TP Individual
Middleware y Coordinación de Procesos

[75.74]
Sistemas Distribuidos

Alumno	Padrón	Email
Fanciotti, Tomás	102179	tfanciotti@fi.uba.ar

Índice

1. Introducción

El presente trabajo práctico muestra una propuesta de solución de un diseño de una arquitectura distribuída para el procesamiento (en el caso ideal productivo) de grandes volúmenes de datos utilizando Message Oriented Middlewares y patrones de procesamientos como Pipe & Filters.

En particular, este sistema esta orientado a resolver tres consultas predefinidas para el siguiente set de datos: kaggle.com/datasets/jeanmidev/public-bike-sharing-in-north-america.

2. Diseño de la arquitectura

Se plantea un diseño inicial que es descripto utilizando el patron de documentación $4+1 Views$, en el que se recorreran las distintas perspectivas de la implementación.

2.1. Vista de Escenarios

Los casos de uso son principalmente las consultas que el cliente debe hacerle al servidor luego de procesar todos los datos.

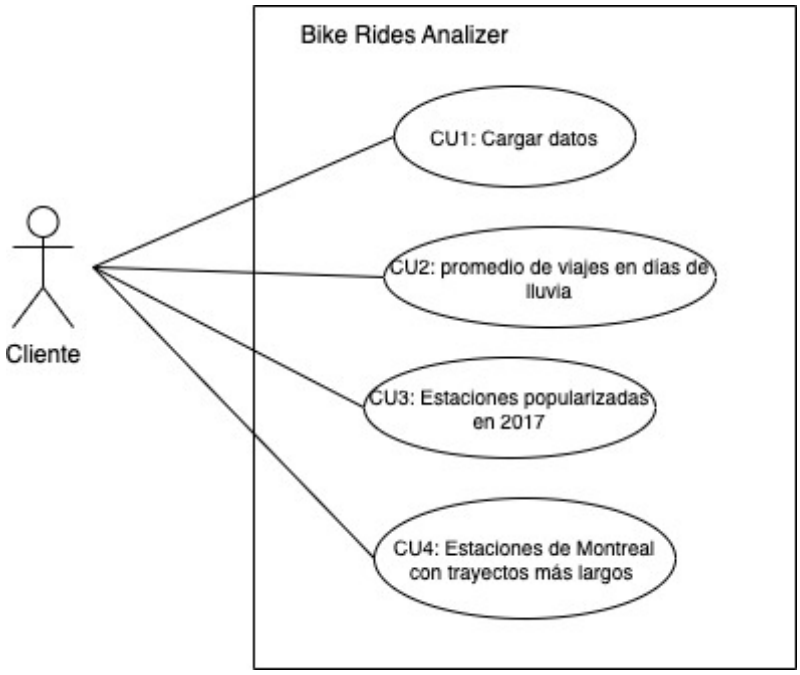

Figura 1: Diagrama de casos de uso

- **CU2: Viajes en dia de lluvia** (Query #1: Duración promedio de los viajes en días de lluvia con mas de 22mm de precipitaciones.
- **CU3: Estaciones popularizadas en 2017** (Query #2): Los nombres de aquellas estaciones que del año 2016 al 2017, duplicaron los viajes iniciados en ellas.
- **CU4: Estaciones de Montreal** (Query #3): Los nombres de aquellas estaciones de Montreal cuya distancia promedio de los viajes finalizados en ellas, es mayor a 6km.

También, se agrega el caso de uso C1 que corresponde a la ingesta de archivos estáticos y dinámicos.

2.2. Vista Lógica

Para la vista lógica se presenta un diagrama de clases que contiene la implementación de las principales clases que hacen a la comunicación interna del sistema (RabbitInterface) y a la comunicación con el cliente que representa cualquier conexión externa.

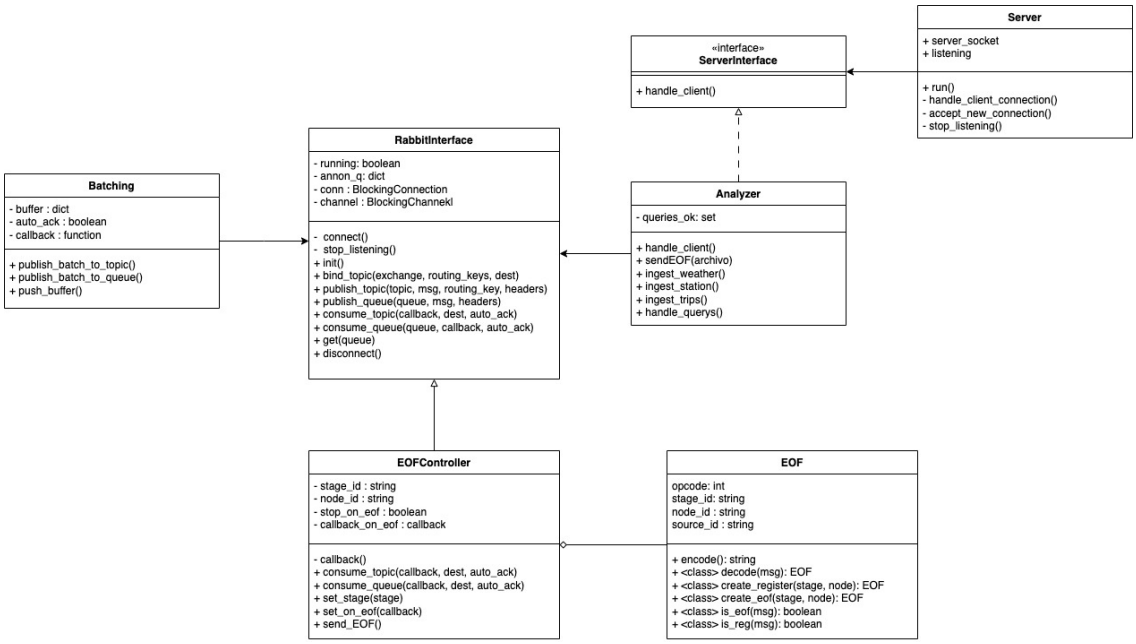

Figura 2: Diagrama de clases: Server - Rabbit Interface

2.2.1. RabbitInterface

Esta clase, junto con los módulos que la complementan, formalizan una serie de métodos que simplifican y regulan el accesso a los recursos de mensajería de RabbitMQ: Queues/Tópicos.

- **connect():** - Connecta con rabbit e inicializa las queues y tópicos.
- **bind():** - Crea una cola anónima identificada por la clave "dest" en la que se encolarán los mensajes publicados en el exchange indicado.
- **publish_topic()/publish_queue()** - Publica un mensaje en el tópico o en la queue indicada.
- **consume_topic()/consume_queue()** - Consume mensajes de un tópico o una queue, indicando el *callback* que procesará ese mensaje y el "dest" de donde consumir en caso del tópico.
- **get()** - Consume de manera no bloqueante los mensajes de una queue.
- **disconnect()** - Libera recursos y cierra la conexión con Rabbit.

