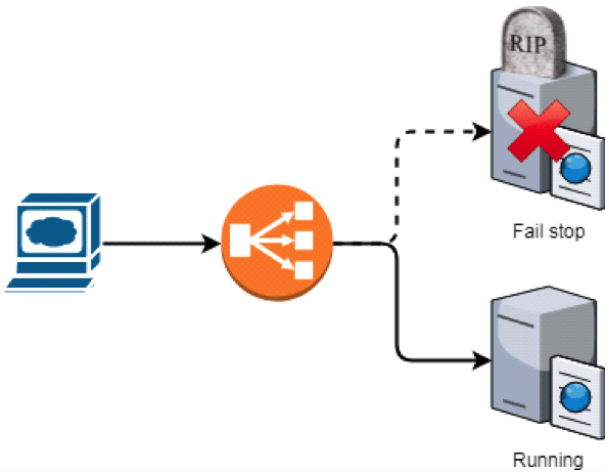
.UBAfiuba
FACULTAD DE INGENIERÍA

Sistemas Distribuidos 1: TP N° 2



1° Cuatrimestre, 2023

Apellido y Nombre	Email
Sabatino, Gonzalo	gsabatino@fi.uba.ar
Capón Blanquer, Mateo	mcapon@fi.uba.ar
Sportelli Castro, Luciano	lsportelli@fi.uba.ar

Índice

1. Scope del Trabajo **2**

2. Flujo de datos **2**

3. Escenarios: Casos de uso **3**

4. Desarrollo de diagramas: Vistas **4**
4.1. Vista lógica 4
4.2. Vista física 8
4.3. Vista de Procesos 14
4.4. Vista de Desarrollo 22

5. Issues pendientes **24**

1. Scope del Trabajo

Se solicita un sistema distribuido que analice los registros de viajes realizados con bicicletas de la red pública provista por grandes ciudades. Los clientes deben enviar los registros que caracterizan a las estaciones de bicicleta, la cantidad de precipitaciones, y a los viajes realizados en la ciudad que desean analizar.

Se debe obtener:

- 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 las estaciones de Montreal para las que el promedio de los ciclistas recorren más de 6km en llegar a ellas.
- El promedio de la duración de todos los viajes.

2. Flujo de datos

En la Figura 1 se muestra el DAG propuesto para un cliente dentro del sistema, mostrando cómo es el flujo de los tres distintos tipos de datos (station, weather, trip) desde que arriban hasta que, según sean o no filtrados previamente, llegan al resultado de cada query.

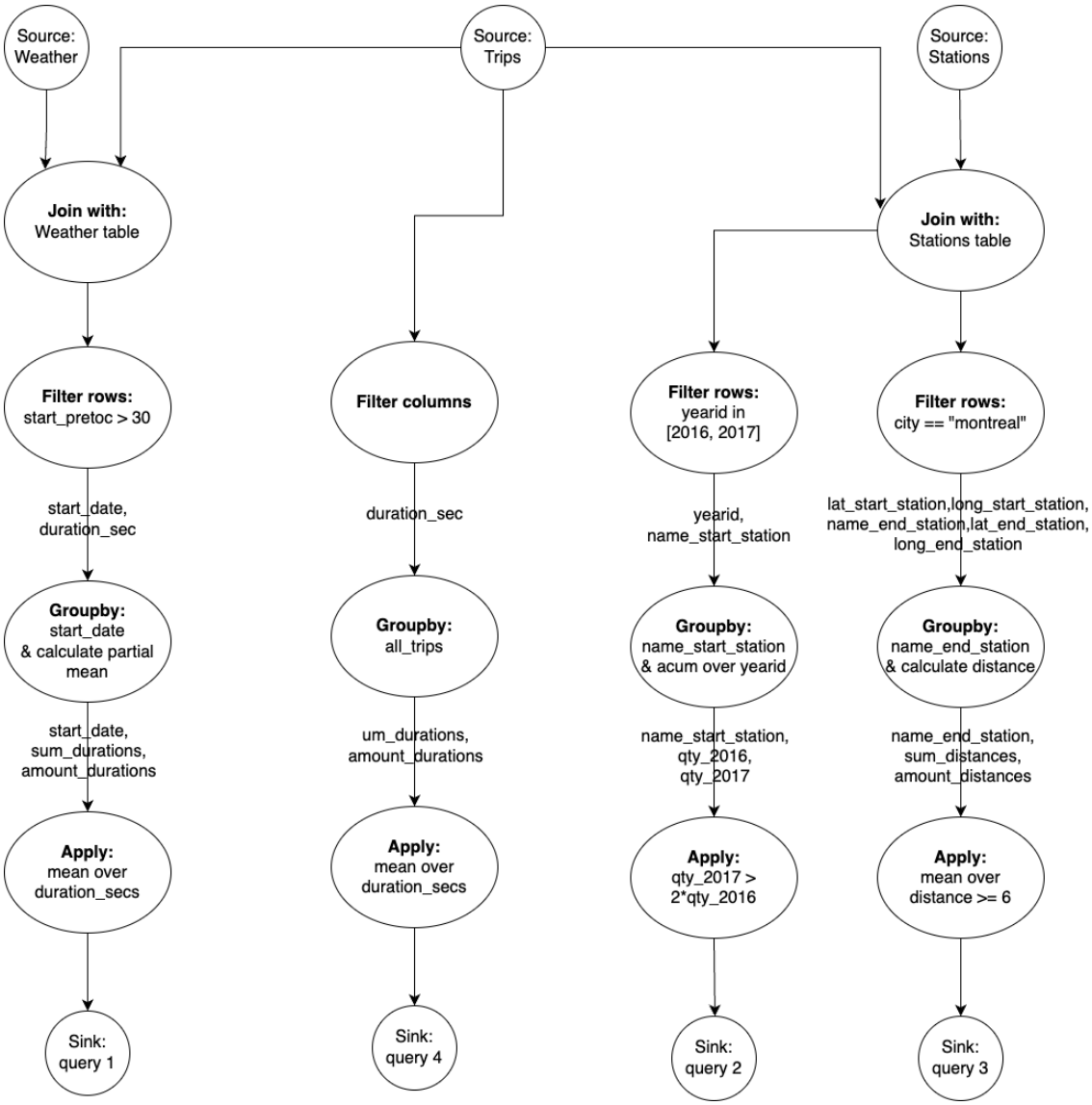


Figura 1: DAG mostrando el flujo de datos

A partir de entender las tres fuentes de datos, se propone el siguiente esquema de pasaje de datos:

1. Envío de datos estáticos (stations y weather) para las tres ciudades, los cuales serán almacenados en memoria en el nodo apropiado.
2. Envío de los trips para las tres ciudades.
3. Consulta por parte del cliente por la obtención de los resultados.

3. Escenarios: Casos de uso

En la presente sección se muestran los casos de uso del sistema. Como se aprecia en la Figura 2, los casos de uso del cliente son: postear datos estáticos, postear datos de viajes y pedirle a servidor el resultados de las tres queries.

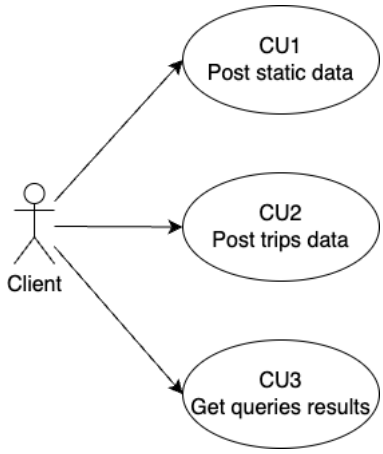


Figura 2: Casos de uso del cliente

Se aprecia entonces que el sistema cuenta de dos etapas: un procesamiento de la información que le brindó el cliente, y la entrega final de resultados una vez que se procesaron todos los datos.

A continuación, se desarrolla el resto de las vistas que especifican distintos aspectos de la arquitectura diseñada.

4. Desarrollo de diagramas: Vistas

4.1. Vista lógica

Supervisores de nodos: Docker Restarter

Dado que el sistema requiere tener alta disponibilidad, y ser tolerante a fallos, se decidió implementar un nodo que supervisa activamente el estado de los demás procesos.

De este modo, cuando se identifique que un nodo está inactivo, se lo volverá a iniciar inmediatamente. Desde el punto de vista del nodo supervisor, los contenedores pueden tener tres estados posibles: activos (en healthcheck), inactivos (deben ser reiniciados) o en un estado intermedio luego de ser reiniciados (esperando a tener una conexión). Las transiciones entre estos estados los mostramos en el siguiente diagrama.

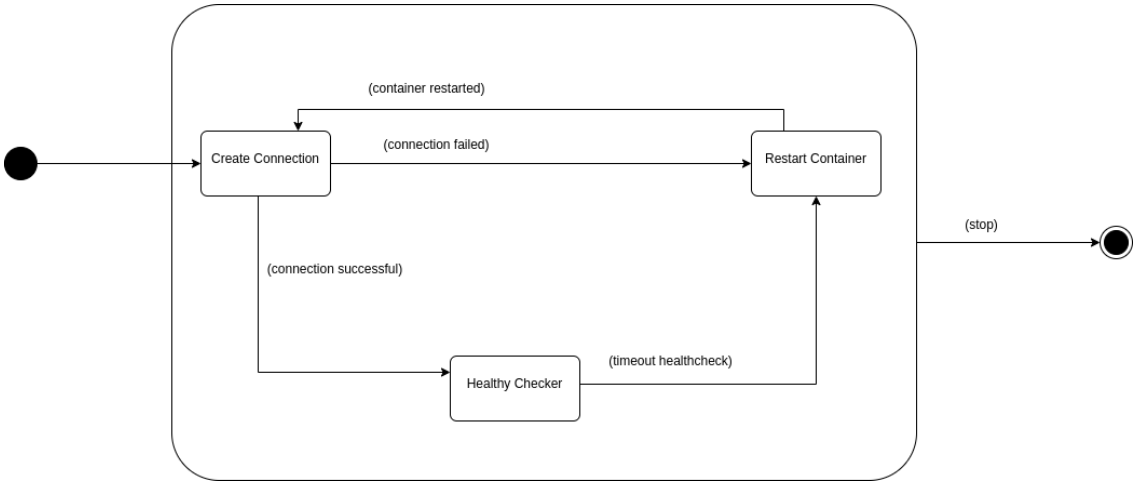


Figura 3: Diagrama de Estados - Estado de los Procesos

Elección de Líder

Siendo que el proceso que verifica el estado de los nodos también puede fallar, se decidió replicarlo. Solo una de las tantas réplicas debe reiniciar al nodo caído. Esto es así, para evitar retrabajo y para evitar una sobrecarga innecesaria en la red. En consecuencia, uno de estos nodos toma un rol activo, y los demás, uno pasivo. En otras palabras, se necesita un líder que se encargue del trabajo, mientras que los demás nodos deben estar preparados para tomar las tareas del líder, si este se cae.

