FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

[75.74] SISTEMAS DISTRIBUIDOS I

1ER CUATRIMESTRE 2023

Alta disponibilidad y Tolerancia a Fallos

Índice

1. Introducción

En el siguiente documento se presenta la arquitectura de Bike Rides Analyzer, un sistema distribuido de alta disponibilidad y tolerante a fallos de análisis de viajes en bicicleta de las ciudades de Montreal, Toronto y Washington.

El sistema esta compuesto por un broker de RabbitMQ y un servidor con múltiples servicios destinados a operar y transformar los datos para otorgar métricas agregadas a múltiples clientes.

2. Scope

El alcance de Bike Rides Analyzer es, a partir de los datos de viajes, estaciones y clima, poder responder las siguientes consultas:

- Obtener la duración promedio de viajes en días con precipitaciones mayores a 30mm.
- Obtener las estaciones de salida que duplicaron la cantidad de salidas entre 2016 y 2017.
- Obtener las estaciones de llegada para las cuales los ciclistas tuvieron que recorrer, en promedio, más de 6km.

Esto se puede ver reflejado en el siguiente diagrama de casos de uso.

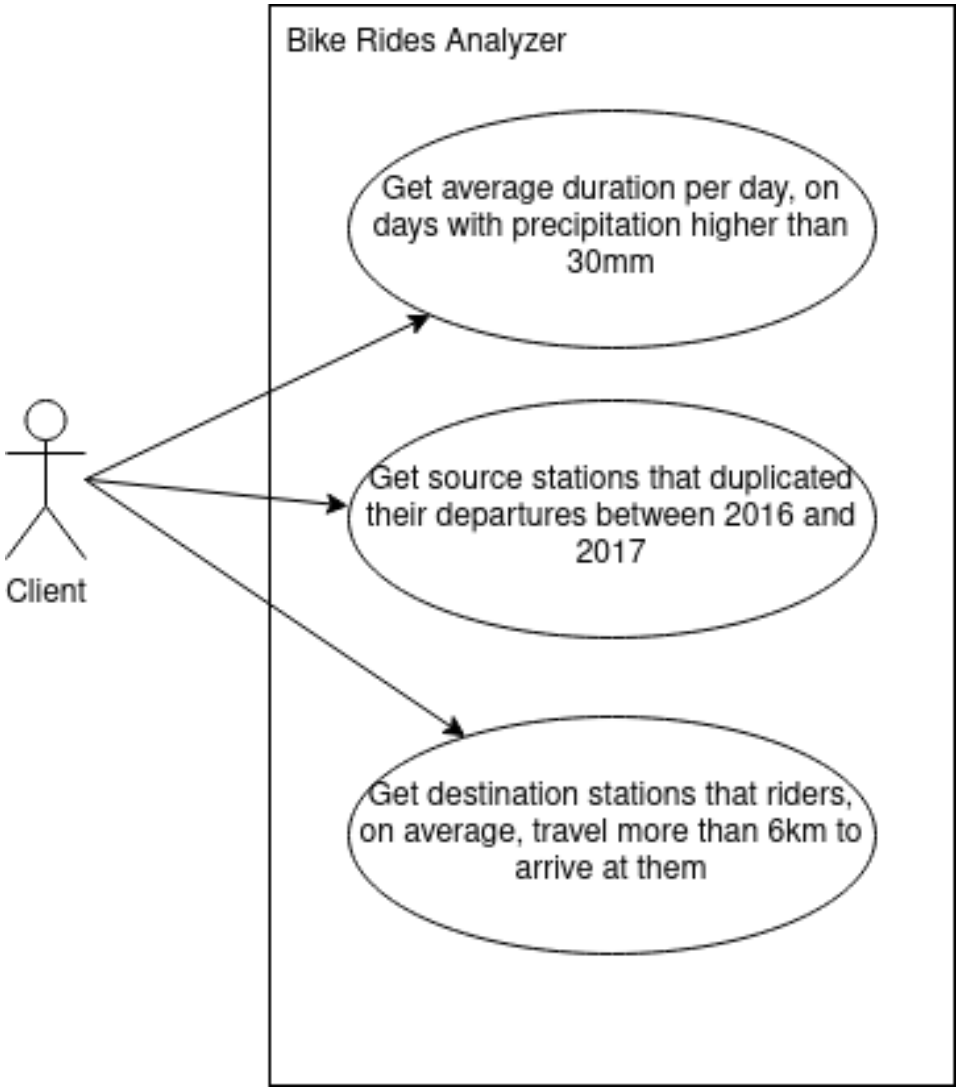

Figura 2.1: Diagrama de casos de uso de Bike Rides Analyzer

El sistema además deberá tolerar fallos, ya sea caidas de réplicas individuales o múltiples, otorgando resultados correctos para cada query, y garantizar alta disponibilidad a los clientes.