Esta clase puede complementarse con EOFController y Batching para agregar nuevas características a la comunicación. La primera hereda de RabbitInterface por lo que tiene los mismos métodos, pero los redefine para sumarle una capa de validación de final de archivo. Es decir, si el mensaje entrante es un EOF (Data Class mostrada en la Figura 2), entonces no se llamará al callback asignado sino que se comunicará con el EOFManager para su posterior administración.

Además permite configurarle un callback específico para ejecutar solo cuando el mensaje entrante es un EOF.

El uso de EOFController implica que existirá comunicación con el EOFManager y requiere que se configuren los datos necesarios para identificar al proceso que envía/recibe el EOF. Estos son los siguientes: STAGE y NODE_ID.

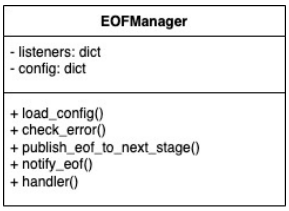

Figura 3: EOFManager

Por otro lado, la clase Batching utiliza un RabbitManager genérico para empaquetar mensajes en batches de tamaño parametrizable, y de este modo hacer un mejor aprovechamiento del overhead de la red transmitiendo un mayor volumen de datos. Esta clase tiene la responsabilidad de organizar los batches que serán enviados/recibidos a cada fuente/destino, controlar su tamaño y ejecutar los métodos **publish_topic()/publish_queue()** de la instancia de RabbitManager cuando corresponda hacerlo, para no afectar el normal funcionamiento del sistema.

2.2.2. Server Interface

La comunicación entre el cliente y el sistema se hará mediante una arquitecutra con el patrón Cliente-Servidor, donde el cliente le envía peticiones al sistema y éste responde en consecuencia enviándole los resultados.

Para implementar esto, se utiliza la clase Server que tiene la responsabilidad de abrir un socket, aceptar conexiones entrantes y delegar la "atención" a una implementación de la interfaz Server Interfaz. En este caso, una instancia de la clase Analyzer será quien procese los pedidos de los clientes y sus métodos responden fielmente a los distintos casos de uso que se pudieron observar en la Figura 1.

A continuación se muestran diagramas de estados de la clase Analyzer y EOFManager.

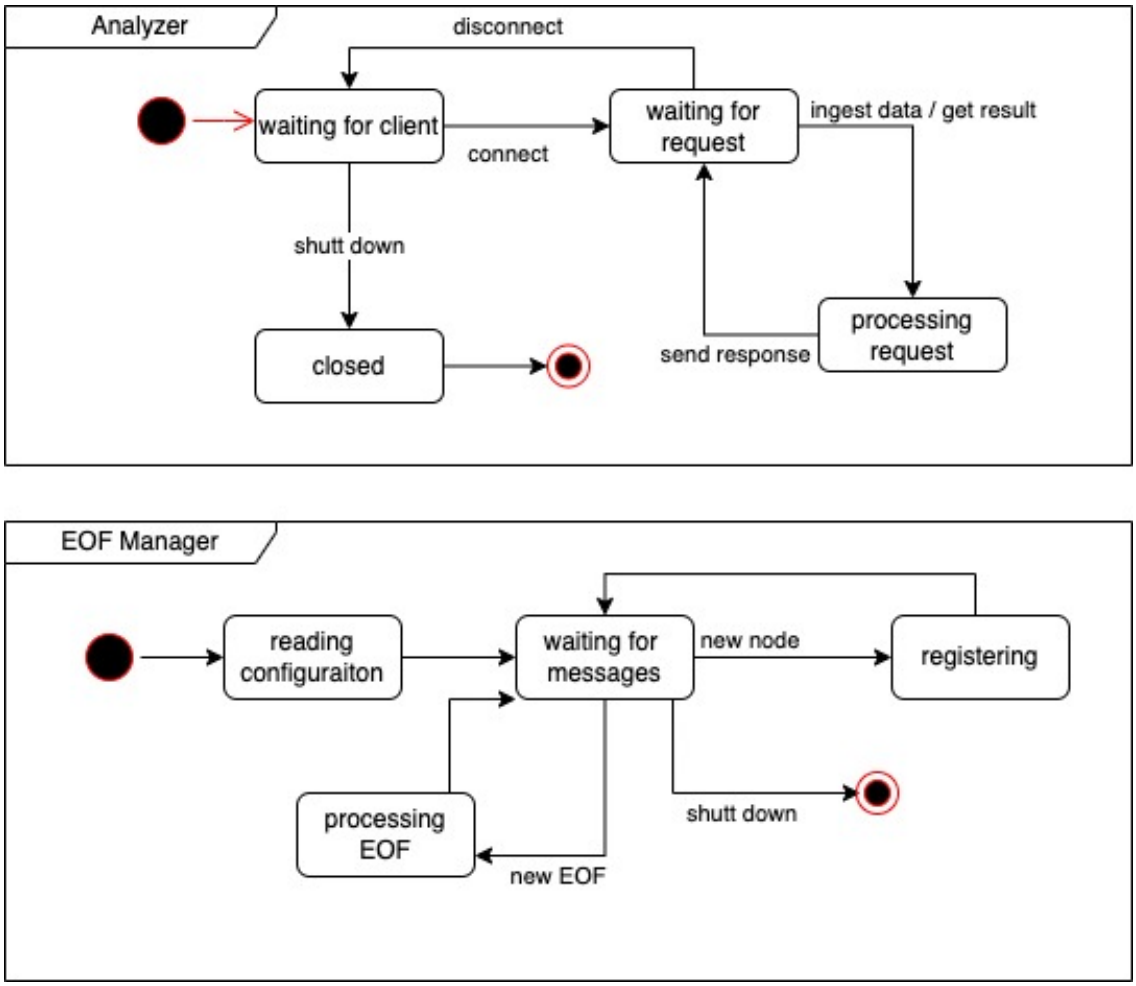

Figura 4: Diagrama de Estados: Analyzer y EOFManager

2.2.3. Nodos

Además de las clases que permiten la comunicación, llamaremos "Nodos" a piezas de código (normalmente funciones que actuan de handlers), que hacen un trabajo puntual con los datos de entrada. Los podemos clasificar en los siguientes tipos:

- Filter: aplican filtros verticales (eliminando campos de datos) y horizontales (eliminan registros).
- Joiner: realizan una union de dos tipos de registros en base a una key común. Pueden implementar filtros internamente.
- Worker: realizan una transformación o cálculo sobre una fuente de datos.