Decidimos implementar una elección de líder basada en el algoritmo Bully. Mostramos a continuación un diagrama de estados de la elección de líder.

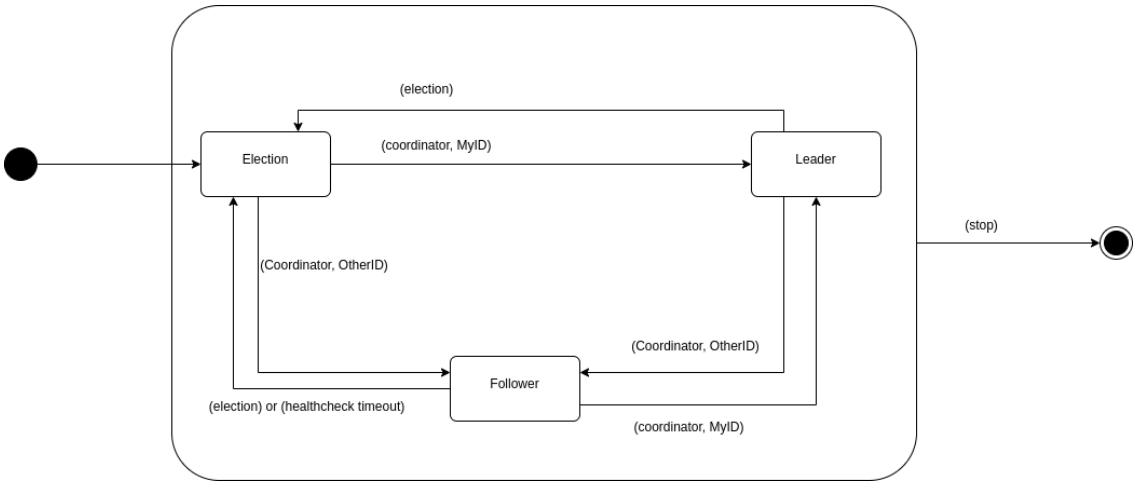


Figura 4: Diagrama de Estados - Estado de la Elección de Lider

En el diagrama hay dos transiciones que son particularmente llamativas: las que pasan del estado Follower al estado Leader, y viceversa. En un caso ideal, en el que los paquetes UDP no se pierden, estas transiciones no ocurren nunca. Sin embargo, en un caso real, las mismas ocurren cuando muchos paquetes particulares del proceso de elección se pierden.

Patrón Routing y Workers para envío de mensajes

Cada proceso que recibe mensajes de otro, tiene su propia cola. Hay distintas causas por las que se decidió implementar una cola de recepción de mensajes por procesos (en vez de compartir una cola entre distintos workers). Por un lado, si comparten la cola, habrá un proceso centralizado en Rabbit que routee el mensaje a cada consumidor que necesite un mensaje. Los mensajes a consumir se deben ordenar en esa cola compartida. En cambio, al tener una cola por consumidor, no es necesario este routeo, y habrá un procesamiento sin un cuello de botella en esa cola.

Por otro lado, el EOF Manager, producirá un mensaje EOF para cada cola. Si los procesos comparten una cola, esto implicaría que el EOF recibido se debería reencolar en caso de no ser el que le corresponde a ese proceso. En el caso de este sistema, no se necesita reencolar.

A continuación se diagrama esta situación para el ejemplo particular de un *Joiner* y los *Filters*.

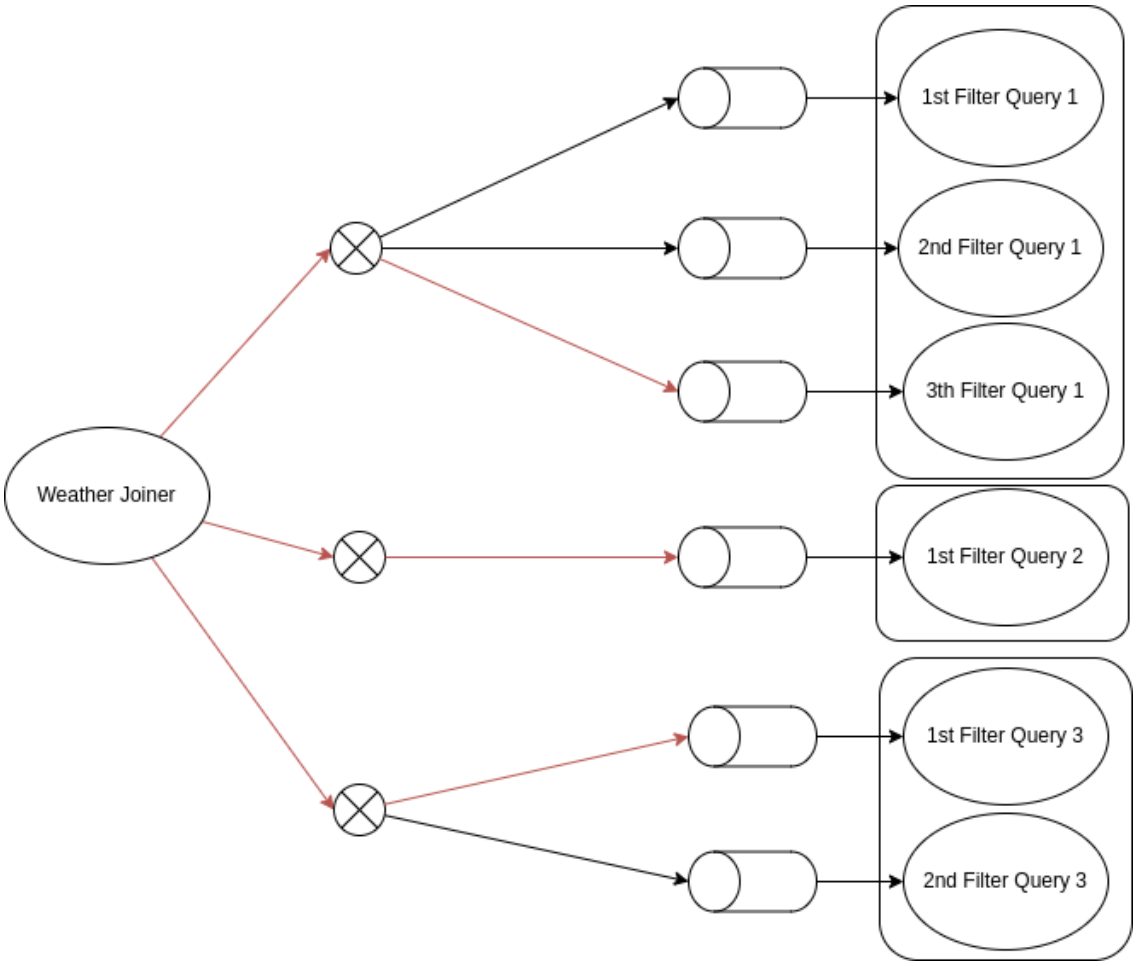


Figura 5: Patron Routing y Workers para envío de mensajes a los filtros

Para un viaje en particular, el mismo debe ser procesado para todas las queries, pero solo una vez por query. Es por esto que se elige una sola cola entre las disponibles para cada query. El envío se hace con Round Robin.

Procesos Genéricos

Desde un punto de vista más cercano a la implementación, se aprecia un diagrama de clases genérico de los nodos que utilizan el middleware implementado.

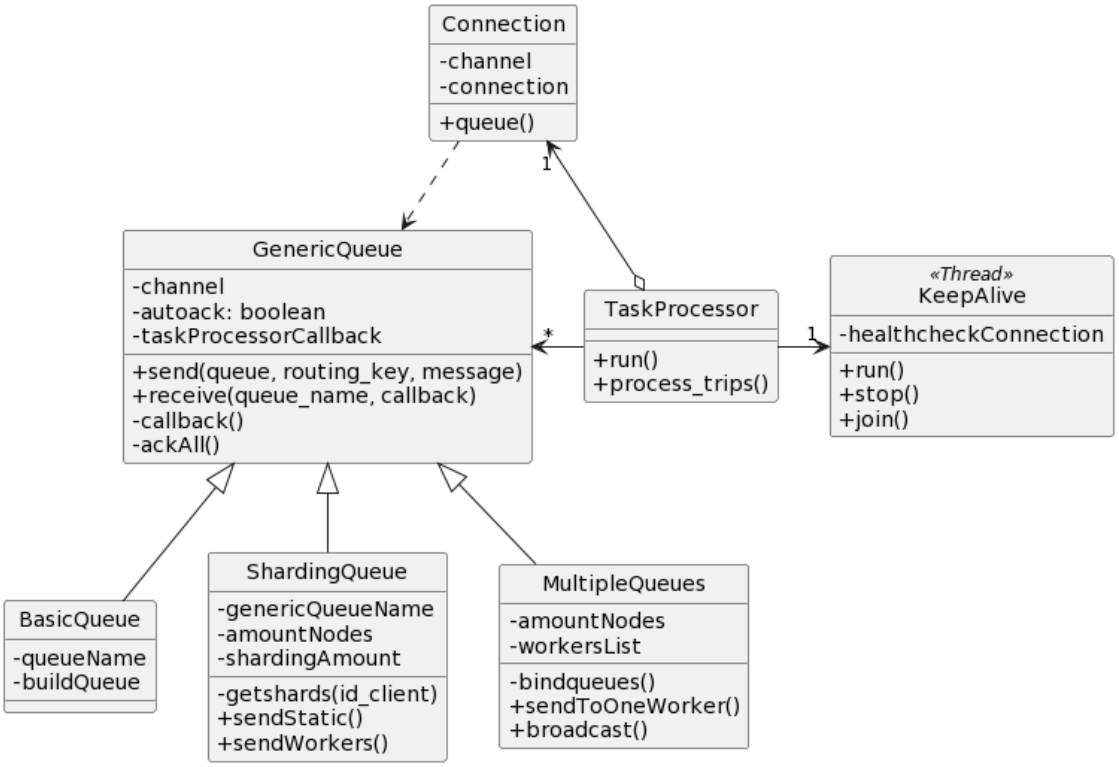


Figura 6: Diagrama de Clases Genérico

Al abstraer el middleware en las clases *Connection* y *GenericQueue*, no hay dependencias fuertes con Rabbit. Se podría modificar fácilmente la comunicación con otra librería.

Se utiliza un mecanismo de herencia para implementar las distintas colas que son de utilidad. Utilizando el ejemplo dado en la figura previa, una instancia de *WeatherJoiner* envía a los *Filter* a través de una *MultipleQueues* invocando el método *sendToOneWorker()*. Por su lado, el Weather Joiner, recibe datos de una cola del tipo *ShardingQueue*. Esto es así para evitar replicar en todos los nodos los datos estáticos.