3. Arquitectura de software

Bike Rides Analyzer se compone de distintos servicios modulares con responsabilidades marcadas:

- Loader: Funciona como el punto de entrada al sistema. Recibe los datos de los clientes y los ingesta para que sean consumidos por los nodos consumers.
- Consumers: Se encargan de obtener los datos, eliminar columnas innecesarias para el procesamiento más adelante y enviar los datos resultantes. En el sistema hay Trips Consumers, encargados de consumir datos de viajes, Stations Consumers, que consumen datos de estaciones, Weather Consumers, que consumen datos del clima, y un único Metrics Consumer, que funciona como pasamanos de métricas finales al Loader, para luego poder servirlas al cliente particular que hizo esa consulta.
- Filters: Estos se encargan de filtrar datos por una o múltiples columnas siguiendo alguna condición predefinida.
- Joiners: Los nodos de tipo Joiner son encargados de hacer una junta entre los datos estáticos (clima y estaciones) con los datos streameados (viajes). Persistirán además sus memtables para poder soportar eventuales caidas y continuar desde un checkpoint.
- Aggregators: Se encargan de agregar datos dada una clave. Pueden ser agregaciones de promedios o de contadores. Al igual que los joiners, deberán guardar sus resultados agregados periódicamente para no perder información. Además de esto deberán filtrar datos ya procesados ya que sus agregaciones no son idempotentes.
- Appliers: Los nodos Applier ejecutan una funcion sobre los campos de algun tipo de dato para obtener resultado derivado de los mismos. En el caso de Bike Rides Analyzer, el Haversine Applier es el nodo encargado de tomar coordenadas de origen y destino de los viajes y calcular la distancia de Haversine.
- Watcher: Se encargan de revivir a nodos caidos (inclusive entre ellos). Cada nodo reportará ante un watcher periódicamente a modo de healthcheck. Entre ellos siempre habrá un único watcher activamente escuchando, y esto se garantiza mediante el algoritmo de consenso distribuido, Bully.

4. Objetivos arquitectónicos

- Escalabilidad: Bike Rides Analyzer proporciona escalabilidad a modo de mantener la latencia de las operaciones acotada ante mayores volúmenes de carga de datos. Para cumplir este objetivo, casi todos los nodos de procesamiento son escalables (excluyendo al Loader y al Metrics Consumer). De esta manera, se podrá distribuir la carga en mayores unidades de procesamiento de asi ser requerido. Para lograr esta distribución, se utiliza como herramienta de mensajería a RabbitMQ.
- Tolerancia a fallos: A pesar de caidas de nodos, la información procesada será persistida y recuperada de manera tal de devolver resultados consistentes independientemente de los fallos que puedan ocurrir en el sistema.
- Alta disponibilidad: Los clientes podrán obtener resultados a pesar de que hayan servicios caidos.
- Múltiples consultas en paralelo: Se puede configurar el sistema para soportar una cantidad máxima de clientes procesando consultas en paralelo.