Aquí se muestran diagramas de estados que aplican en el sistema para dos de estos tipos de nodos:

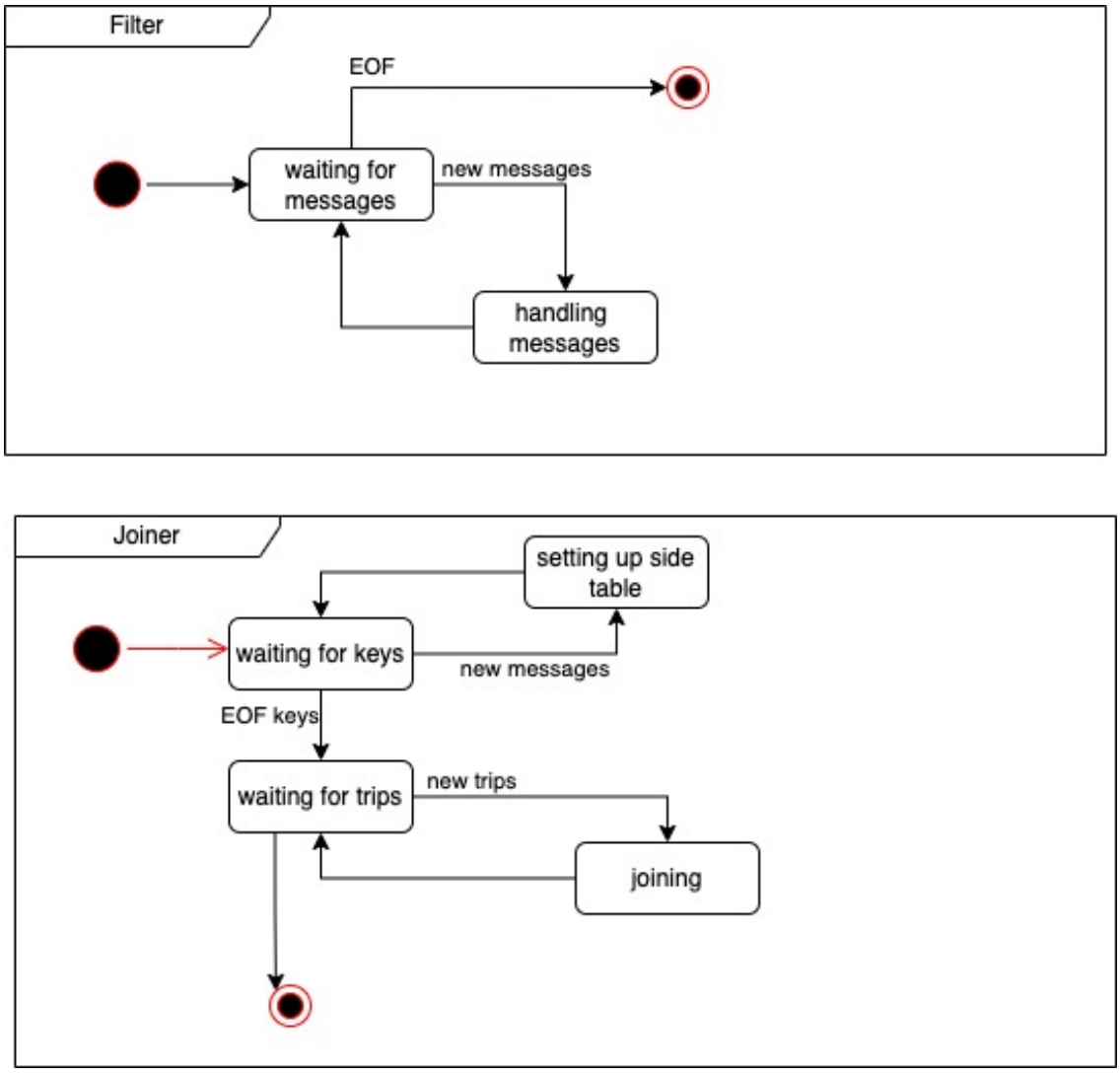

Figura 5: Diagrama de estados: Nodos filters y joiners

2.2.4. DAG

Estos nodos que realizan operaciones sobre los datos son encadenados para formar pipelines de sucesivas transformaciones que permiten responder las queries que se requieren. Estos pipelines generan una serie de dependencias que se pueden visualizar muy fácilmente mediante un DAG: Directed Asyclic Graph

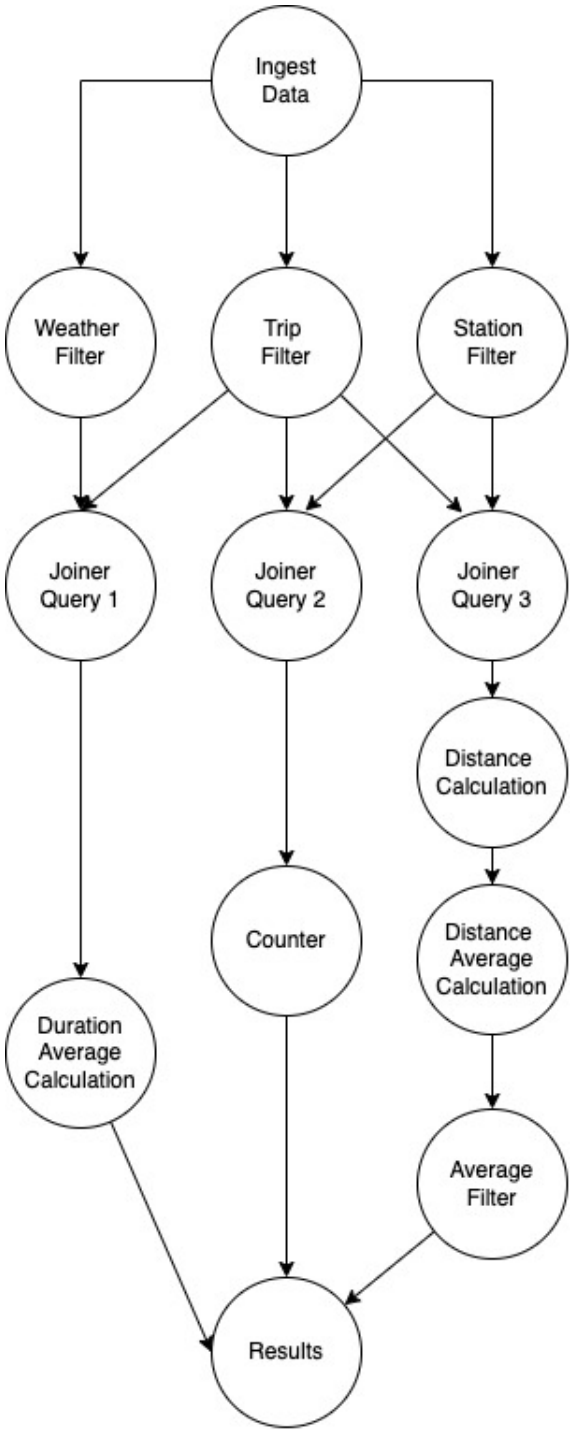

Figura 6: DAG - Filters

2.3. Vista de desarrollo

Para que el cada parte del sistema pueda realizar su función correctamente, existen otros módulos o paquetes que le proveerán funciones y/o soluciones a problemas específicos. Como ya se comentó, la instancia Analyzer debe poder comunicarse tanto con el cliente como con el resto del sistema, es por esto que hará uso de las fucniones que proveen RabbitInterface y Messaging

Protocol. En este último módulo se abstraen las capas mas bajas de la comunicación con el cliente utilizando sockets. Además se definen funciones que Encodean y Desencodean los datos que maneja el sistema para poder ser enviados por la red.