Por su lado, todos los procesos cuentan con un hilo *KeepAlive*, el cual se encarga de notificar al supervisor de los nodos que está activo.

Estado de la sesión

A continuación mostramos un diagrama de estados del inicio de la sesión del cliente en el sistema.

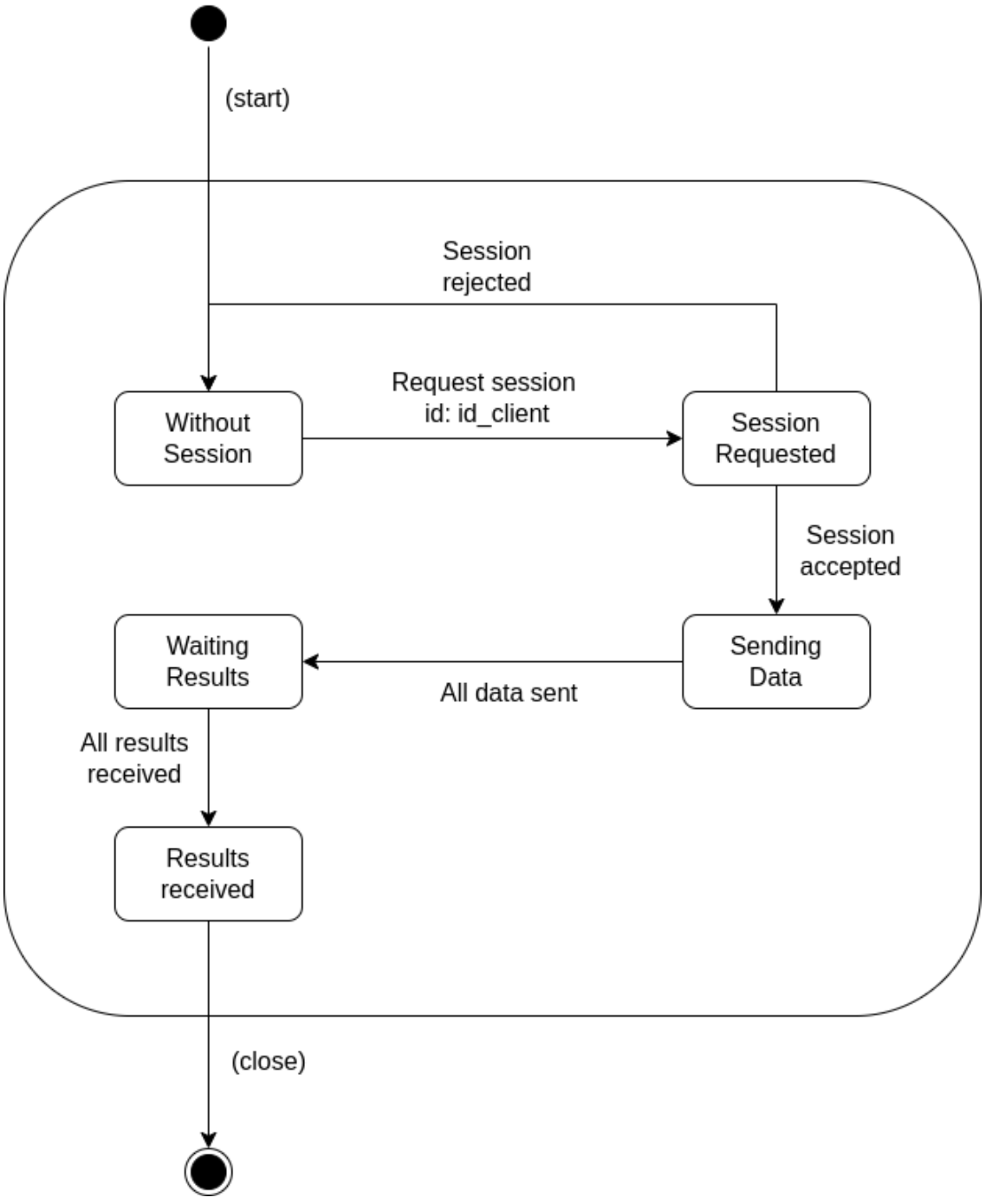


Figura 7: Estado de la sesión de un cliente

4.2. Vista física

En esta sección se desarrollan los diagramas relativos a los componentes físicos del sistema, entendiendo cómo se despliegan y cómo se relacionan los distintos componentes entre sí.

Diagramas de Robustez

En la Figura 8 se muestra un diagrama genérico de la arquitectura. En él mostramos cómo el cliente envía datos una vez aceptada la sesión.

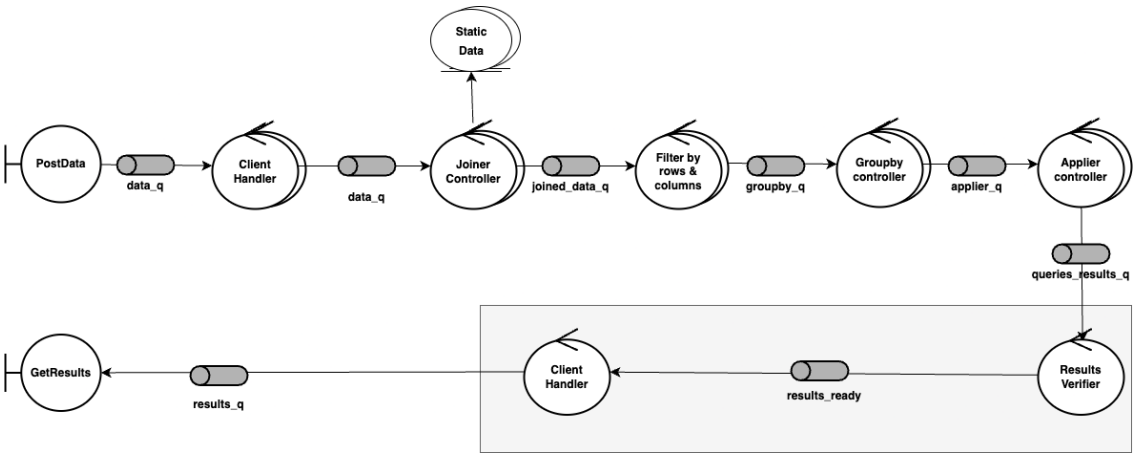


Figura 8: Diagrama de robustez - Arquitectura General

En la siguiente figura, resumimos brevemente cómo el cliente inicia sesión (mostrando los hilos que se crean en un mismo nodo).

