajes

Mas cercano a la implementación se muestra un diagrama de clases genérico, el cual detalla la relación del negocio con la serialización de los mensajes, y con la capa de middleware. Al dividir estas responsabilidades, es fácil modificar el protocolo, o el Middleware, sin necesidad de modificar el negocio.

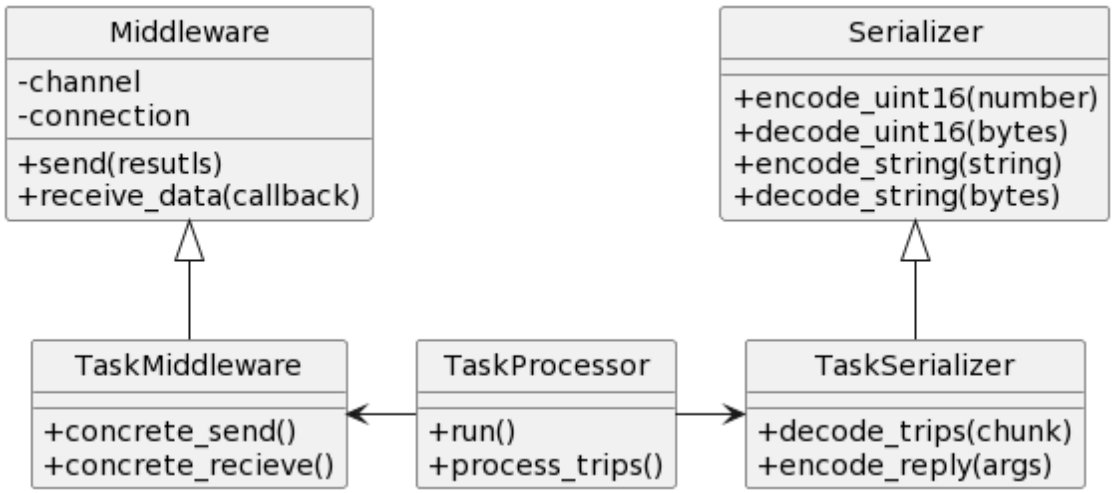

Figura 4: Diagrama de Clases Genérico

Se decidió utilizar Rabbit para la implementación del middleware. Por su lado, el protocolo de serialización y deserialización de mensajes es binario.

El procesamiento de los viajes se hace por batches. Cada paquete contiene un tipo de paquete. Luego, los otros datos del paquete tienen una estructura similar para respetar un protocolo común.

Existen determinadas etapas del DAG que pueden ser genéricas para queries que surjan en el futuro. Por ejemplo, es posible que más de una query quiera hacer un calculo de distancias.

Por esta razón, al enviar un batch que debe ser consumido por uno de estos procesos, se envía también el nombre del proceso que consumirá el resultado (un identificador que permite decidir a donde enviar los resultados). De este modo, la calculadora de distancias, o el joiner, no necesitan conocer ni el proceso del cual consumen mensajes, ni el proceso al cual le producen los resultados.

La implementación con rabbit se realizó aplicando el patrón Routing o Topics, según el proceso. A continuación se muestra el caso particular del calculador de distancias.

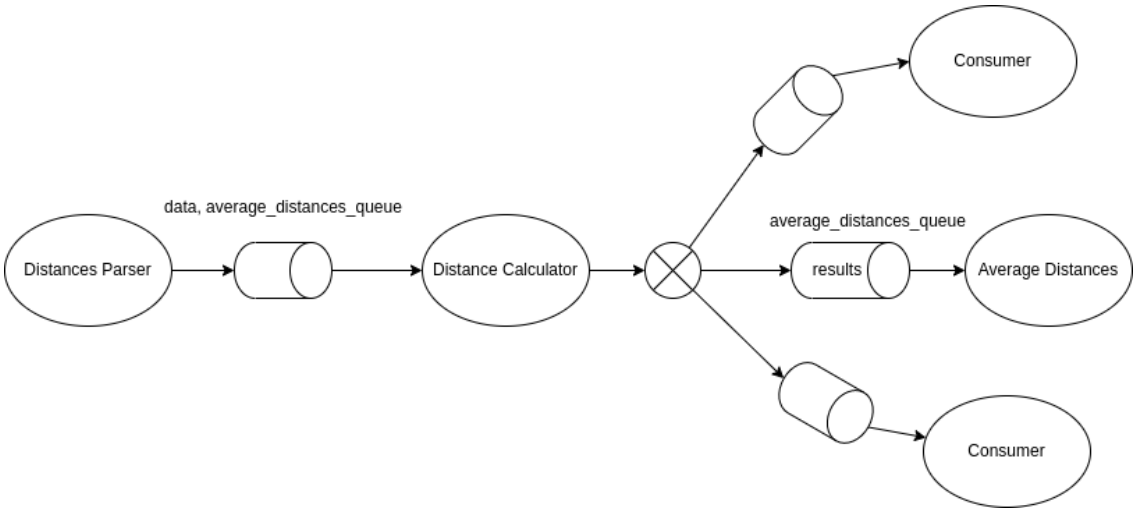

Figura 5: Patrón Routing para envío de mensajes

6. Vista Procesos

A continuación se observa cómo se resuelve el pasaje de información entre los procesos para la etapa de finalización del envío de viajes, y comienzo de recolección de resultados.