También el servidor debe conocer las estructuras de datos EOF y Result para poder hacer el control de fin de ingesta y devolver los resultados correspondientes.

El diagrama a continuación muestra las relaciones que existen entre los desplegables y sus dependencias.

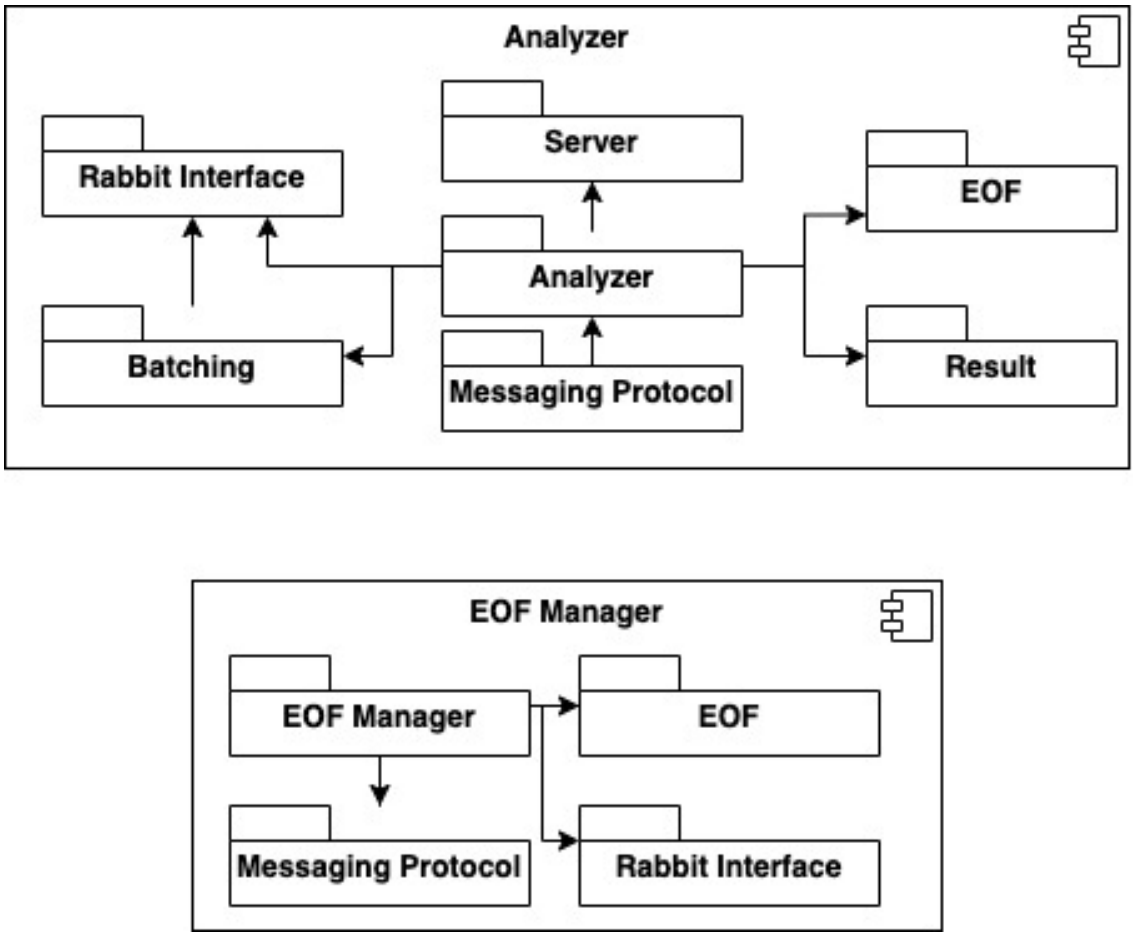

Figura 7: Diagrama de paquetes

2.4. Vista Física

Continuando con las capas del diseño del sistema, en la capa física podemos describirlo en función de los componentes físicos que tiene y sus enlaces de comunicación.

Para una primera propuesta, se provee una vista general de todos los nodos de sistema en el diagrama de robustez utilizando las siguiente clasificación de nodos o "actores" según su responsabilidad:

- Cliente: El cliente propiamente dicho.
- Analyzer: Interfaz expuesta al cliente para el uso del sistema.
- Rabbit MQ: Middleware de Mensajeria

- Filters: nodos encargados de recibir, filtrar y persistir los datos estáticos y dinámicos.
- Joiners: Nodos encargados de hacer la junta de datos en base a uno o varios campos (key).
- Counter: Contabiliza los trips de cada estación, en los años 2016 y 2017.
- Distance Calc: calcula la distancia entre dos ubicaciones, dadas sus coordenadas. Utiliza Haversine.
- Average Calc: calcula un promedio (distancias/duración).
- Average Filter: Filtra por un umbral los promedios calculados.

En la siguiente imagen se puede ver la interacción entre el cliente y servidor según cada caso de uso, y el servidor consultando la queue que almacena resultados para devolverle al cliente.

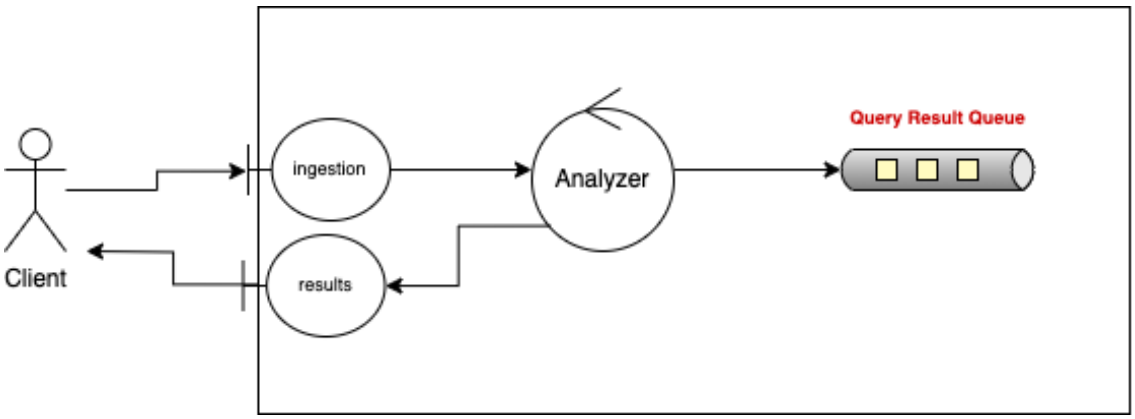

Figura 8: Diagrama de Robustez del sistema: Cliente - Servidor - Results Queue

El siguiente diagrama de robustez, enmarca una propuesta inicial para la ingesta de los tres tipos de archivos y sus respectivos pipelines.