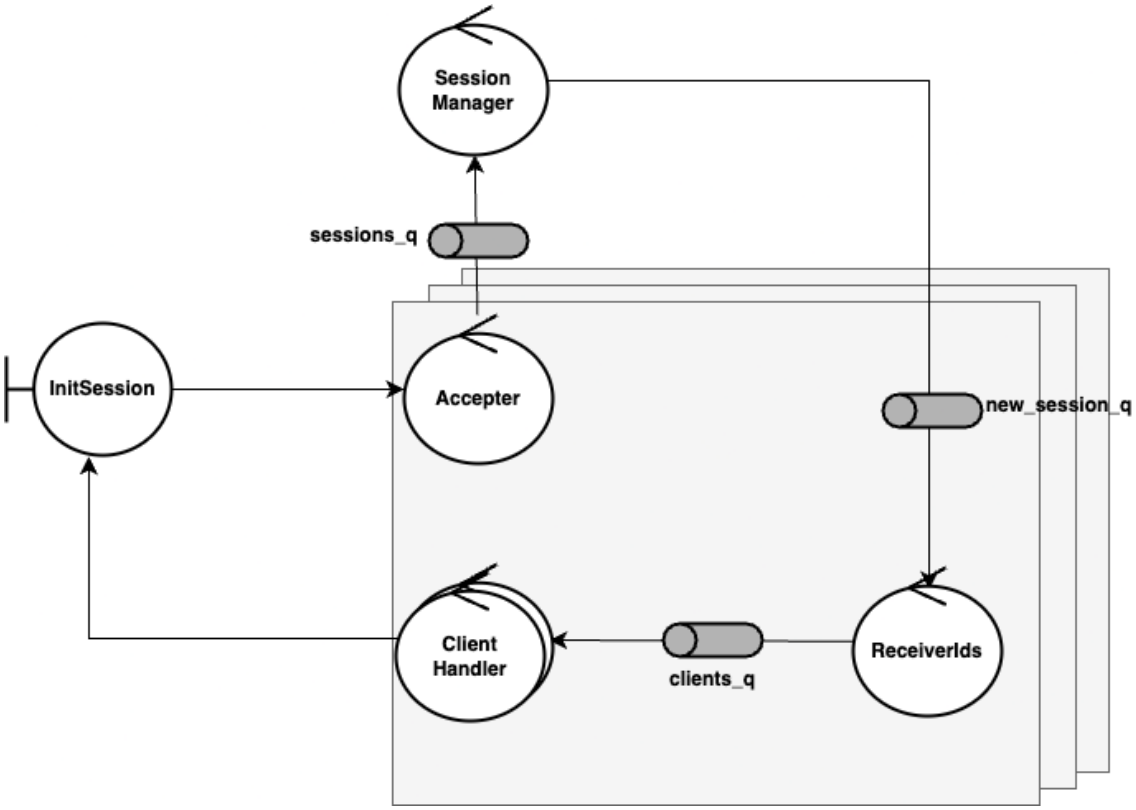


Figura 9: Diagrama de robustez - Inicio de sesión

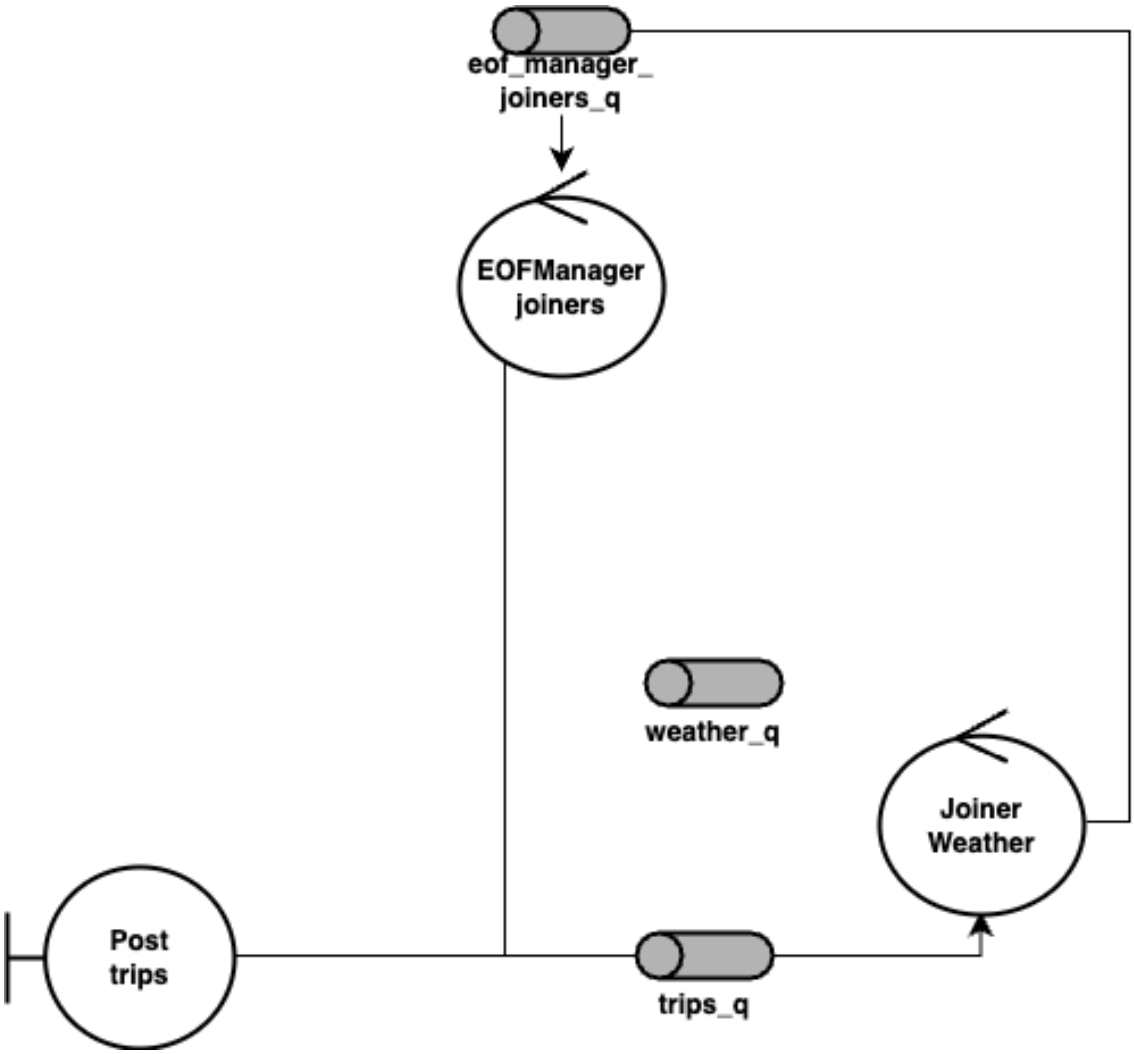


Figura 10: Diagrama de robustez - eof manager (1)

En las siguientes figuras (10, 11, 12 y 13) se desarrolla en varias partes cómo se despliegan el middleware distribuido para manejar los últimos paquetes de cada tipo, llamado *EOFManager*. El mensaje de EOF debe ser pusheado a la misma cola por la que se reciben los mensajes para evitar condiciones de carrera.

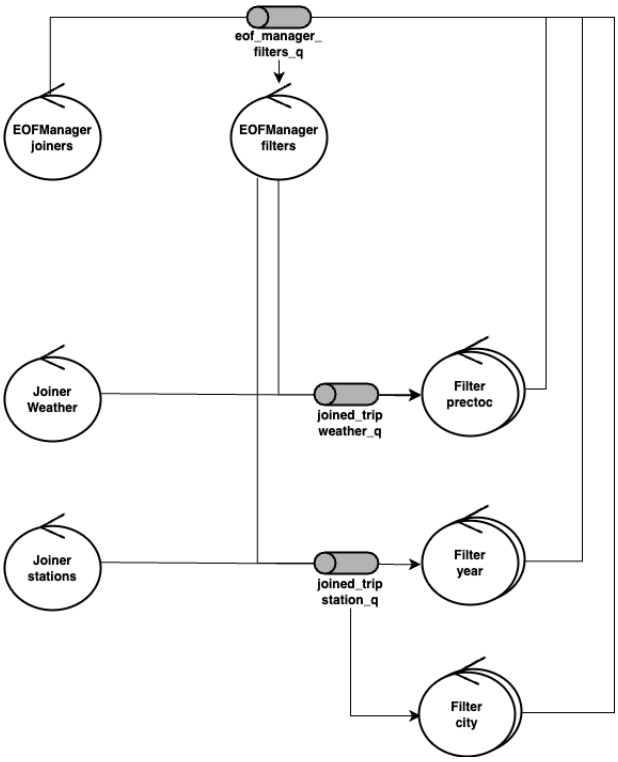


Figura 11: Diagrama de robustez - eof manager (2)

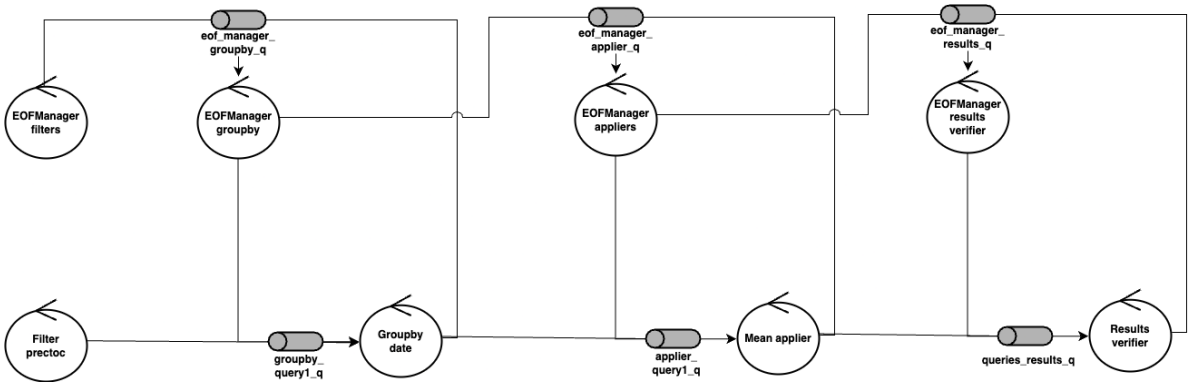


Figura 12: Diagrama de robustez - eof manager (3)

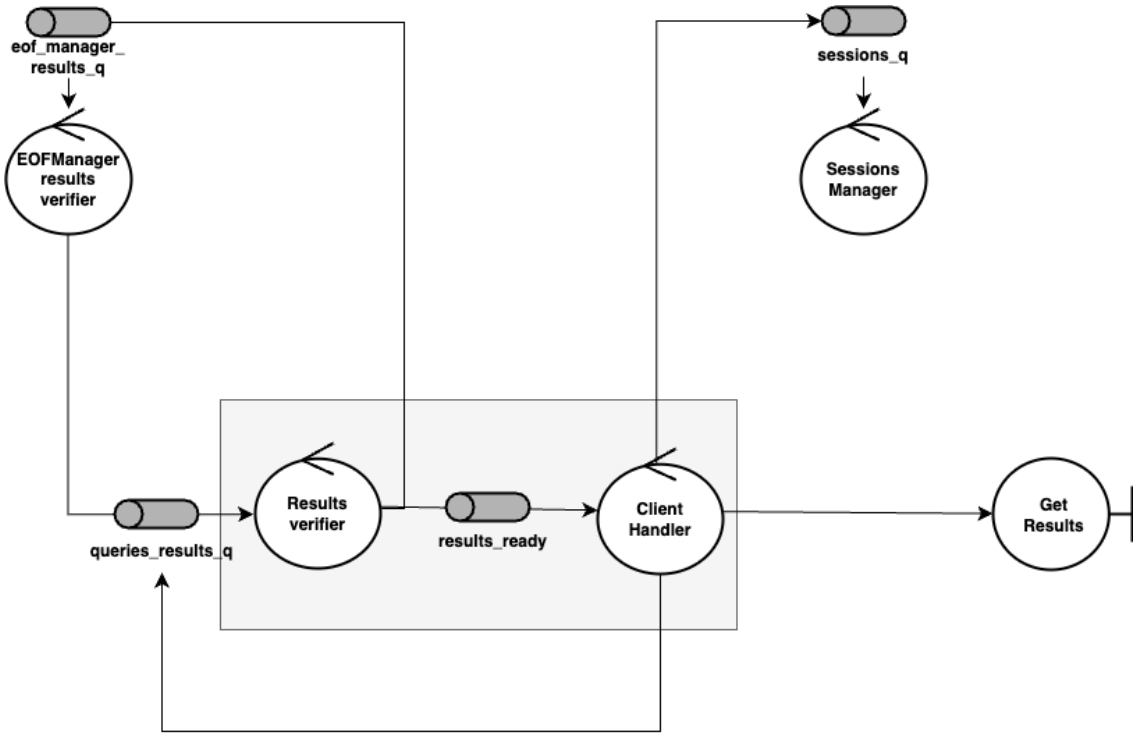


Figura 13: Diagrama de robustez - eof manager (4)

En la Figura 13 se aprecia que, luego de obtener el EOF de los datos procesados, el nodo comienza a enviarle los resultados procesados al cliente.

Elección de Lider

En la elección de líder participan los controladores mostrados en la figura 6. Todos deben existir en la misma computadora que ejecuta la elección. En nuestra implementación, decidimos lanzar hilos de Python, considerando que los hilos estarán casi siempre bloqueados esperando por eventos.

Al modelar estos controladores, nos permitimos separar claramente las etapas y los roles de cada uno de ellos en la eleción. Por un lado, el hilo sender y el hilo receiver, se encargan de enviar y recibir datos a través de un socket UDP. Se decidió implementar la comunicación con Sockets UDP, evitando así mantener conexiones activas.

Al tener un transporte no confiable, puede suceder que, si se pierden determinados paquetes (no se recibe ninguno de los mensajes ELECTION, ni el mensaje COORDINATOR), pueden coexistir por unos segundos dos líderes. Este caso es muy particular y sumamente improbable, porque implicaría que el líder pierde conexión con todos los seguidores, pero puede seguir trabajando como líder (y reiniciando contenedores). Al pasar unos pocos segundos, se conectará un líder con el otro (para hacer el healthcheck a nivel de reiniciado de contenedores), generando una inconsistencia, la cual es manejada en el código.