Se puede pensar al sistema como como un pipeline que se bifurca en distintos pipelines. Es necesario que todos los productores de un consumidor finalicen sus tareas, para que luego el consumidor pueda finalizar.

La responsabilidad de serializar el orden en el que los procesos finalizan está en el proceso *EOF Manager*. Este se encarga de esperar el EOF de la primera capa de productores (*Server Gateway*). Luego, broadcastea el EOF a la segunda capa de productores (*Packet Distributor*). Una vez que los *Packet Distributor* confirman que finalizaron, se continúa con la siguiente capa de productores. Y así, hasta que terminan todos los procesos. Una primera etapa de este protocolo de finalización se puede observar en el siguiente diagrama.

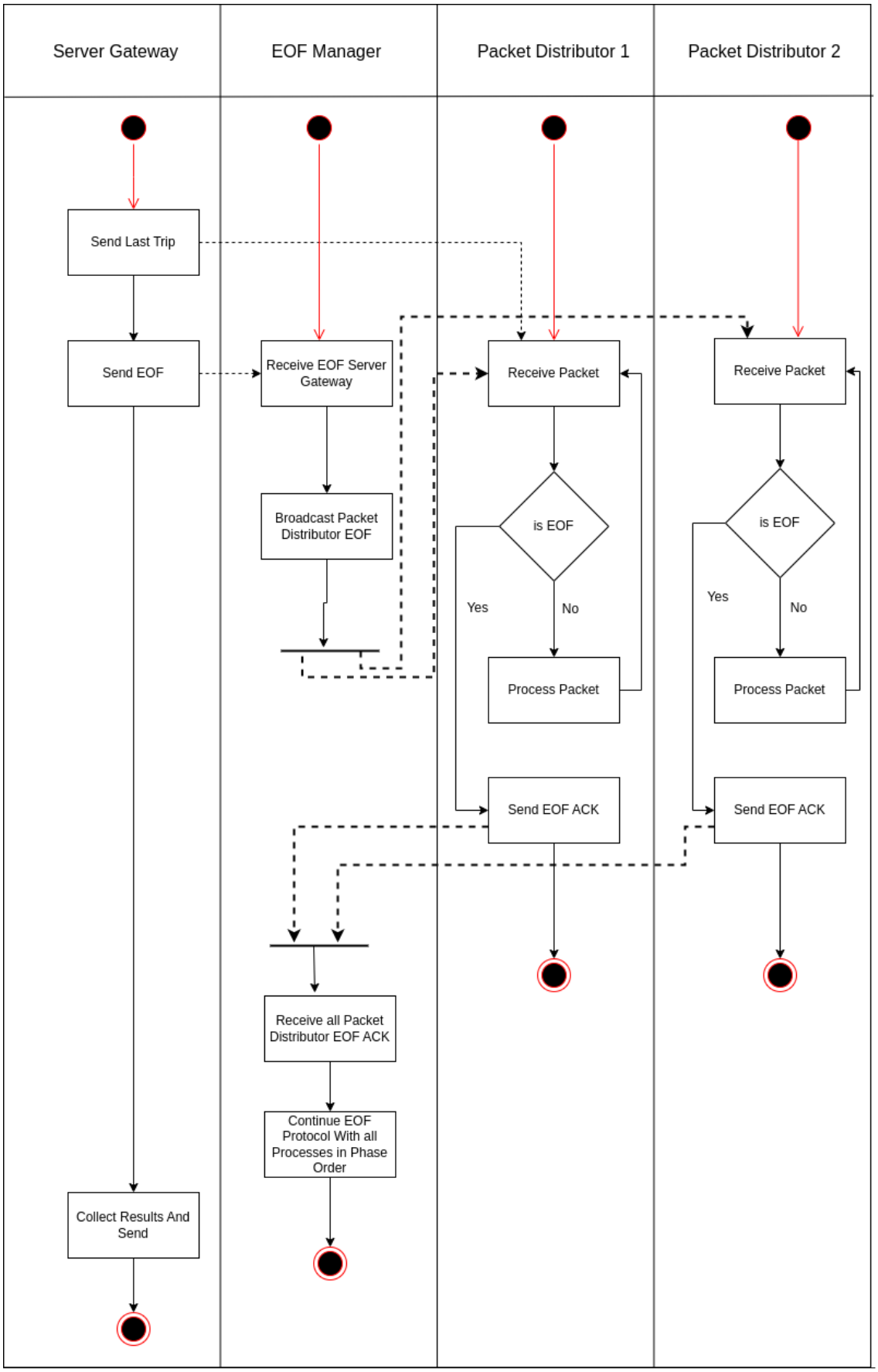

Figura 6: Diagrama Actividad - Envío de último Viaje - EOF

A continuación se muestra cómo se resuelve la obtención de resultados por parte del cliente. El mismo hace polling de resultados, hasta tanto se encuentren todas las consultas procesadas.