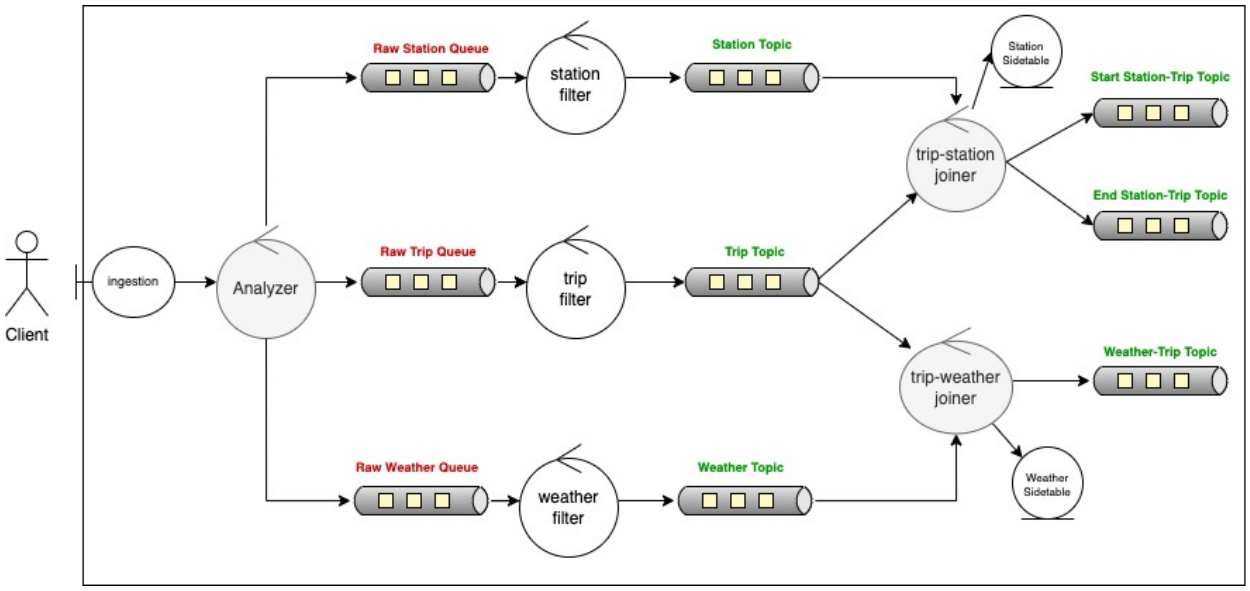

Figura 9: Diagrama de Robustez del sistema: Pipeline de Querys

Luego de ser publicados en los distintos tópicos, solo es cuestión de consumir de ellos para responder las consultas que se deseen sobre los datos.

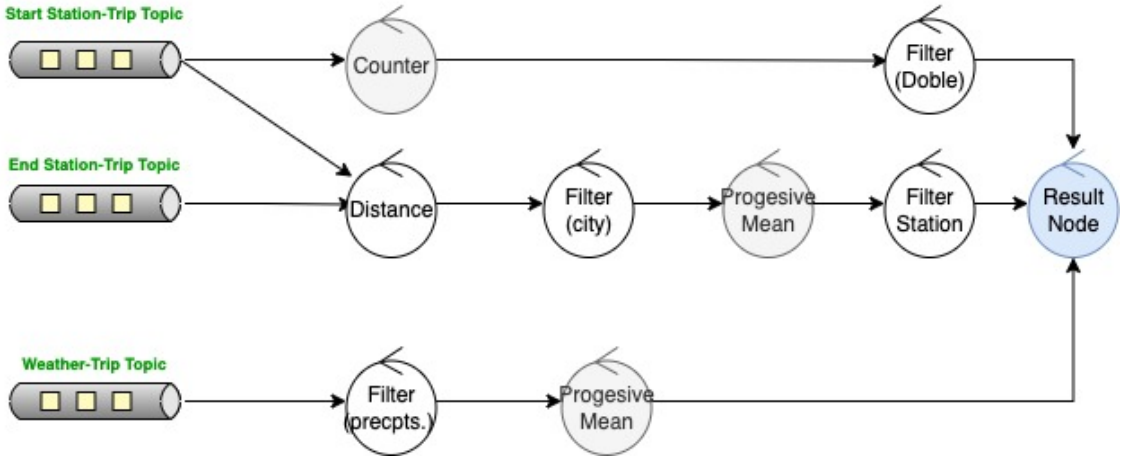

Figura 10: Diagrama de Robustez del sistema: Pipeline de Querys

La principal ventaja de este diseño es que responde al principio de extensión sin necesidad de mayores cambios. Es decir, si se quiere realizar otra consulta estadística sobre los Trips, Stations o Weathers, solo es cuestión de consumir mensajes del tópico deseado y desarrollar el pipeline para la nueva query. En la práctica, este diseño tuvo varios problemas principalmente de performance ya que frente a grandes velocidades en la ingesta de archivos, los joiners comienzan a multiplicar esa información en memoria, generando problemas de escalabilidad en entornos locales.

Se propuso un segundo diseño, más acoplado al problema pero a cambio de grandes mejoras

en la pérformance:

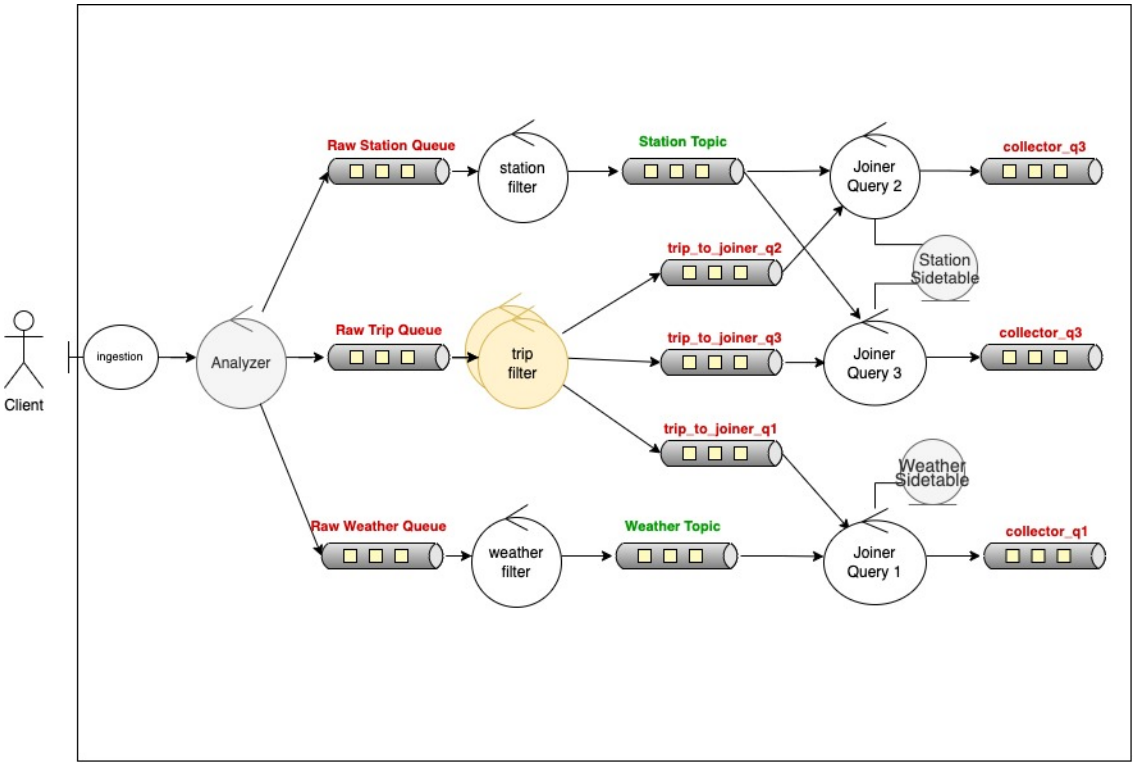

Figura 11: Diagrama de Robustez del sistema: Pipeline de Ingesta de archivos

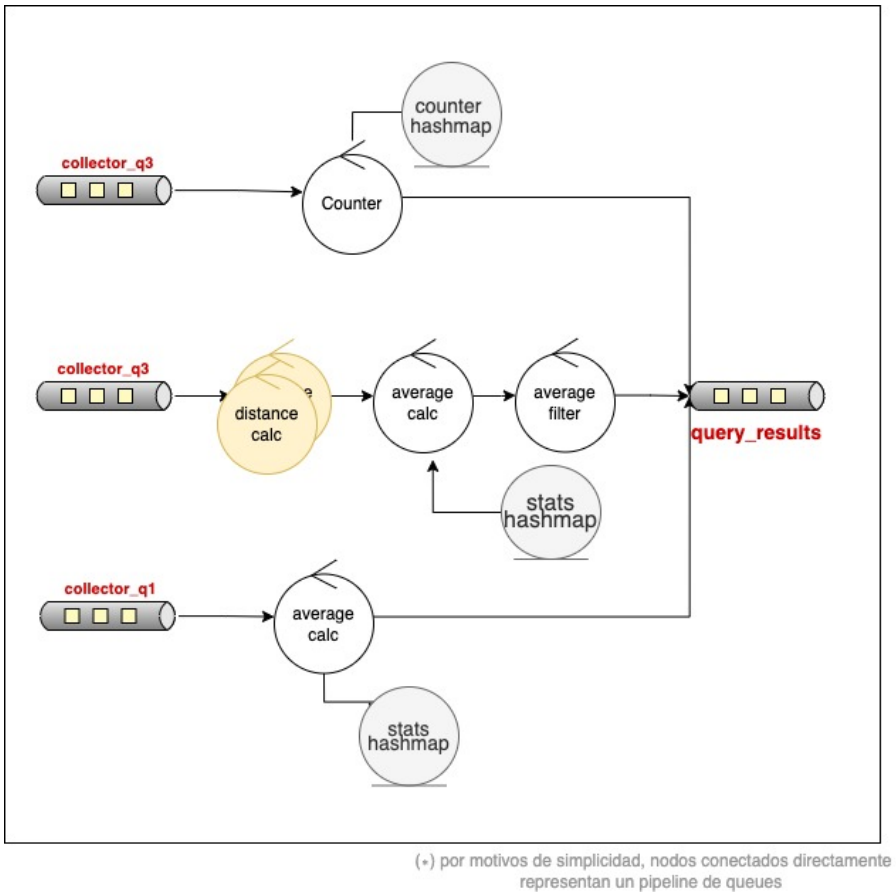

Figura 12: Diagrama de Robustez del sistema: Pipeline de Querys

Esta nueva arquitectura elimina la mayoría de los tópicos y agrega un joiner para cada query. De esta manera se asegura que los datos a joinear son los mínimos y necesarios. Además, varios criterios de filtrado que eran aplicados por los nodos Filters posterior a los Joiners, ahora son implementados en los nodos Trip/Weather/Station Filter, lo cual permite que el volumen de datos se mantenga controlado.

Se muestra una porción del diagrama de robustez que aplica a todos los nodos que sean replicados y muestra la interacción con el EOFManager a la hora de recibir un EOF.

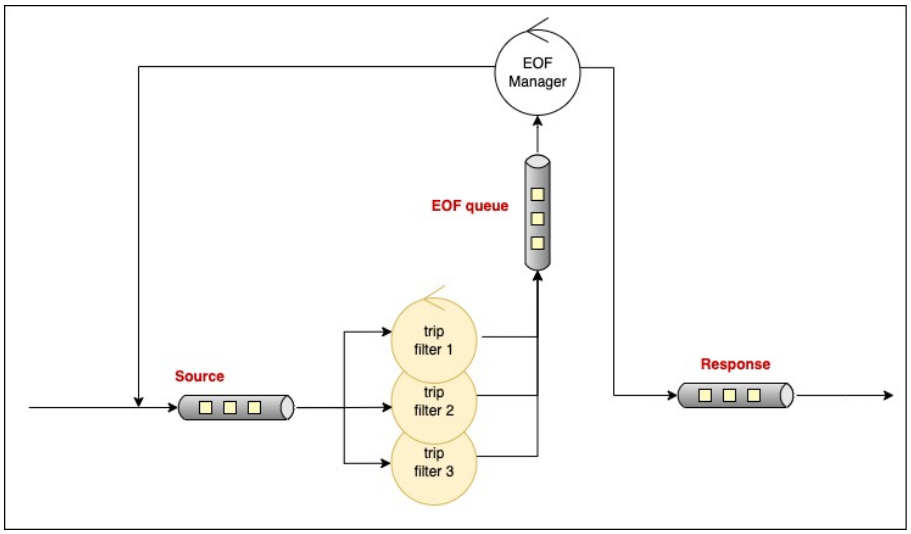

Figura 13: Diagrama de Robustez: Replicas y EOFManager

A continuación se muestra un diagrama de despliegue de esta solución:

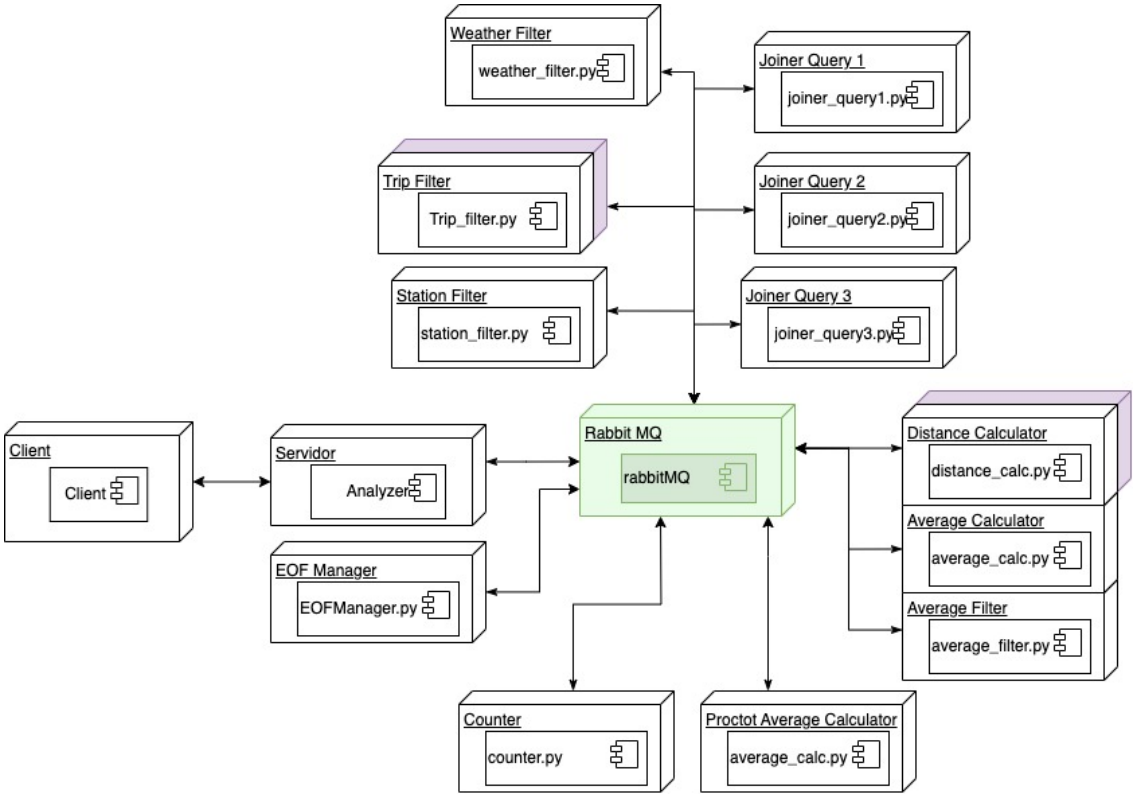

Figura 14: Diagrama de Despliegue

2.5. Vista de Procesos

Para la vista de procesos se muestra, por un lado, la dinámica de consulta entre el Cliente, Servidor y un Data Filter Node que representa a alguno de los 3 filtros de los datos crudos. Por otro lado, se presenta otro diagrama sobre el proceso de la ingesta de archivos dentro del sistema en los Station/Trip Filters y el nodo Joiner para la query 3.