5. Vista Lógica

5.1. Jerarquías

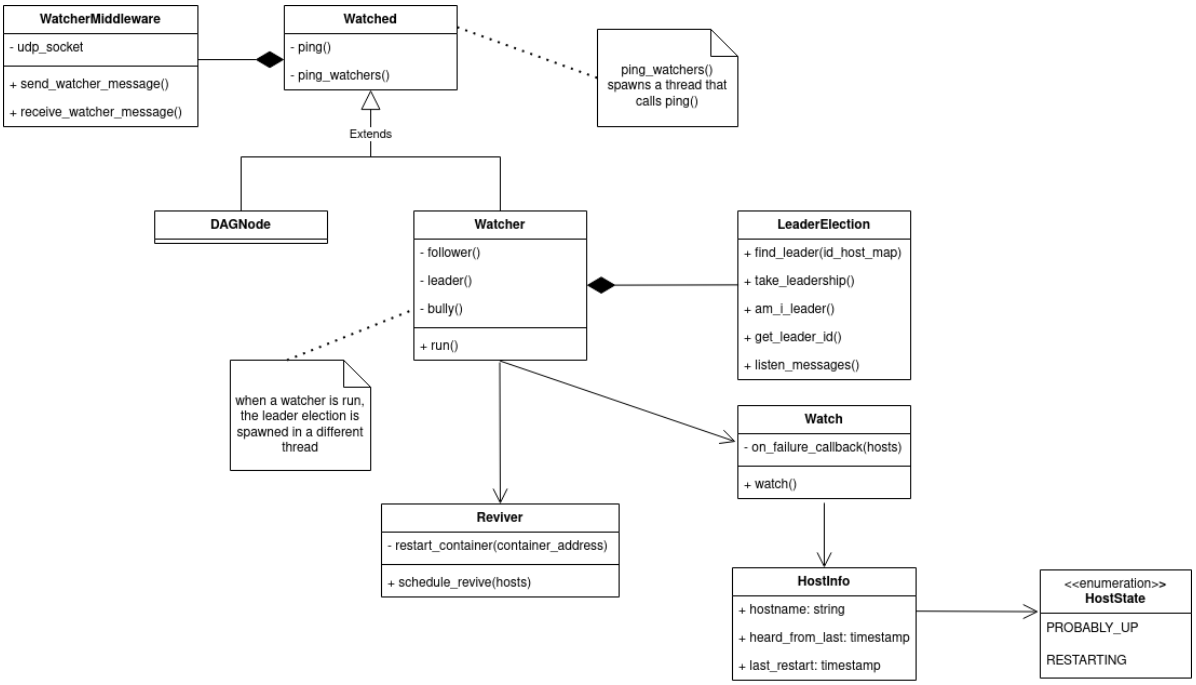

Figura 5.1: Abstracciones fundamentales de Bike Rides Analyzer

El sistema fue diseñado inicialmente considerando responsabilidades bien definidas para cada nodo, desde filtros hasta agregadores. Toda esa jerarquía hereda de la clase abstracta DAGNode. Dentro de esta jerarquía entonces encontraremos encapsuladas las responsabilidades de los nodos encargados de transformar y agregar los datos proporcionados por los clientes. Luego, al agregar tolerancia a fallos, fue necesario considerar que cada nodo pueda emitir una señal de vida de manera periódica. De tal manera, se generó la abstracción Watched, que encapsula ese comportamiento. Por otro lado, los healthchecks previamente mencionados deben ser registrados por otra entidad que además pueda encargarse de, en caso de detectar una falla, revivir a los nodos posiblemente caidos. Esto está encapsulado en la clase Watcher, que cabe destacar a su vez es Watched, ya que como cualquier otro nodo del sistema, puede caerse. En particular, cada Watcher hará uso de otras abstracciones, como lo son el Reviver y Watch para revivir nodos y para escuchar healthchecks, respectivamente. Internamente se guardará metadata sobre cada nodo observado junto con el estado que el Watcher considere para ese nodo. Por otra parte, cada Watcher tendrá una abstracción que logrará generar consenso entre el resto de los nodos Watcher para decidir cual será el lider y por ende quien será el encargado de escuchar por healthchecks de otros nodos.

En el siguiente diagrama se muestra como la jerarquía se extiende para otorgar funcionalidad a los nodos Filter del pipeline.

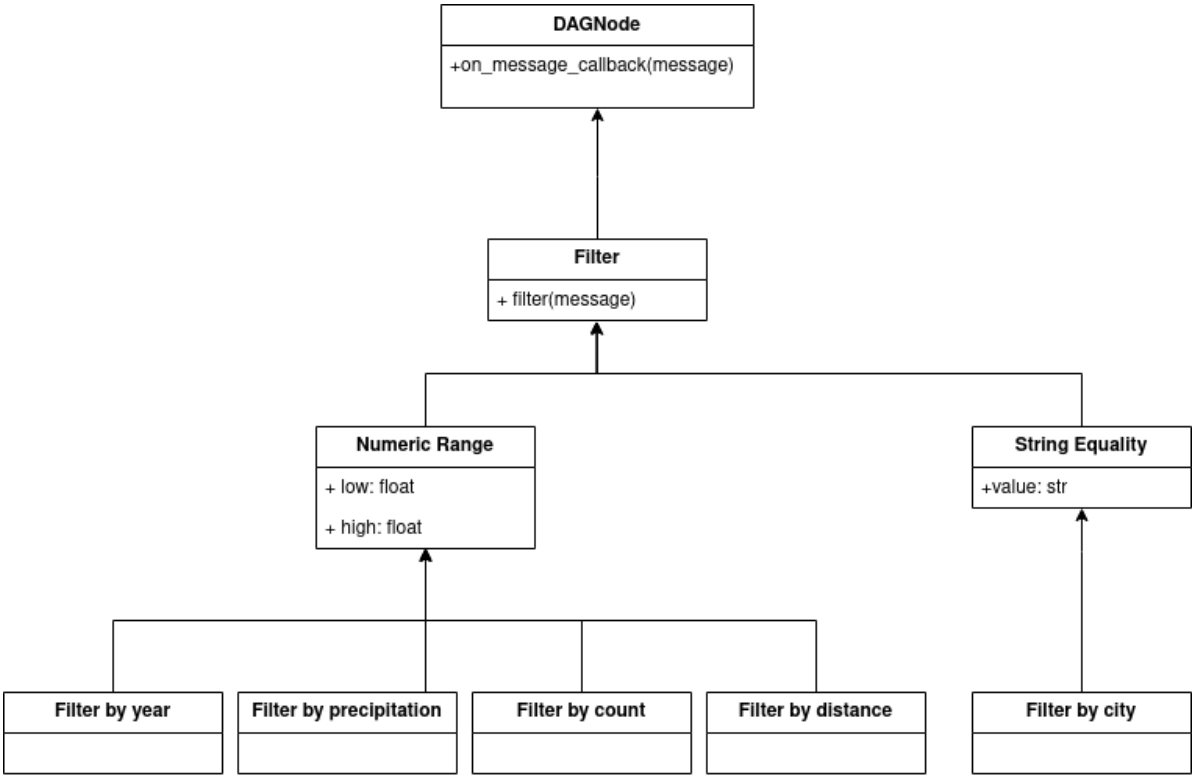

Figura 5.2: Diagrama de clase, jerarquía de Filters

6. Vista Física

6.1. Robustez

Los diagramas de robustez están divididos de manera tal de ilustrar los diferentes nodos que intervienen en la resolución de cada una de las consultas.