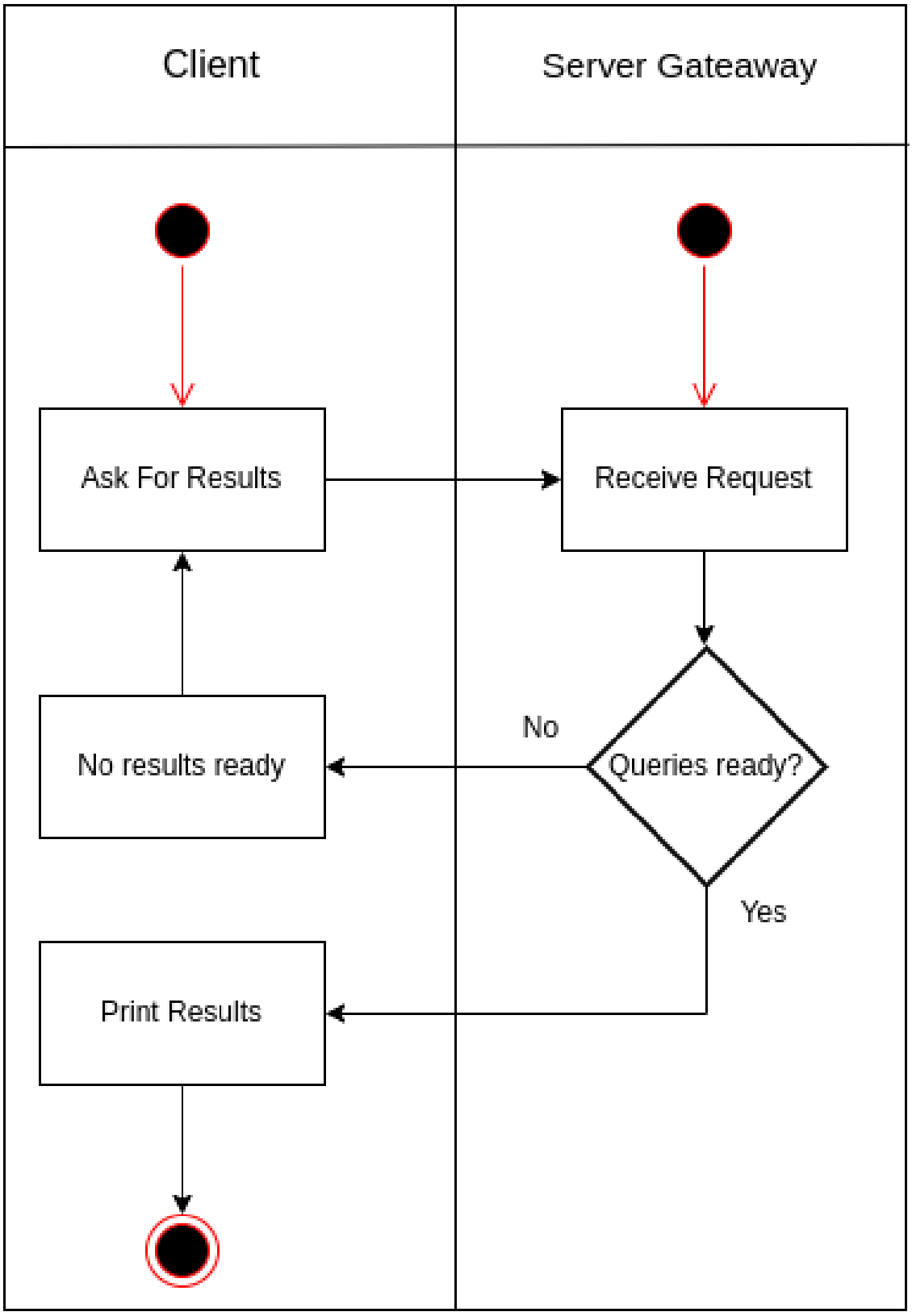

Figura 7: Diagrama Actividad - Polling de resultados

Haciendo foco en la implementación de los procesos, se muestra un diagrama de secuencia que expone un orden genérico de un proceso que recibe un paquete, lo procesa y envía los resultados que obtiene. Algunos procesos que pueden ser catacterizados por este diagrama genérico son *Packet Distributor*, *Weather Joiner Filter*, *Stations Joiner*, *Distance Calculator*, etc.