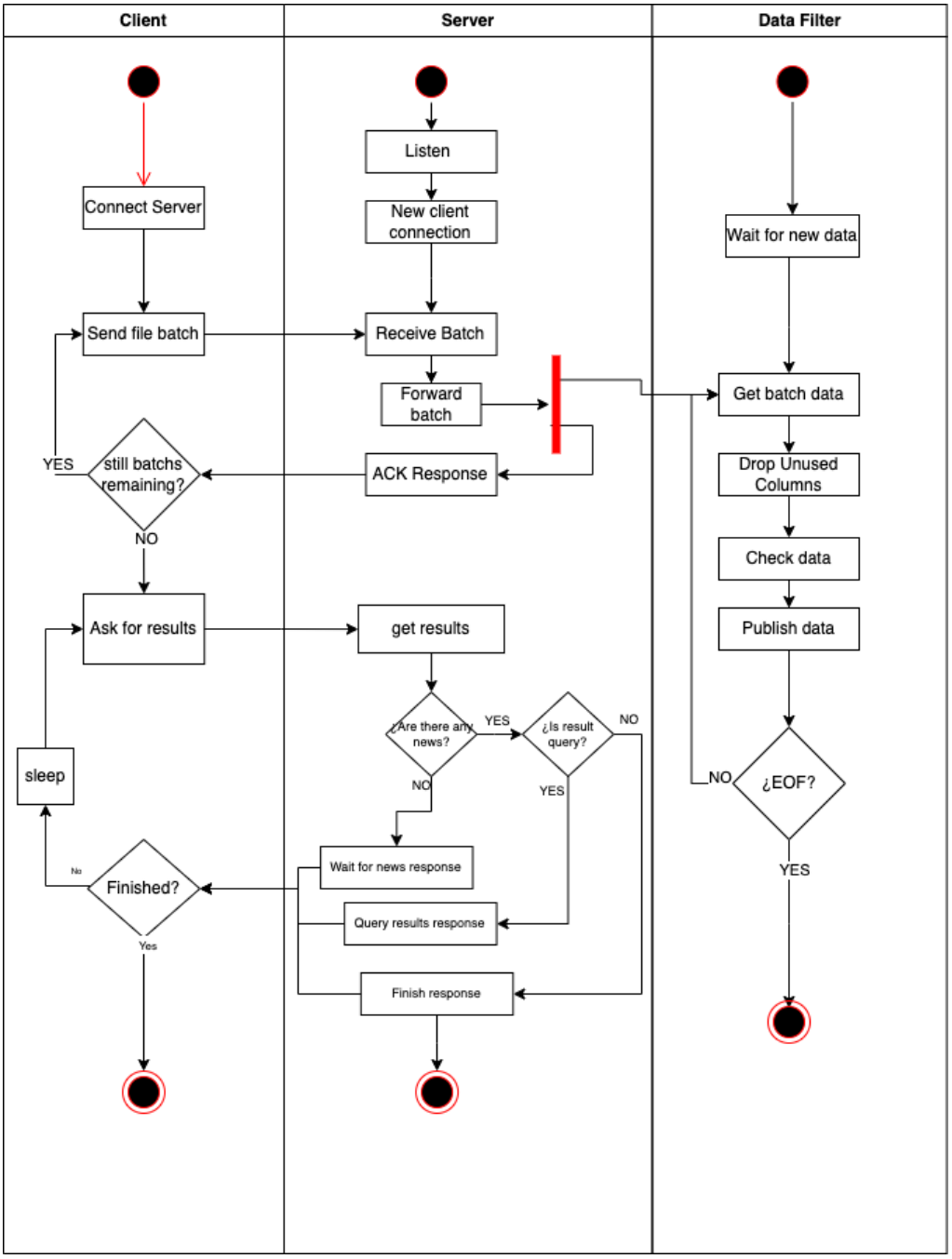

Figura 15: Diagrama de Actividades: Cliente - Servidor - Data Filter

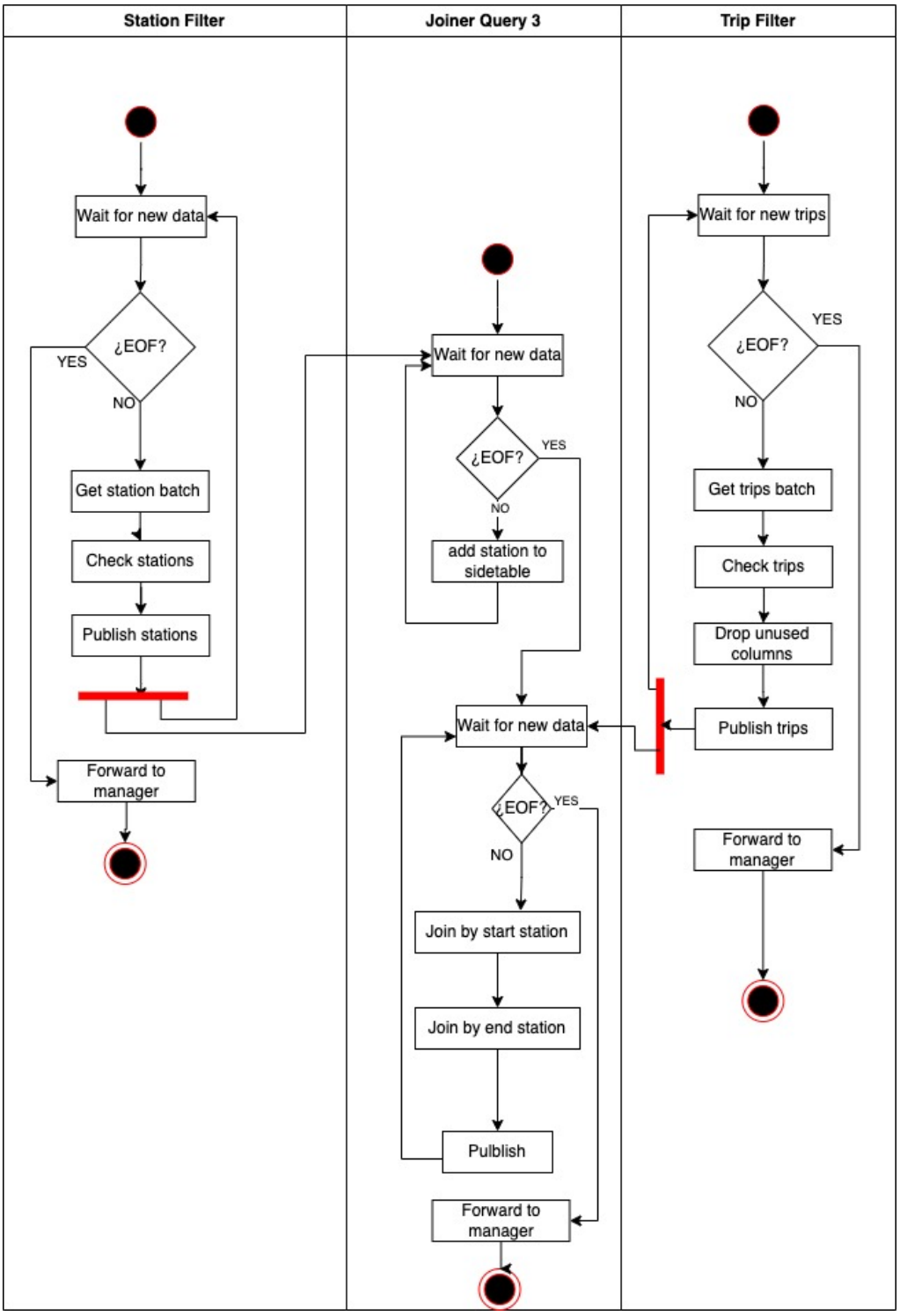

Figura 16: Diagrama de Actividades: Filters y Joiner

En el siguiente diagrama de procesos se muesta la evolución en el tiempo de los filtros notificando al EOFManager y cómo este espera por todos los EOF de una misma etapa antes de enviárselo a la siguiente etapa.

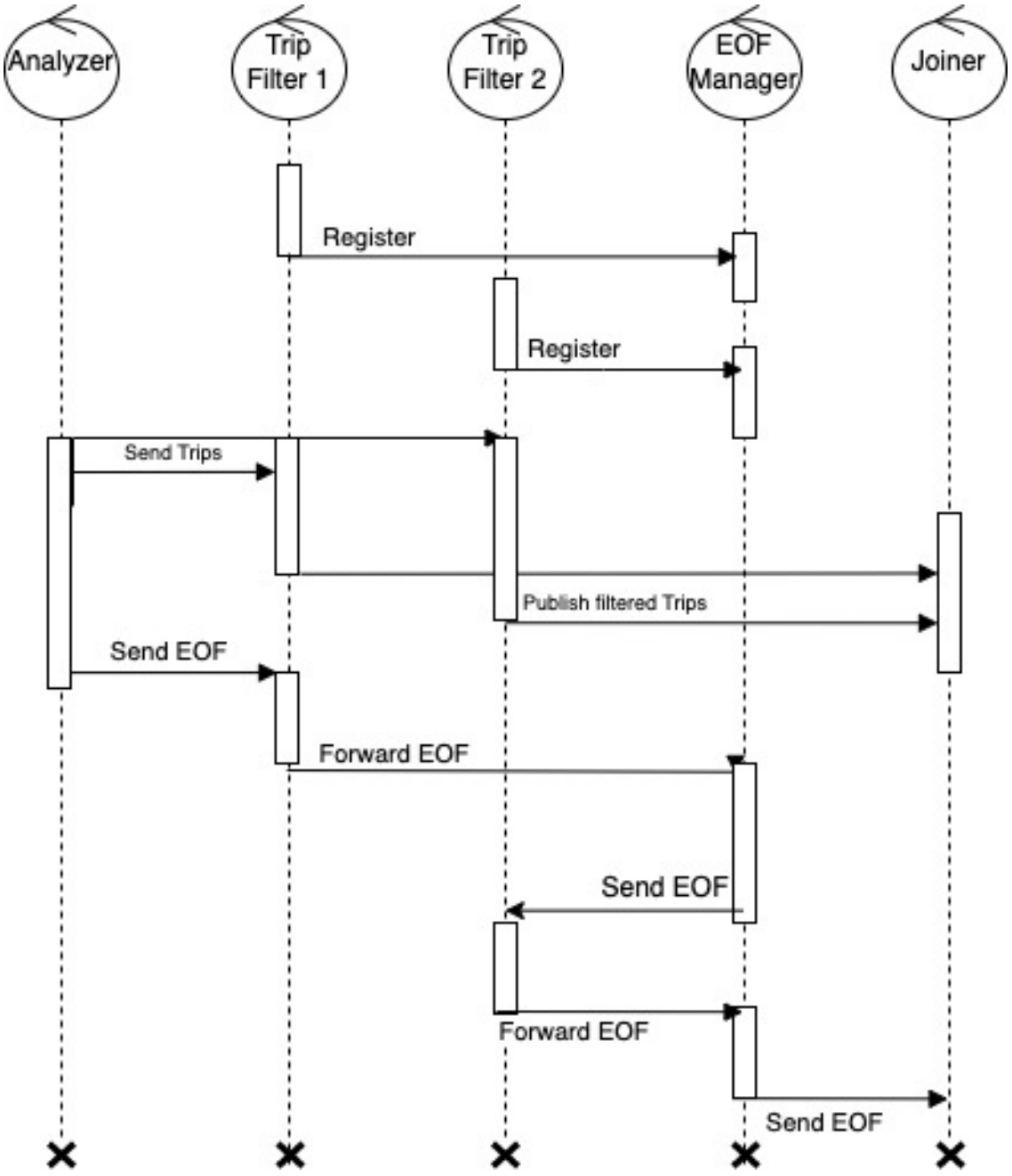

Figura 17: Diagrama de Secuencia