Para todos los casos, los datos son inicialmente ingestados al sistema por el cliente. Un patrón que notaremos tanto para los filtros, los consumers, los joiners, aggregators y appliers es que podrán ser escalados para soportar un mayor volumen de carga. Esto es así debido a que los mismos no comparten ningun tipo de estado entre ellos, de manera tal que se puede aprovechar la distribución de carga para generar paralelismo en el sistema.

Se comenta de antemano que el motivo de que la mayoria de los nodos esten ubicados dentro de una caja replicada, con una cola tambien alli dentro, es para mostrar que cada nodo esta replicado con una cola individual. Es decir, si hubiesen 5 weathers consumers trabajando en paralelo, existirian tambien 5 colas, una individual para cada uno de ellos.

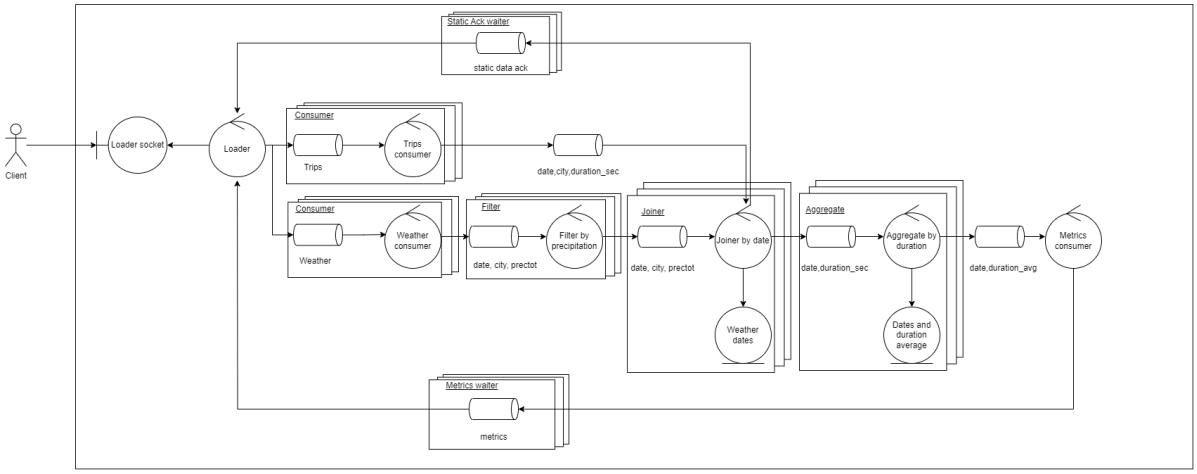

Figura 6.1: Diagrama de robustez que incluye a los nodos relevantes en el calculo del promedio de duración de viajes en dias con altas precipitaciones

Para esta consulta, cada worker de tipo Weather Consumer se encargará de extraer los datos del clima relevantes para la consulta y enviarlos al grupo de filtros Filter By Precipitation. Luego de ser procesada esta data estática, se guardará en el Joiner. Los joiners podrán ser replicados debido a que la carga estática será broadcasteada a ellos. Al llegar la data de trips, cada uno hará la operación de join de manera paralela. Los nodos Aggregate by Duration recibiran de manera independiente datos de la etapa de join para poder agregarlos (en este caso calcularán un rolling average). Estos a su vez irán guardando los datos agregados en memoria. Finalmente, al recibir el mensaje de End Of Stream, los datos agregados serán enviados al Metrics Consumer, que será el encargado de enviarsela al loader, quien en un hilo combinara los resultados parciales a fin de enviarle al cliente el resultado final.

Se tiene a continuación el diagrama de robustez con los nodos que intervienen en la resolución de la segunda consulta.

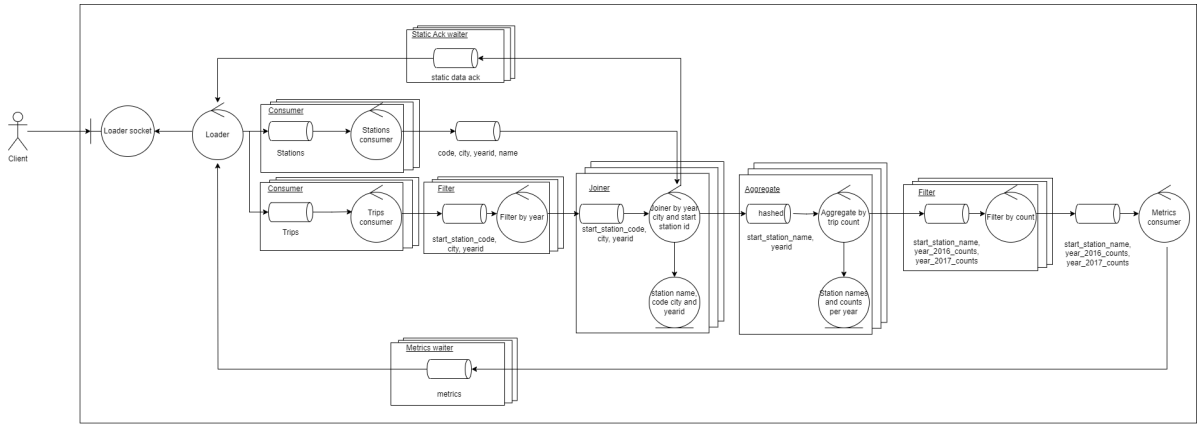

Figura 6.2: Diagrama de robustez que incluye a los nodos relevantes en el calculo de las estaciones que duplican las salidas entre 2016 y 2017

Como en la consulta anterior, los grupos de nodos que se presentan tienen la misma responsabilidad, agregar, agrupar y filtrar. Estos son desplegables de manera independiente y escalables, siguiendo la misma lógica.