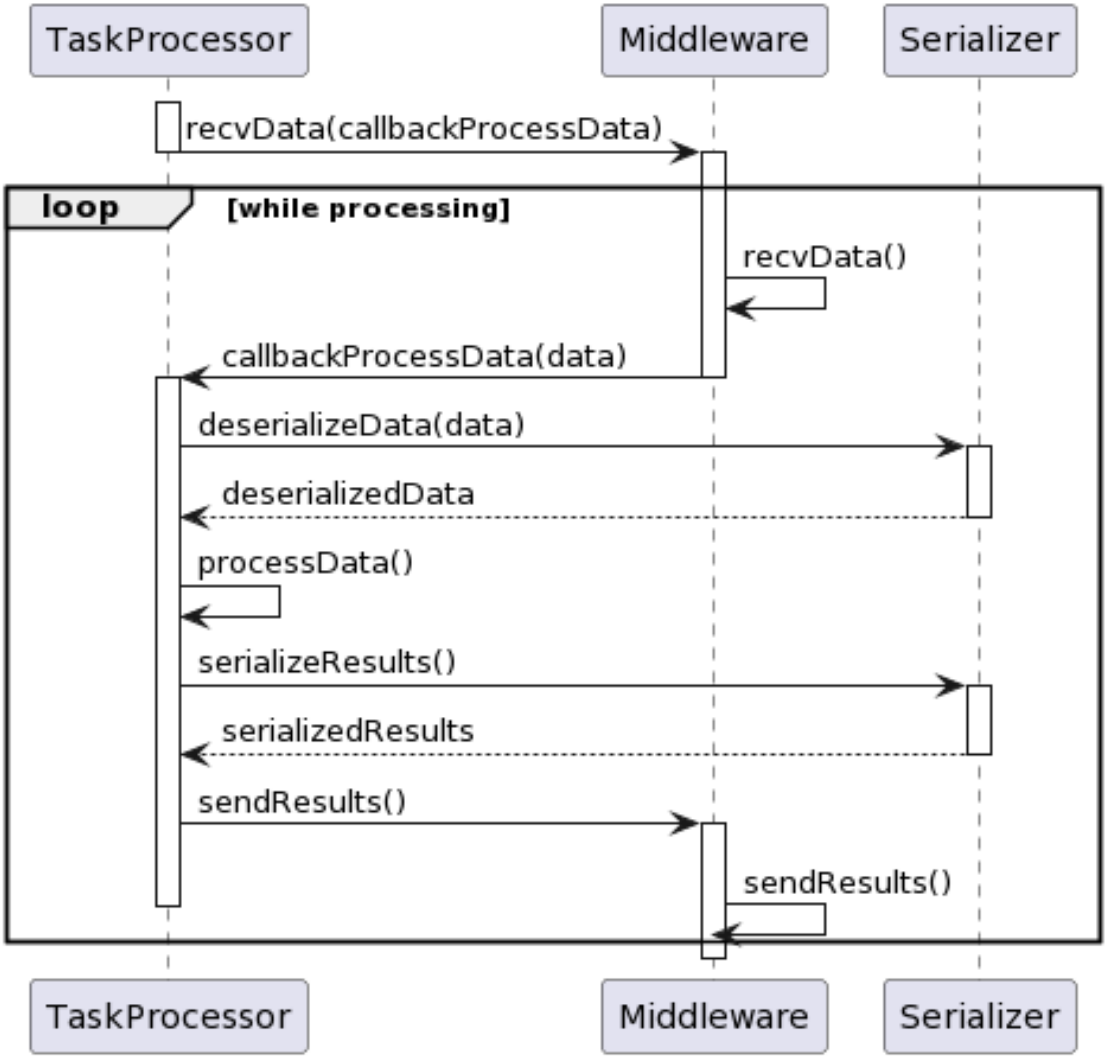

Figura 8: Diagrama Secuencia - Proceso Genérico que Ejecuta una Tarea

7. Vista Desarrollo

El sistema se desarrolla en un único repositorio. El link al mismo es el siguiente:
https://github.com/mateocapon/sistemas-distribuidos-tp1

A continuación se muestra el diagrama de paquetes. Todos los directorios incluyen al paquete común middleware, el cual facilita la comunicación.