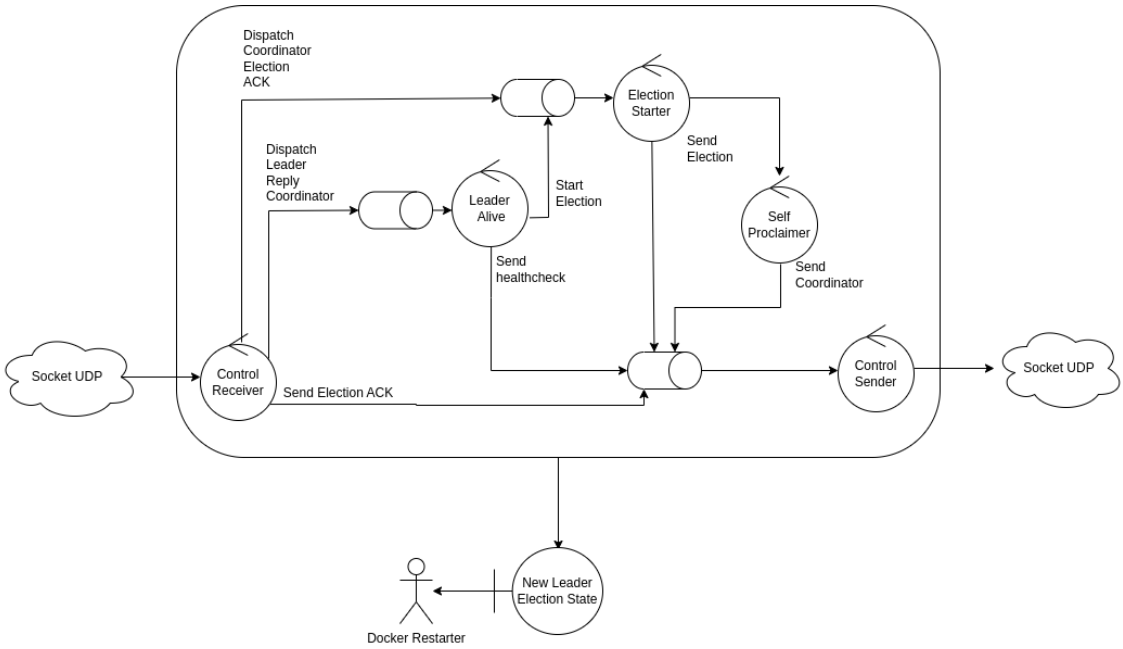


Figura 14: Diagrama de robustez - Elección de Lider

Diagramas de Despliegue

Una vez entendido la arquitectura general, examinando también ciertos casos particulares, se procede a describir cómo se despliegan los distintos componentes. En este caso, el componente que no aparece en el resto de los diagramas es el broker de rabbit, que tiene interacción con todos los componentes del servidor.

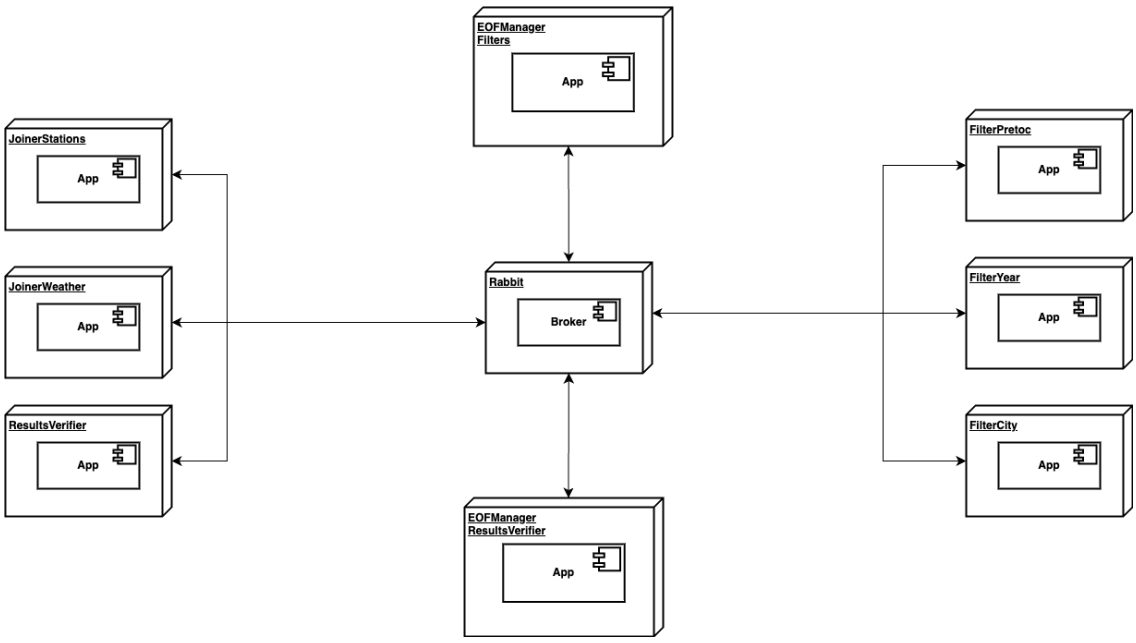


Figura 15: Diagrama de Despliegue (1)

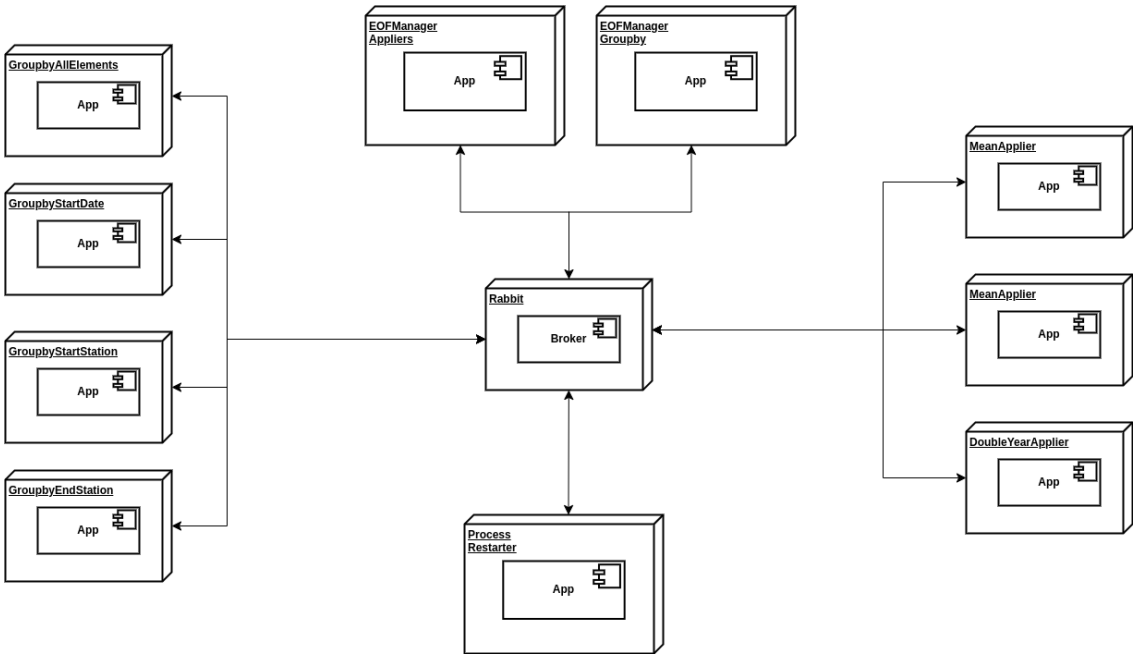


Figura 16: Diagrama de Despliegue (2)

4.3. Vista de Procesos

Diagramas de Actividades

En la Figura 17 se aprecia el primer diagrama de actividades, mostrando la actividad del cliente enviando datos estáticos para que sea procesado por los diversos nodos hasta llegar a almacenarse en memoria.

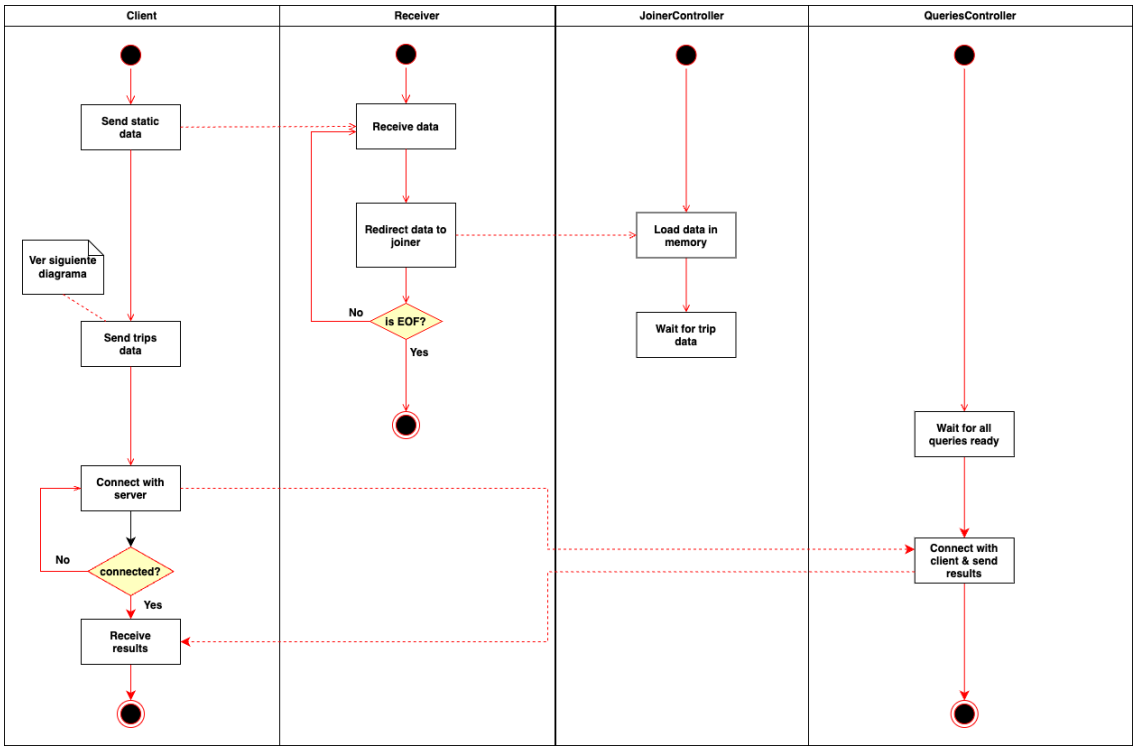


Figura 17: Diagrama de actividades - envío de datos estáticos

En las Figuras 18 y 19 se observa la actividad del cliente enviando viajes de forma iterativa, y cómo es procesado por los diversos nodos hasta poder obtenerse el resultado final.