Por último, se muestra el diagrama de robustez relevante a la resolución de la tercera consulta.

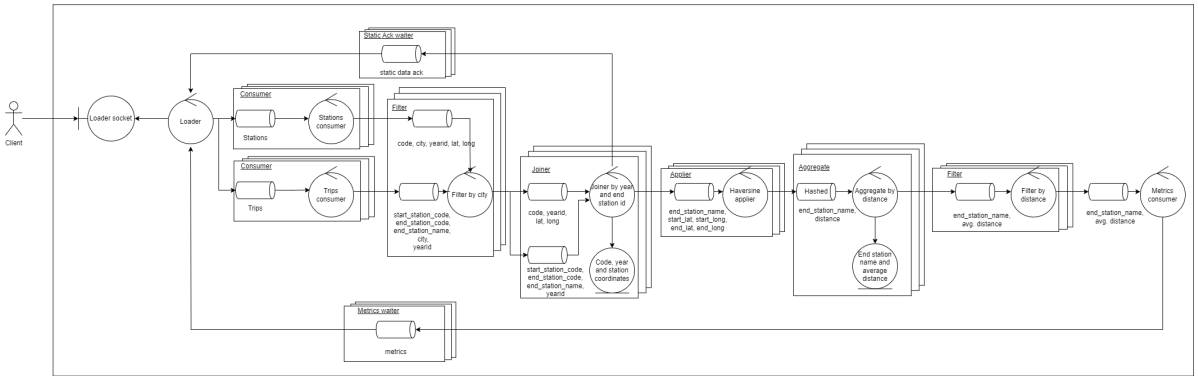

Figura 6.3: Diagrama de robustez que incluye a los nodos relevantes en el calculo de las estaciones con media de viaje mayor a 6km en Montreal

En este caso, se agrega otro grupo de nodos encargados de calcular distancias, Haversine Applier. Nuevamente replicables debido a que no comparten ningun tipo de estado. En este caso vemos además como se pueden reutilizar los nodos para filtrar datos sin importar el origen de los mismos, como es el caso del grupo de Filter By City. Si se quisiera ahora agregar otra ciudad a la consulta, bastaría con cambiar la configuración de estos filtros para que tome en cuenta a esas nuevas ciudades.

6.2. Despliegue

A continuación se muestra el diagrama de despliegue del sistema.