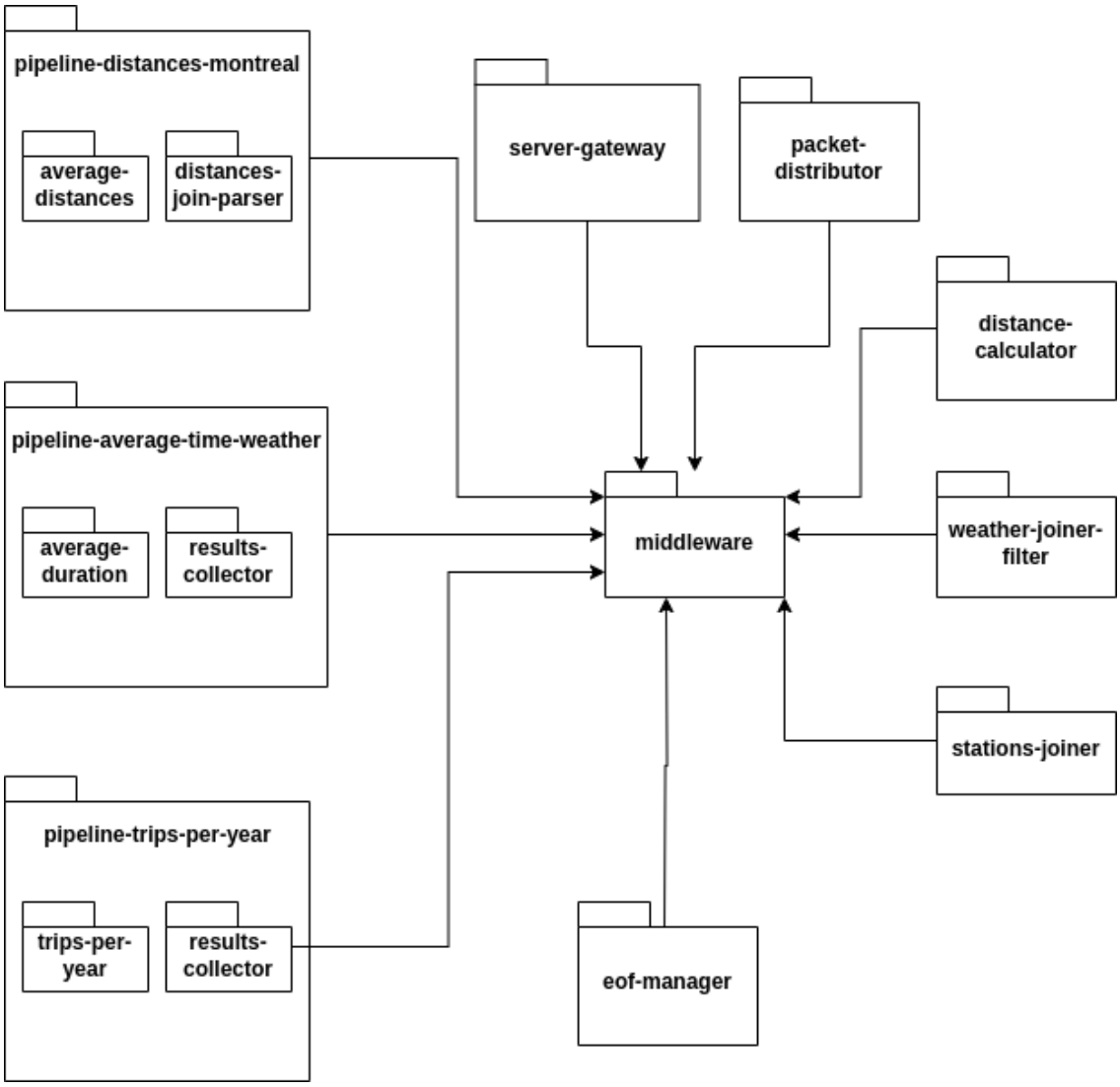

Figura 9: Diagrama de Paquetes del Servidor

En el archivo README.md del repositorio se puede observar cómo correr el sistema con docker. Los resultados se obtendrán en la carpeta *client/results*.

8. Vista Física

8.1. Diagramas de Robustez

Se divide a los diagramas de robustez con respecto a los casos de uso. En primer lugar, se muestra como se resuelve el envío de los registros del clima.

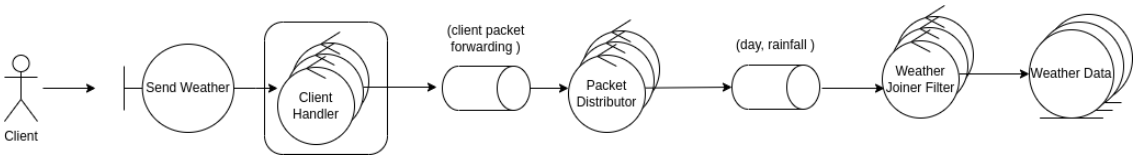

Figura 10: Envío de Clima

De modo simétrico se resuelve el envío de los registros de las estaciones.

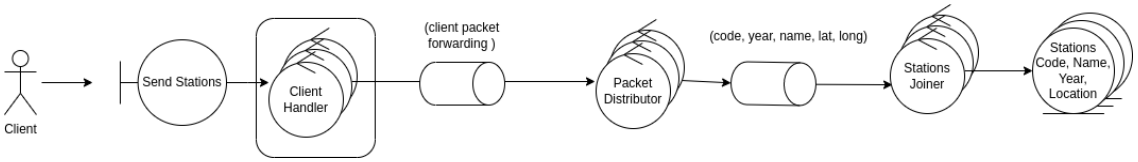

Figura 11: Envío de Estaciones

Una vez que se envían los datos, el sistema pasa al estado de esperar por los registros de los viajes. Se separa el accionar de cada pipeline en un diagrama distinto.

En primer lugar se presenta la resolución de la consulta de la duración promedio de viajes para las fechas con precipitaciones mayores a 30mm.

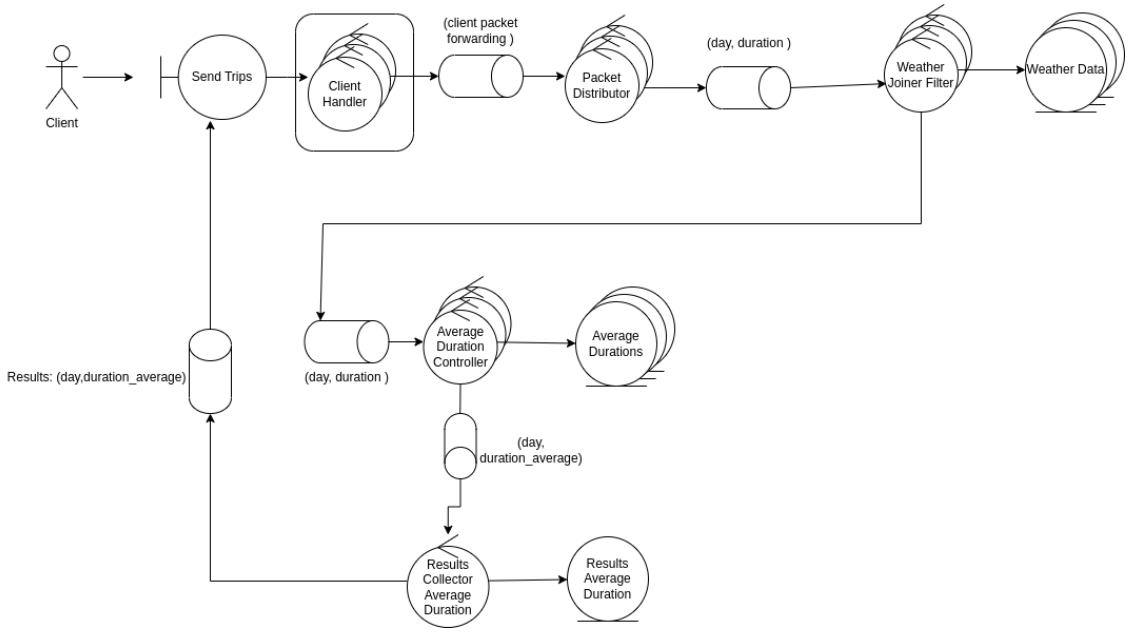

Figura 12: Envío de Viajes - Caso Duración Promedio con Filtro por Precipitaciones

En segundo lugar se presenta el diagrama que resuelve la consulta que busca el nombre de las estaciones que duplicaron la cantidad de viajes entre 2016 y 2017.