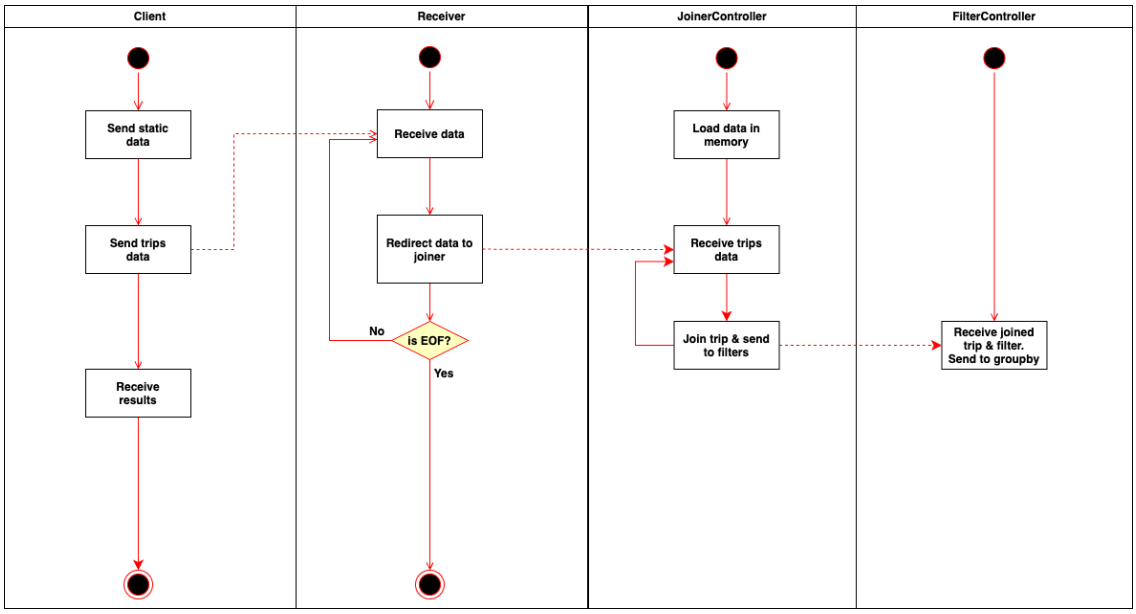


Figura 18: Diagrama de actividades - envío de viajes (1)

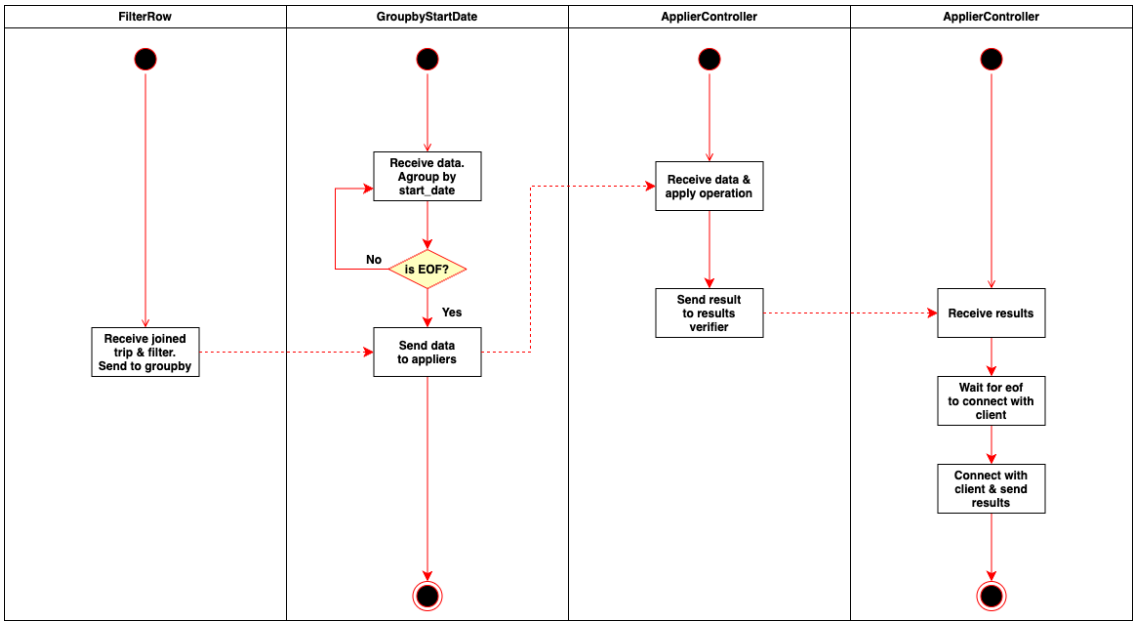


Figura 19: Diagrama de actividades - envío de viajes (2)

Cabe destacar que no se muestra la finalización de muchos componentes en estos diagramas, para no alargar la cantidad de los mismos. Se implementa graceful quit que al recibir una señal SIGTERM, se destruyen los recursos apropiados y cada nodo puede finalizar.

Por último, mostramos el diagrama de actividades del inicio de sesión del cliente, y del cierre del mismo por parte del SessionManager dentro del sistema.

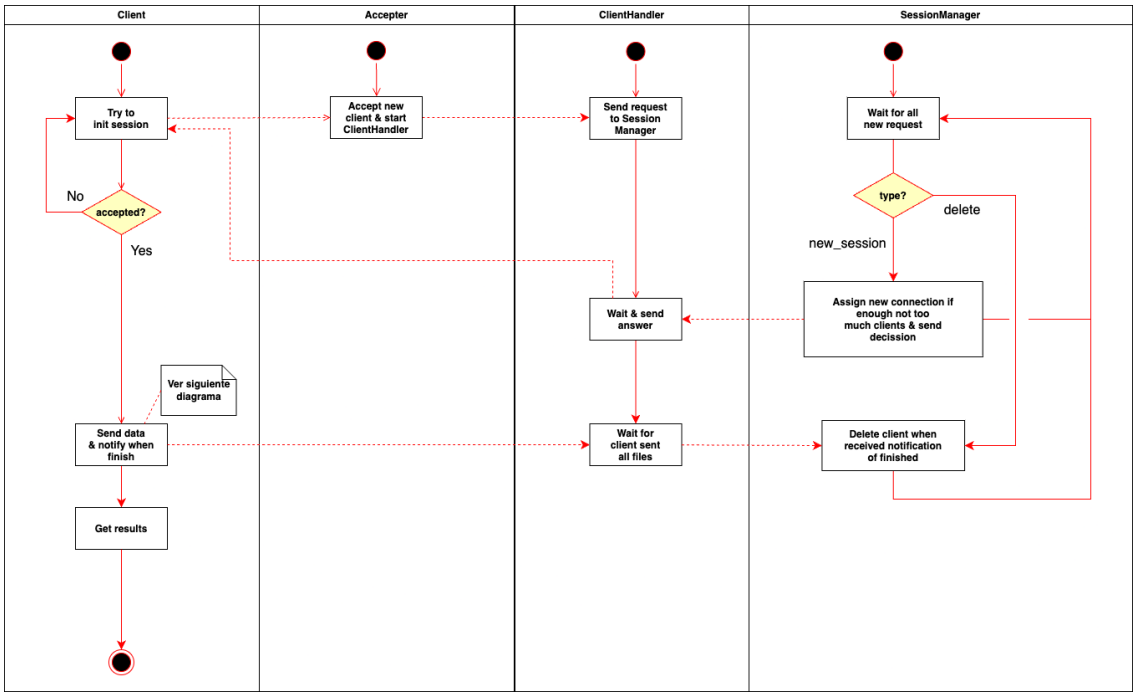


Figura 20: Diagrama de actividades - Inicio y cierre de sesión

Diagramas de secuencia

En esta subsección se expanden las actividades de los diagramas anteriores para mostrar secuencias de mensajes. Entre ellos, se aprecia el caso particular de lo que ocurre cuando llega el último mensaje de cada tipo de dato (tanto estático como de viajes).

En la Figura 21 se muestra la secuencia de envío de datos estáticos, particularmente haciendo foco en la situación de envío de un último paquete de tipo *station*.

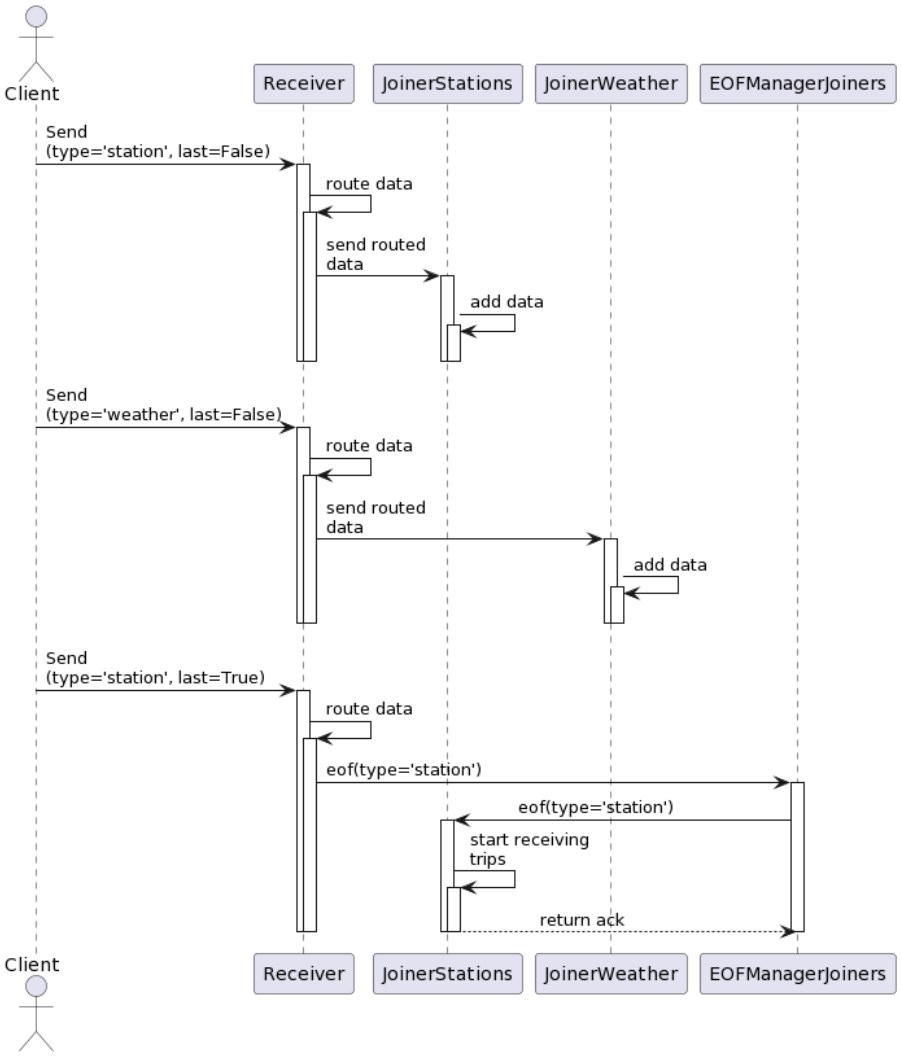


Figura 21: Diagrama de secuencia - Envío de datos estáticos

De una forma similar al diagrama anterior, se muestra lo que ocurre para el envío de un viaje (que no es el último) en la Figura 22.