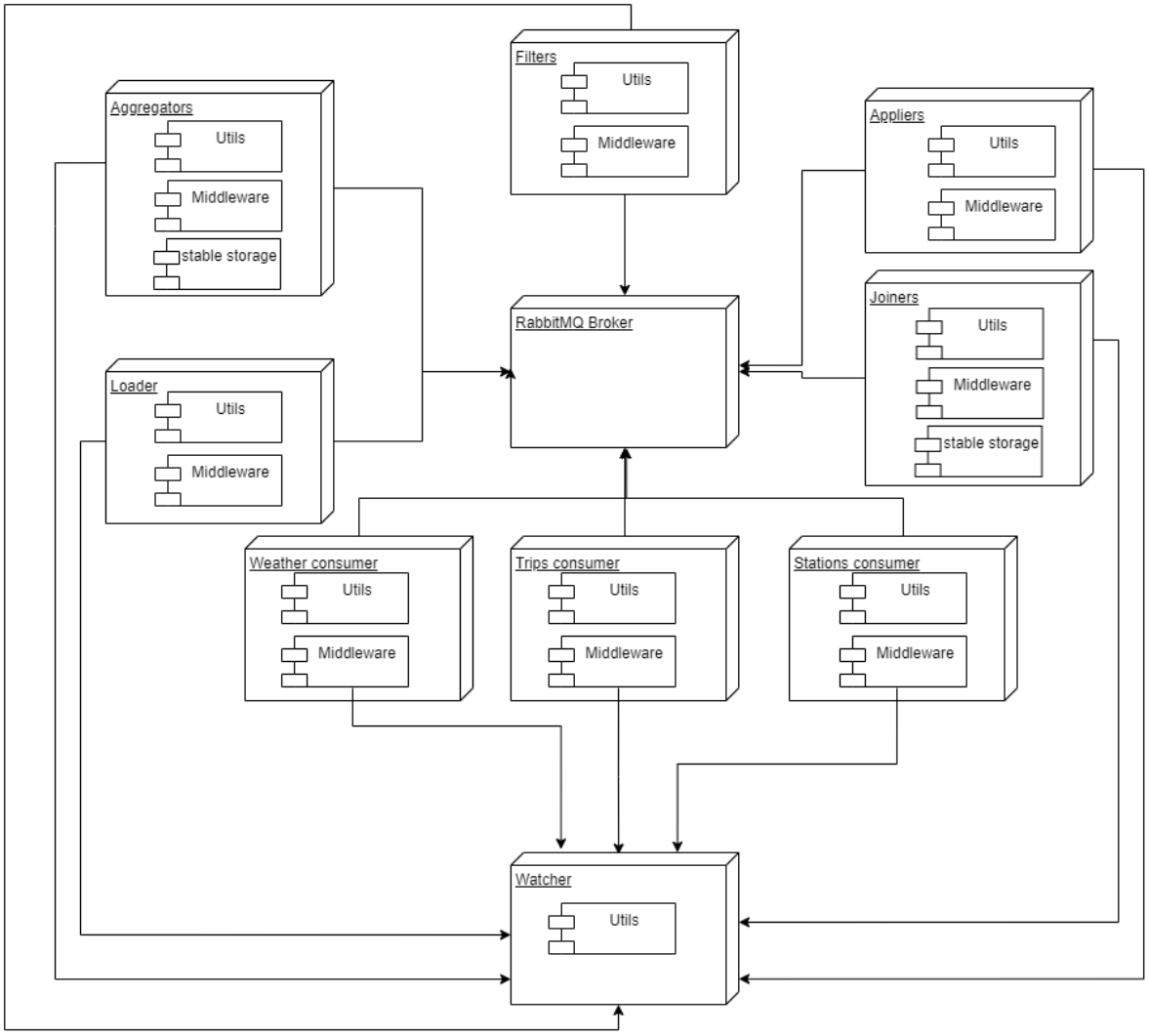

Figura 6.4: Despliegue de Bike Rides Analyzer

Se puede apreciar una fuerte dependencia del sistema al broker de RabbitMQ. Esto es asi ya que todos los servicios utilizan al broker para poder comunicarse entre si. El Loader, además, funciona como servicio de frontera con el cliente. Para comunicarse utiliza un socket TCP/IP, de tal manera que se aisla al cliente del modelo de comunicación interno del sistema. Los nodos de tipo Filter, Aggregator y Joiner se agrupan en un único componente genérico simplemente para no cargar la figura, pero cada uno de ellos podrá ser desplegado en un nodo independiente.

Ademas, los nodos tienen una dependencia con el Watcher. El watcher es una entidad replicada, a pesar de que haya un solo watcher lider, justamente porque si ese lider cae habra otro watcher que lo reemplace. Los nodos dependen del watcher para iniciar su funcionamiento y luego deben enviar un ping para informar su estado.

Dentro de cada nodo se puede visualizar en forma de layer los modulos que utilizan. Se puede ver que cada nodo del sistema tiene un modulo de utils donde se agrupan diversas funciones comunes a fin de reutilizar codigo. Ademas, la mayoria de nodos del sistema tienen

tambien un modulo de Middleware que permite la comunicación via Rabbit. Los nodos de tipo Joiner y Aggregator realizan un almacenamiento en disco para persistir los snapshots de data. Como comentario, en realidad todos los nodos persisten data a disco (lo hacen con los EOFs recibidos), pero esta persistencia de EOFs esta incluida dentro del modulo de Middleware.