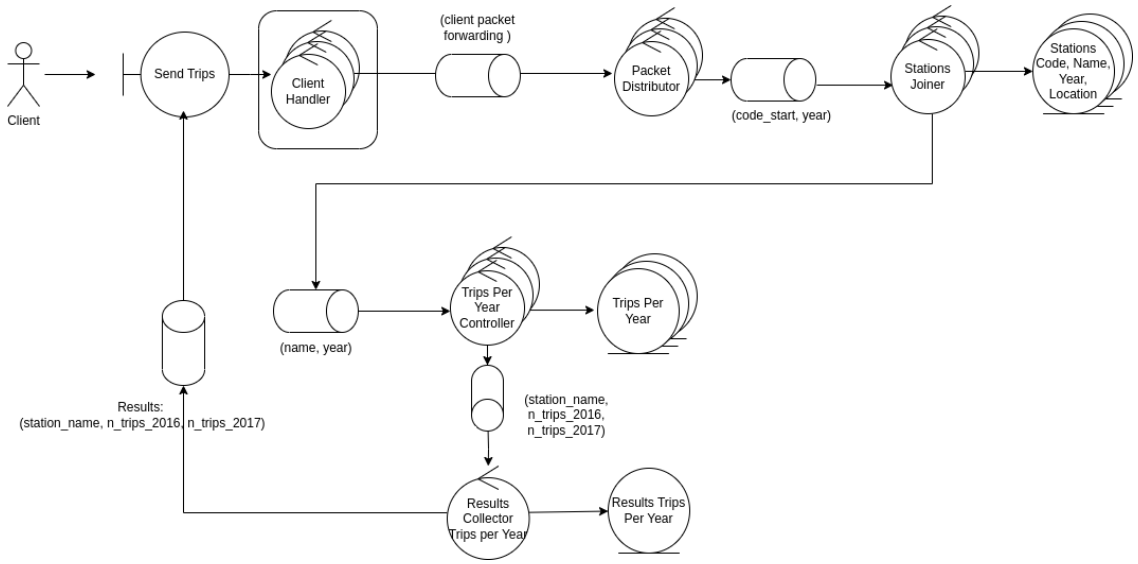

Figura 13: Envío de Viajes - Caso Estaciones que duplicaron Viajes

Por último, el pipeline que calcula las estaciones de Montreal a las que en promedio se recorren más de 6km para llegar a ellas.

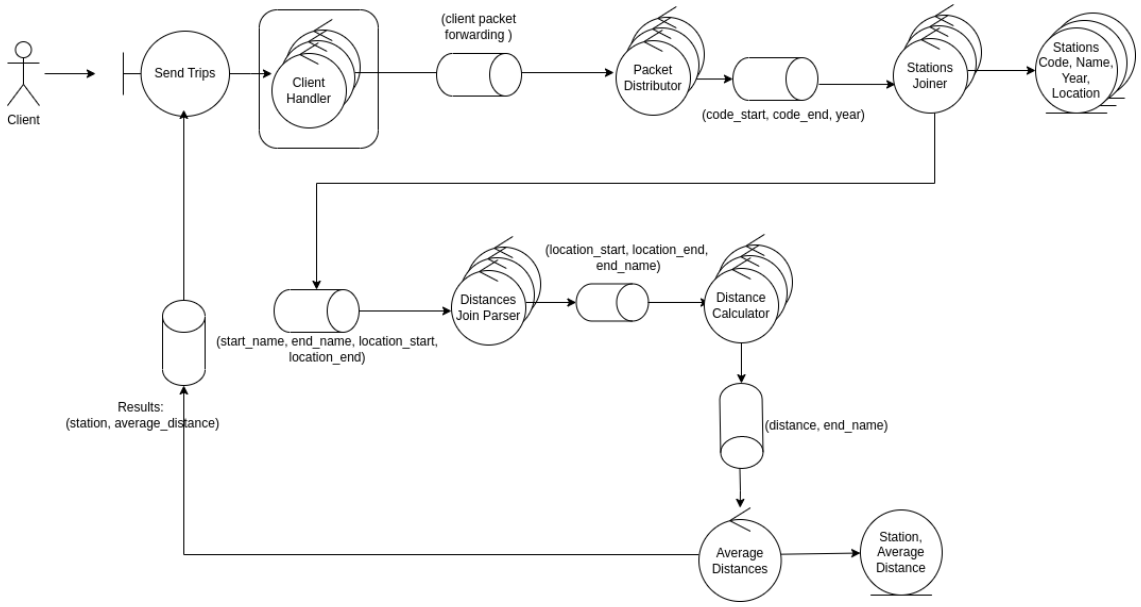

Figura 14: Envío de Viajes - Caso Distancias en Montreal

El único proceso que no se mostró en los anteriores diagramas es el *EOF Manager*. En el diagrama a continuación se muestra cómo el mismo envía los mensajes de EOF en las colas donde se consumen los mensajes de los procesos. De este modo, el mensaje de EOF se procesa una vez que se hayan procesado todos los anteriores mensajes.