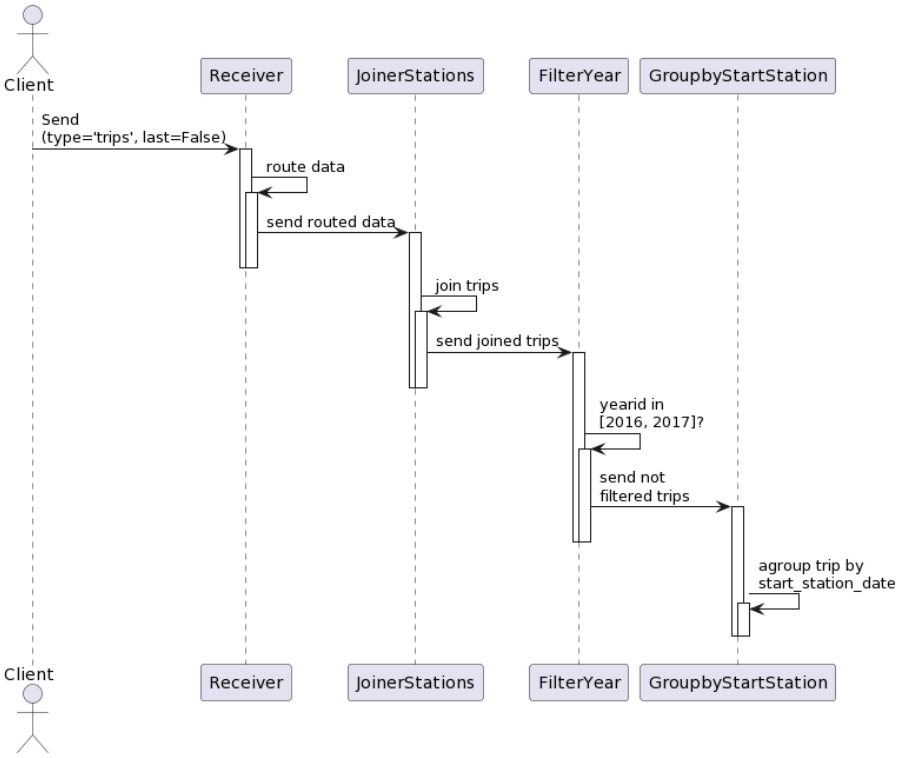


Figura 22: Diagrama de secuencia - Envío de viajes

A continuación, se muestran diagramas partidos (puesto que la longitud era muy grande para hacerlo en uno solo) mostrando lo que ocurre cuando se envía un último viaje. Se aprecia la secuencia de mensajes hasta que el cliente recibe los resultados.

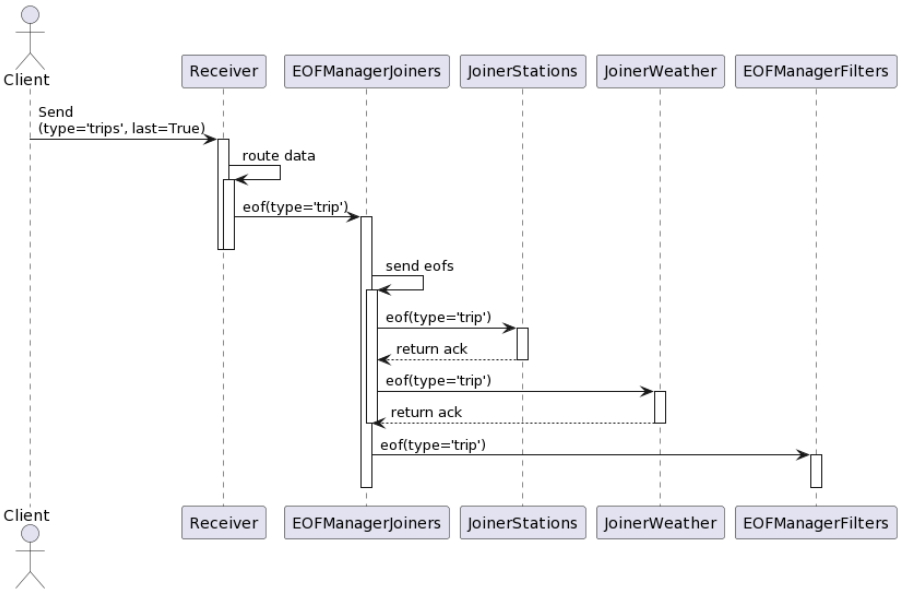


Figura 23: Diagrama de secuencia - Procesamiento de último viaje (1)

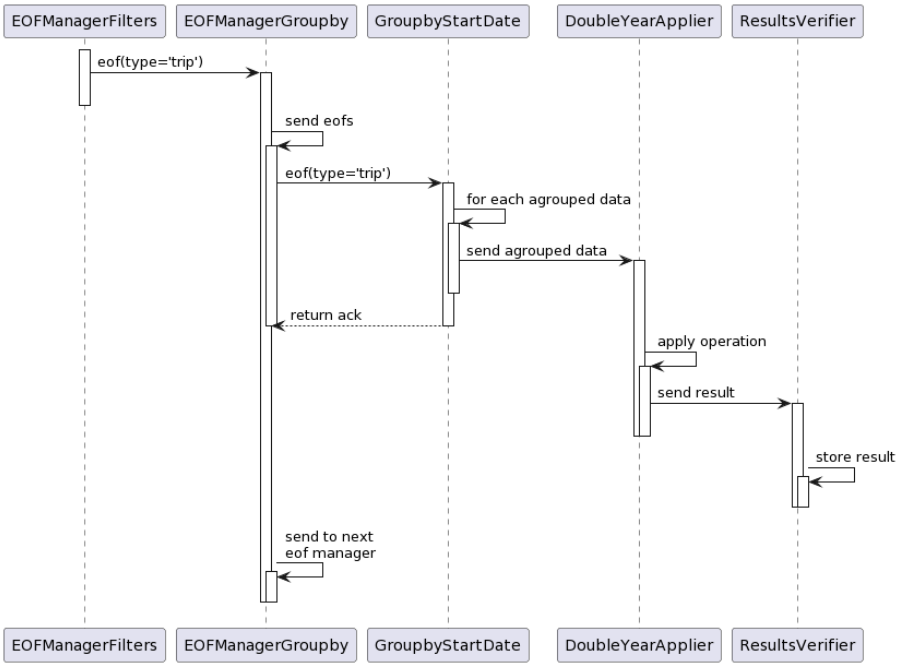


Figura 24: Diagrama de secuencia - Procesamiento de último viaje (2)

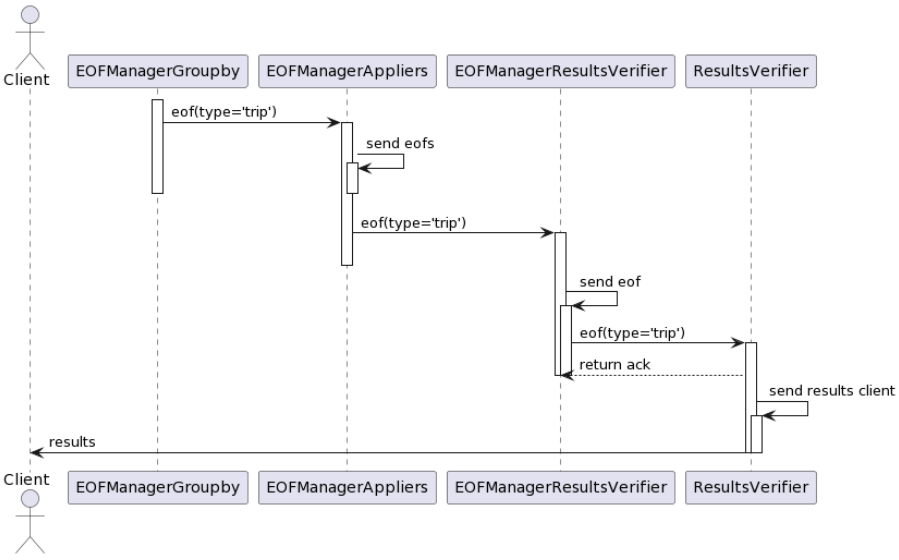


Figura 25: Diagrama de secuencia - Procesamiento de último viaje (3)

Sharding - Envío de datos estáticos y viajes a los Joiners

A continuación se muestra la secuencia de envío de datos a los Joiners. Suponiendo un caso en el que hay 5 Joiners, solo se envían los datos estáticos a un subconjunto de estos. En este ejemplo, solo a dos. Luego, los viajes deben ser procesados una sola vez. Por lo tanto, el viaje sólamente debe ser enviado a uno de estos dos Joiners que lo puede recibir.

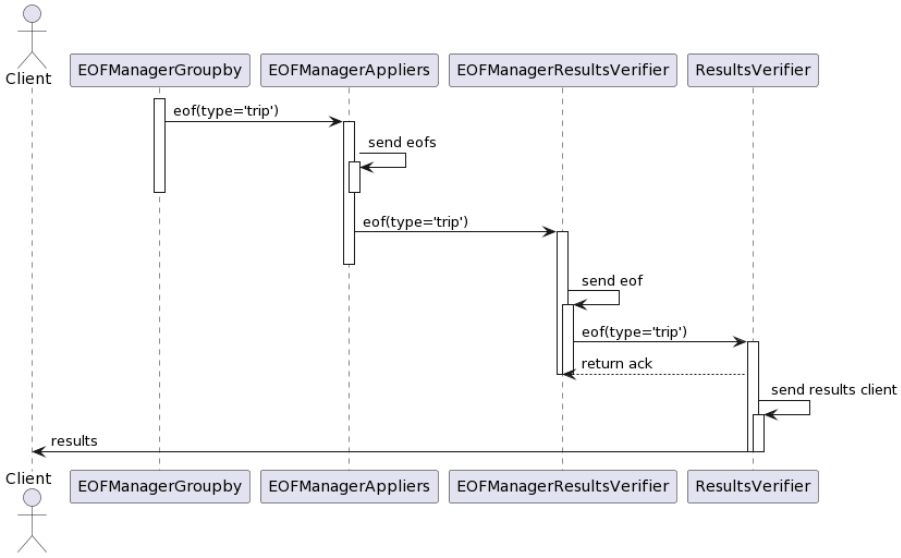


Figura 26: Diagrama de secuencia - Sharding en los Joiners

Persistencia

Para la tolerancia a fallos se implementó un sistema de persistencia genérico que se incorporó en cada entidad que tenía un estado intermedio.