7. Vista de Desarrollo

7.1. Diagrama de Paquetes

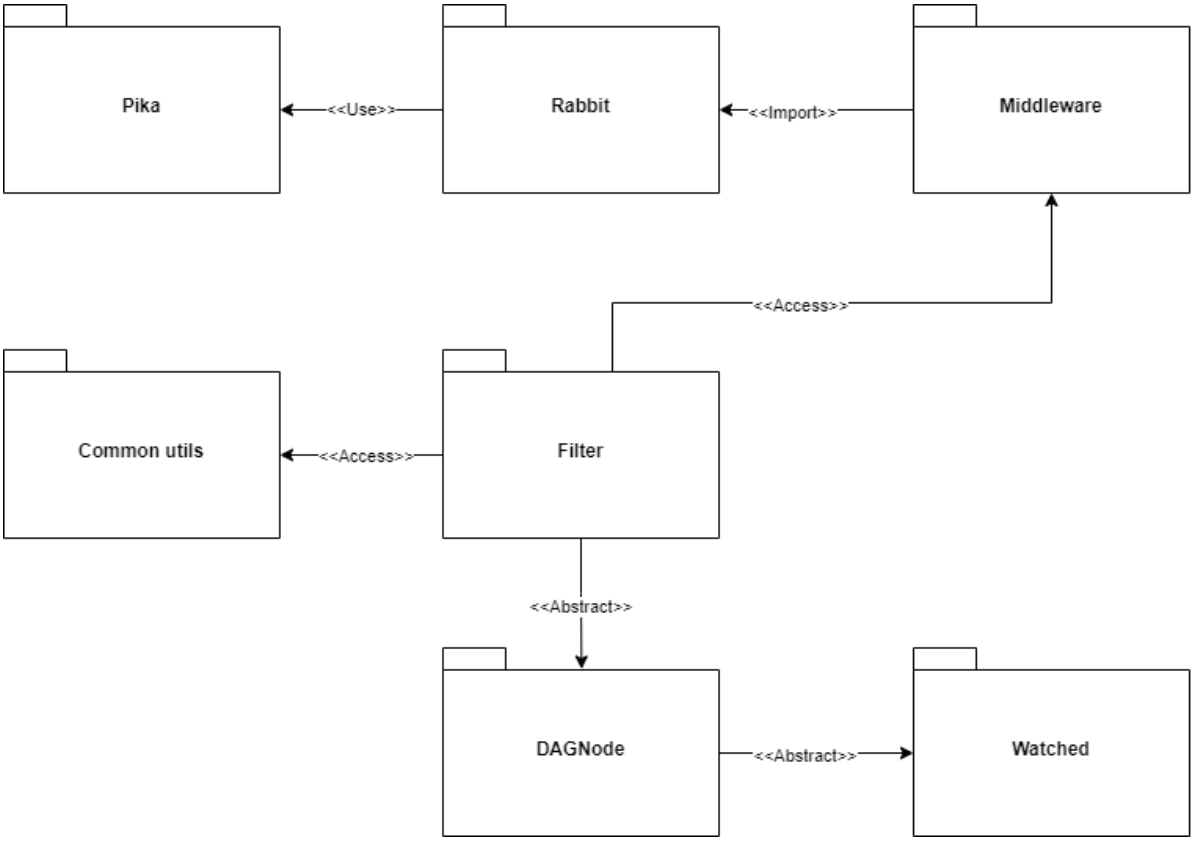

Figura 7.1: Diagrama de paquetes enfocado en los Filters

Se puede apreciar como se estructura en este caso el paquete de filtros y cuales son las dependencias que tiene con otros. Este tendrá acceso al paquete Middleware, que le proveerá métodos para poder comunicarse, tanto para recibir como para enviar data. A su vez, el paquete Middleware importará funcionalidad del paquete Rabbit, destinado a encapsular funcionalidad del cliente de RabbitMQ, en este caso Pika. Por otra parte, los filtros importarán el paquete Common Utils para acceder a abstracciones de comunicación al momento de enviar y recibir mensajes. Cabe destacar que cada filter además es una abstracción de DAGNode, lo cual permite reutilizar funcionalidad base.

Los nodos implementan una clase Watched, con la que comparten metodos que les permiten comunicarse con el Watcher lider, haciendole saber que estan levantados y en funcionamiento.

Si bien el diagrama muestra la perspectiva para el paquete Filter, en mayor o menor medida el resto de los paquetes de nodos se comportan de la misma manera.