Vale remarcar que el *EOF Manager* se comunica con muchos controladores más del sistema, que no fueron agregados en el diagrama, para evitar la confusión del lector.

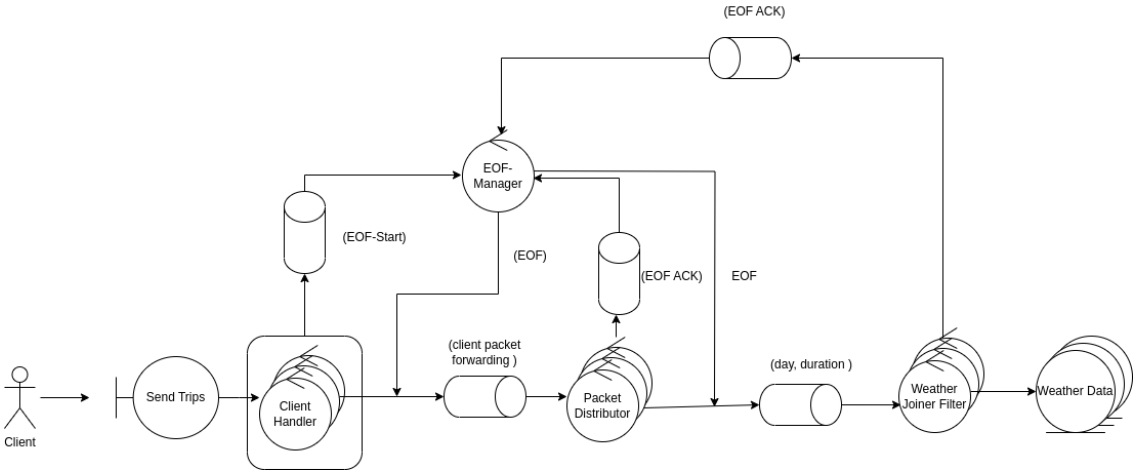

Figura 15: EOF Manager - Comunicación con los otros Procesos

8.2. Diagramas de Despliegue

A continuación se muestran las aplicaciones que se despliegan en el sistema. El primer diagrama muestra los procesos que participan del cálculo de la duración promedio de viajes para las fechas con precipitaciones mayores a 30mm.

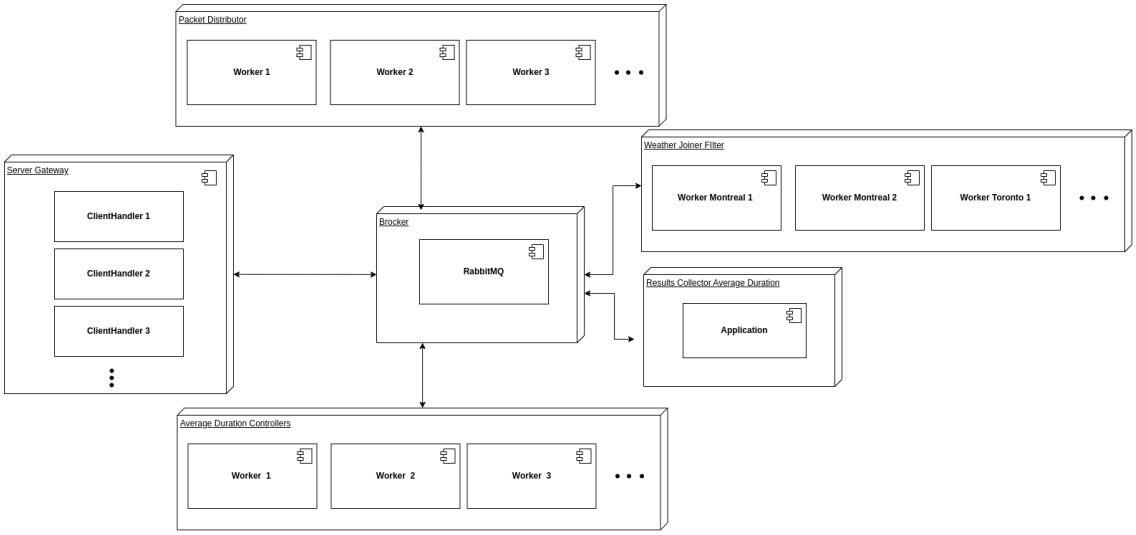

Figura 16: Diagrama de Despliegue - Caso Duración Promedio con Filtro por Precipitaciones

El segundo, el nombre de las estaciones que duplicaron la cantidad de viajes entre 2016 y 2017.