El sistema de persistencia se basa en *buckets* y *collections*. Un bucket es una unidad de persistencia atómica que permite guardar información en formato clave-valor. Es considerada atómica ya que todos los datos que se guarden un mismo bucket se persisten todos juntos o no se persiste ninguno.

Cada bucket puede tener collections. Las collections permiten guardar datos de distintos dominios en un mismo bucket, sin que se mezclen. Esto es útil cuando estos datos de distintos dominios, por razones de consistencia, deben persistirse juntos.

Un ejemplo de uso de este sistema se puede observar en los GroupBy. Aquí se tienen datos pertenecientes a dos dominios:

- Por un lado están los resultados parciales agrupados, que deben ser persistidos
- Por otra parte está el filtro de batches duplicados, que si bien está correlacionado con la información agrupada, no es del mismo dominio.

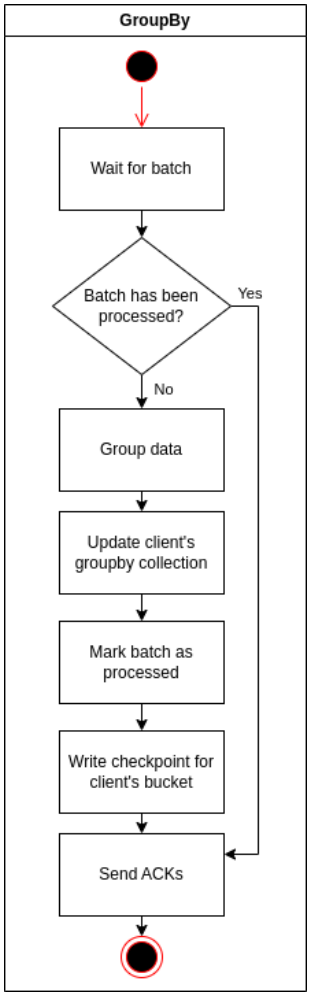
Para persistir esta información manteniendo la consistencia se crea un único bucket que tiene dos collections, una collection denominada `groupby` que contiene los datos agrupados; y otra denominada `dup_filter` que contiene los datos del filtro de duplicados.

El archivo resultante podría verse de la siguiente forma:

```
{
    "groupby#result_1": "value-1",
    "groupby#result_2": "value-2",
    "dup_filter#batches": [1, 2, 3, 4, 5]
}
```

El formato utilizado para los archivos es JSON, ya que nos da suficiente flexibilidad como para poder reutilizar la implementación en distintas entidades que guardan distintos tipos de datos.

En el diagrama de actividades que se muestra a la derecha se puede observar como se actualiza la información en los procesos de tipo GroupBy. En el resto de los procesos el procedimiento es análogo.



Elección de Líder

A continuación mostramos cómo se maneja la recepción de un mensaje del tipo ELECTION desde el controlador *ElectionStarter*.

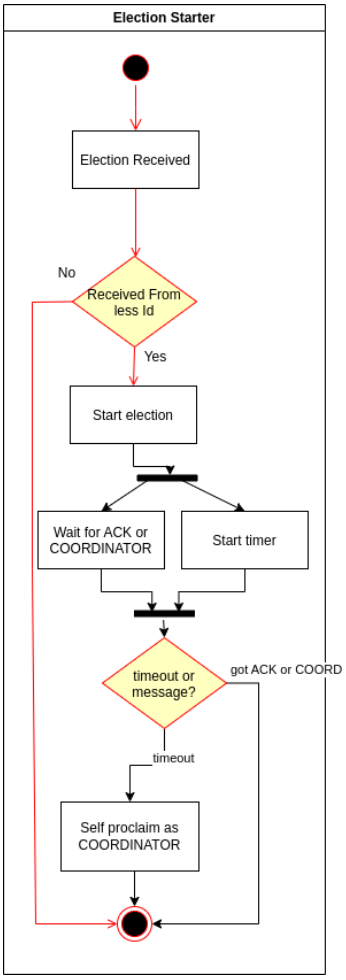


Figura 27: Diagrama de actividad - Manejo del mensaje ELECTION

4.4. Vista de Desarrollo

Diagrama de Paquetes

Se muestran dos diagramas de paquetes para mostrar los artefactos que componen al sistema.

En la Figura 28 se muestra un diagrama general del sistema. En la Figura 29 se hace foco en el paquete *workers* que indica la clase que procesan datos en el sistema.

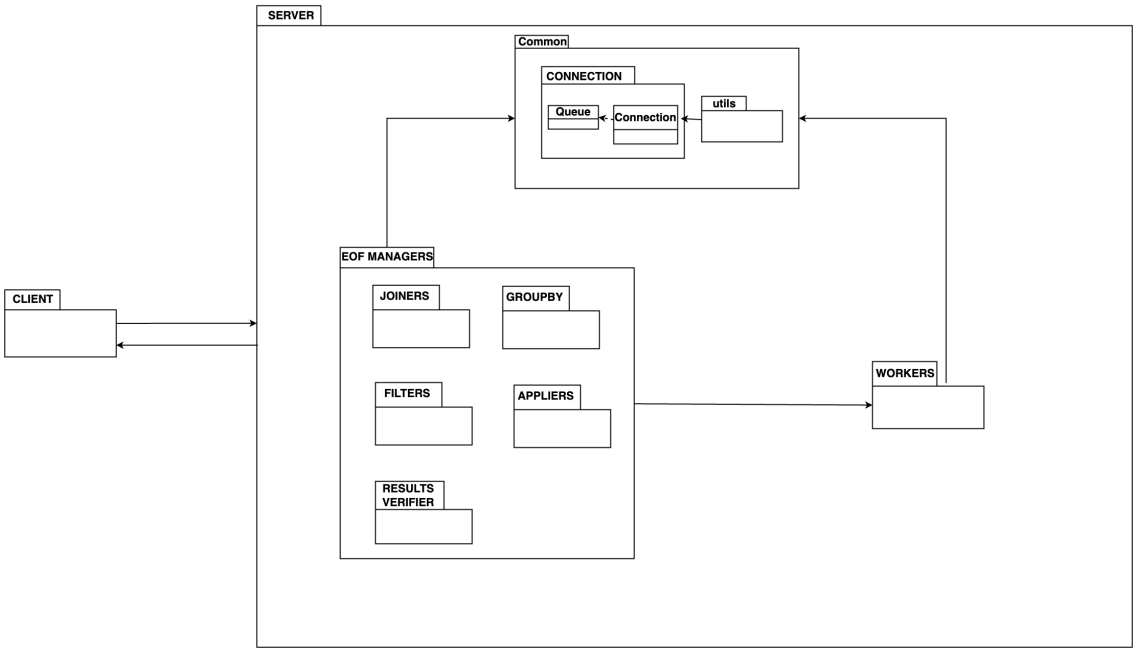


Figura 28: Diagrama de Paquetes - interacción general del sistema

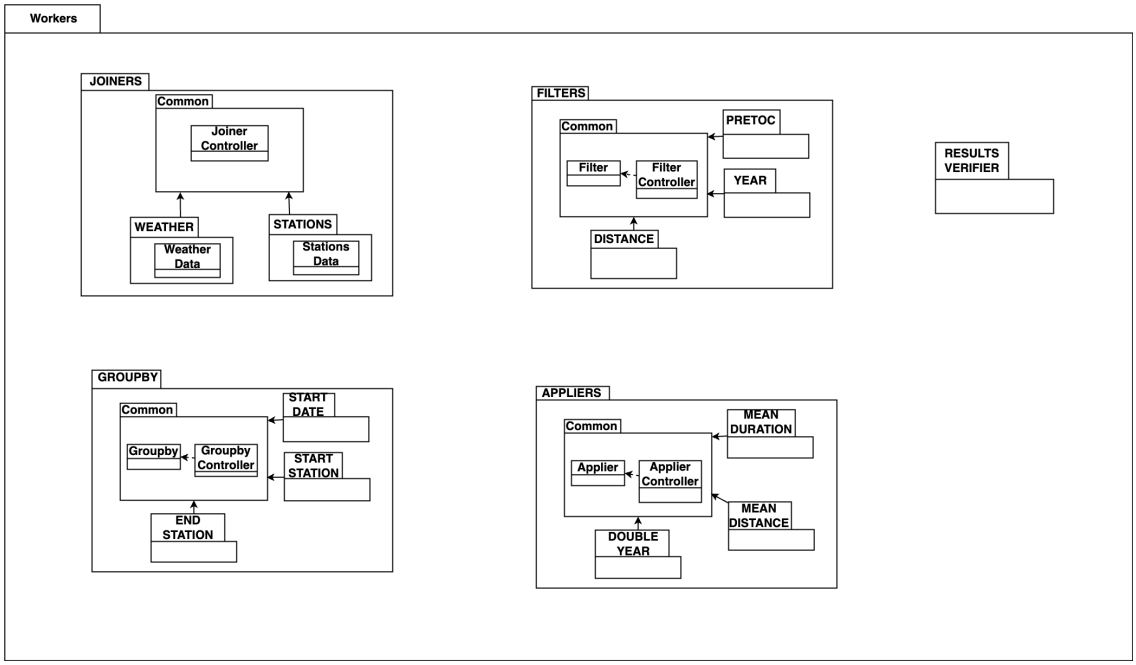


Figura 29: Diagrama de Paquetes - workers

5. Issues pendientes

En esta sección se desarrollan los *"known issues"* que, por cuestiones de tiempo, no se pudieron abordar.

Código

1. Utilizar logging en lugar de prints. Puesto que se decidió eliminar los loggings que arroja rabbit, si se utiliza la librería logging, no hay manera de eliminarlos. Si bien es una mejor práctica permitir la elección de un logging_level para visualizar a gusto los mensajes que uno elija, la estructura utilizando con prints igualmente sigue el estándar del trabajo anterior.
2. Mejor documentación de las clases y funciones. Se decidió incluir comentarios en aquellas funciones que tengan una mayor complejidad o una importancia mayor que el resto, pero una mejor práctica es documentar todas las funciones de forma correcta.
3. En ciertos componentes se tiene código repetido (como en los *EOFManagers*).
4. Se cuenta con algunos ifs/else anidados.

Diagramas

1. No conseguí que el tamaño de la fuente del documento y de los diagramas sea el mismo puesto que para latex algunas imagenes necesité agrandarlas más para que se vean mejor, y las fuentes sufrieron variaciones a lo diseñado.