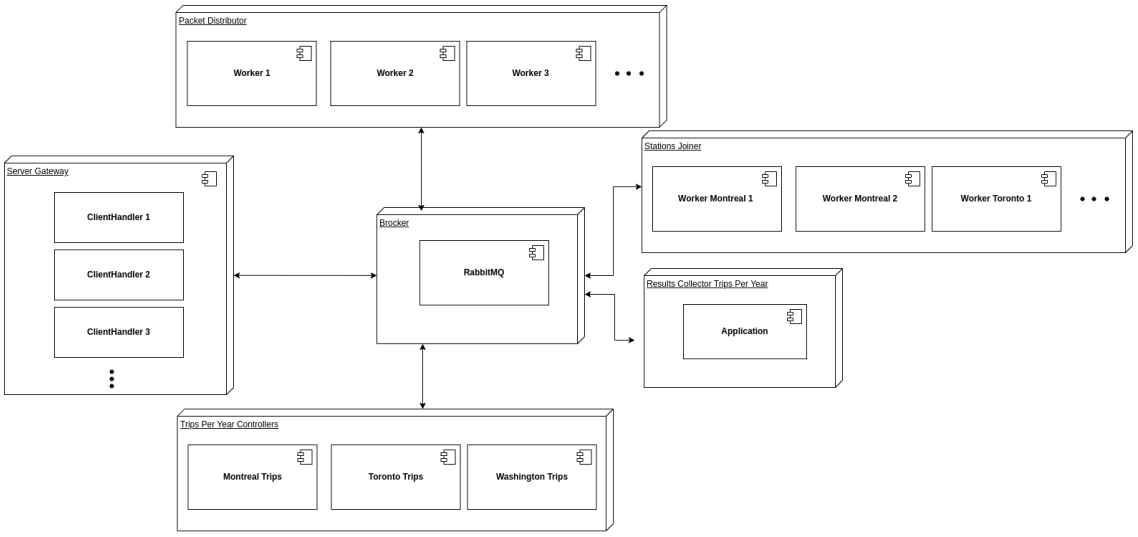

Figura 17: Diagrama de Despliegue - Caso Estaciones que duplicaron Viajes

El tercero, las estaciones de Montreal a las que en promedio se recorren más de 6km para llegar a ellas.

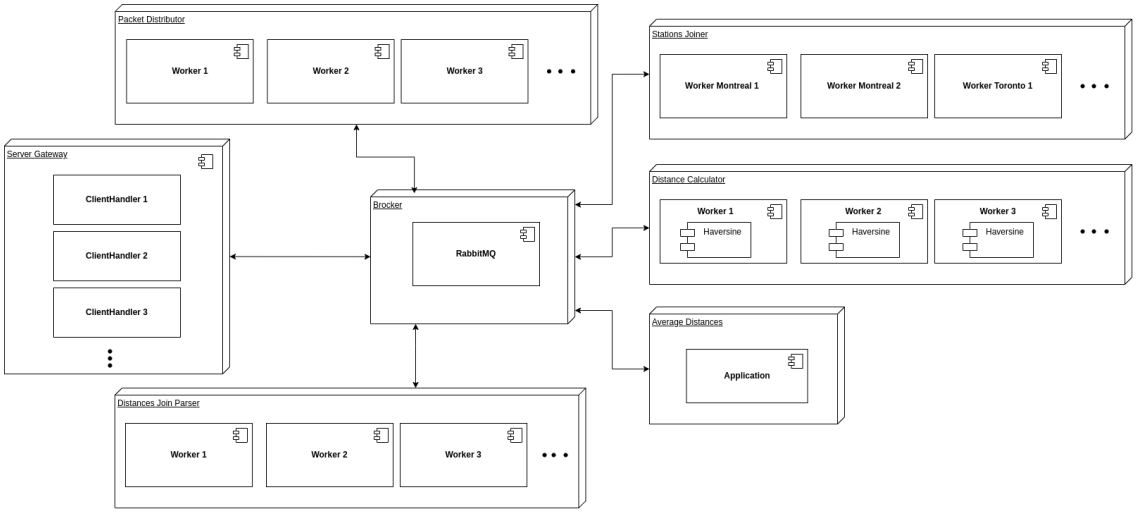

Figura 18: Diagrama de Despliegue - Caso Distancias en Montreal

Por último, el *EOF Manager* también es un proceso que se debe desplegar para el correcto funcionamiento del sistema.

9. Performance y Escalabilidad

Tal como se observa en los distintos diagramas presentados en el informe (principalmente los de robustez y despliegue), el sistema es altamente escalable.

A pesar de que los *Joiner* procesen por ciudad (afinidad), se pueden escalar teniendo los datos estáticos repetidos en múltiples procesos.

En cuanto a la *Performance* del sistema se debe reconocer que el proceso que más demora en ejecutar un batch es el que calcula distancias usando la función *haversine*. Esto se identificó

utilizando la consola de Rabbit, observando el campo que indica el diferencial de paquetes entrada / salida en este proceso, relacionandolo con la cantidad de procesos que se despliegan.

Al observar este suceso, se decidió hacer al proceso más performante, con la contra de que tuve que acoplar al serializador con el cálculo de las distancias (no con el middleware). Si se quiere ver la diferencia de performance entre ambos códigos, se puede modificar una variable de configuración. Esta variable de configuración permite decidir correr al proceso con la versión performante (pero con peor diseño), o con la versión que ofrece una división de responsabilidades más clara, pero es al menos unas veces más lento.

Una de las diferencias en Performance entre ambos casos es que la versión no performante hace al menos dos recorridos del batch, mientras que la primera no.