8. Vista de Procesos

8.1. Cálculo de distancias

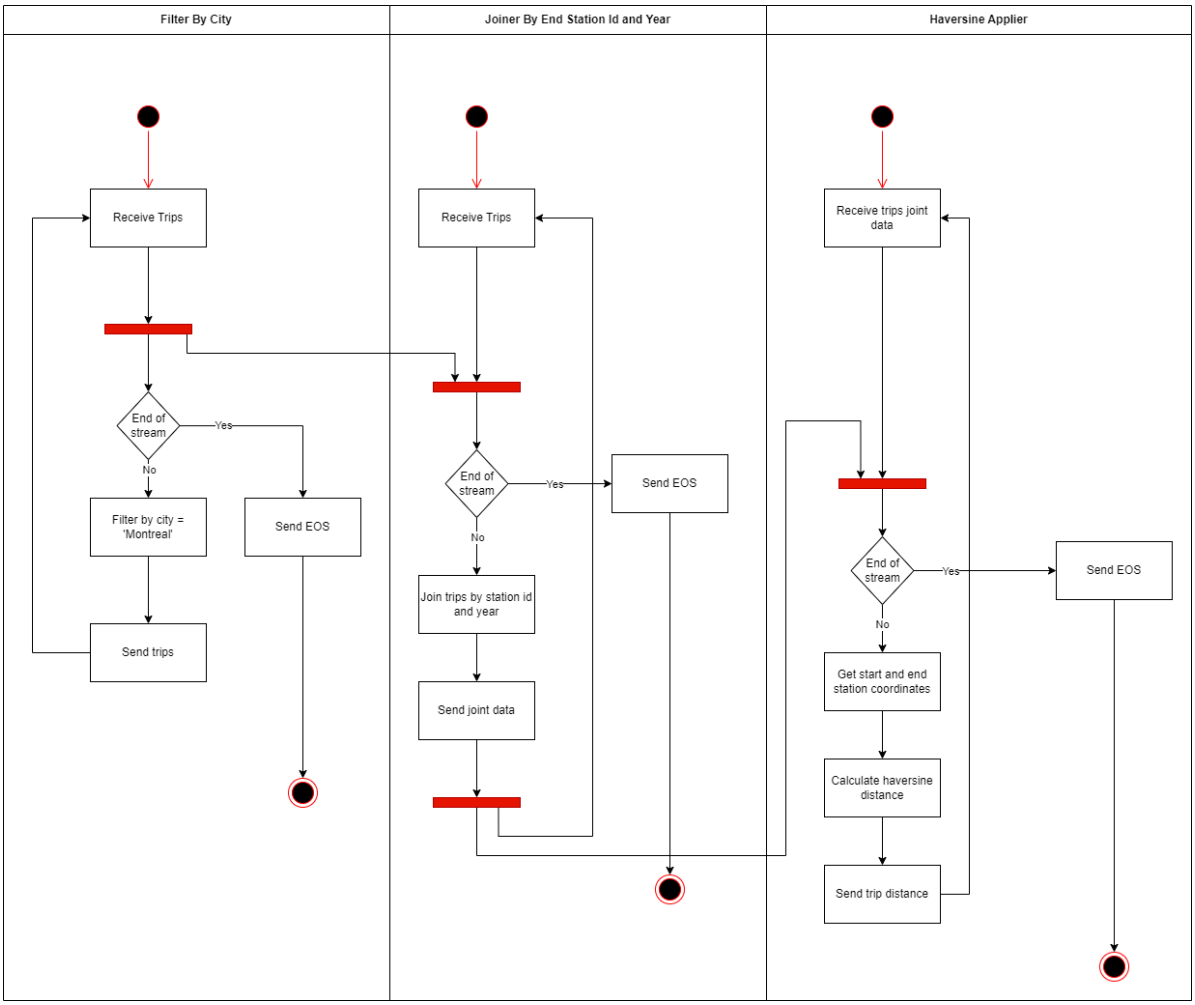

Figura 8.1: Diagrama de actividades, filtro, junta y calculo de distancias

En el siguiente diagrama de actividades, se muestra parte de la resolución de la tercera consulta. Vemos que las diferentes actividades, filtro, junta y calculo de distancias, son realizadas en paralelo. A medida que el filtro selecciona aquellos viajes de la ciudad de Montreal, estos son enviados al Joiner, que los recibe y hace la junta con sus datos estáticos de estaciones, y al realizar la junta envía los datos al Haversine Applier, que se encargará de recibir los datos de la junta, extraer las coordenadas de estaciones de inicio y final y calcular las distancias. Una vez llegado el end of stream, se procederá a enviar la información de una etapa en otra, de esta manera sincronizando a los procesos sobre el final del stream.

8.2. Logs de datos

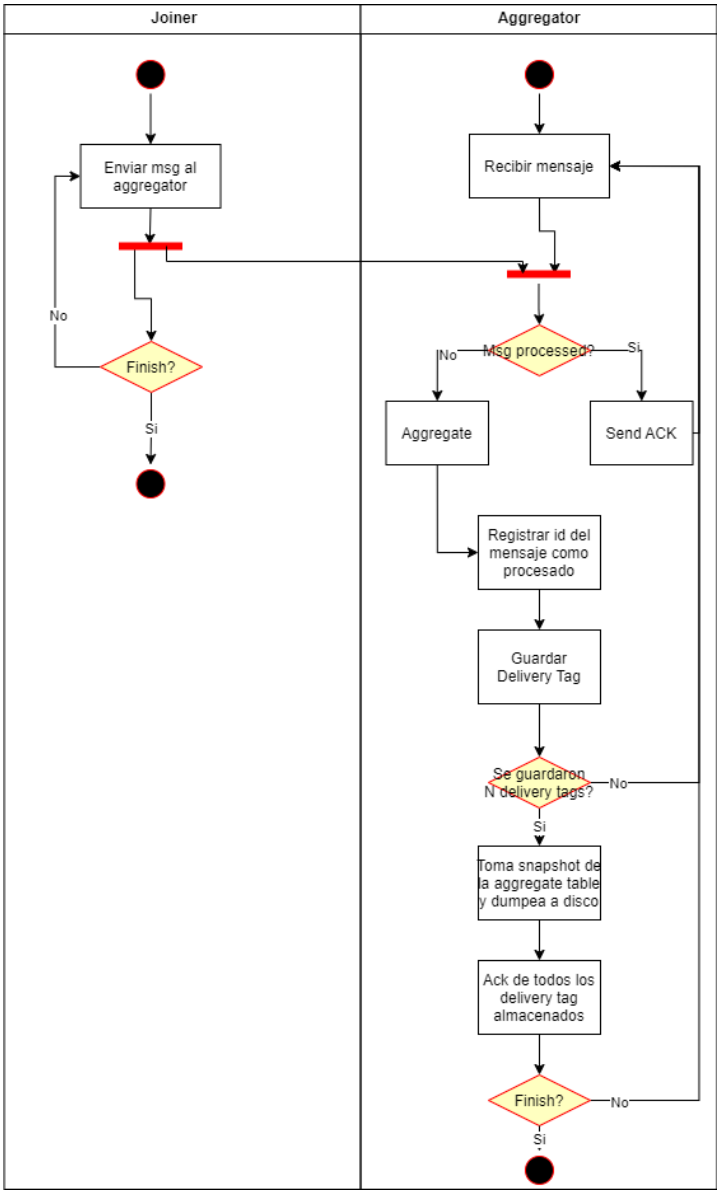

Figura 8.2: Diagrama de Logs para data

El diagrama ilustra el comportamiento de los Aggregator para loggear cuando reciben mensajes de trips o static data desde los joiners. El comportamiento de los Joiners es analogo.

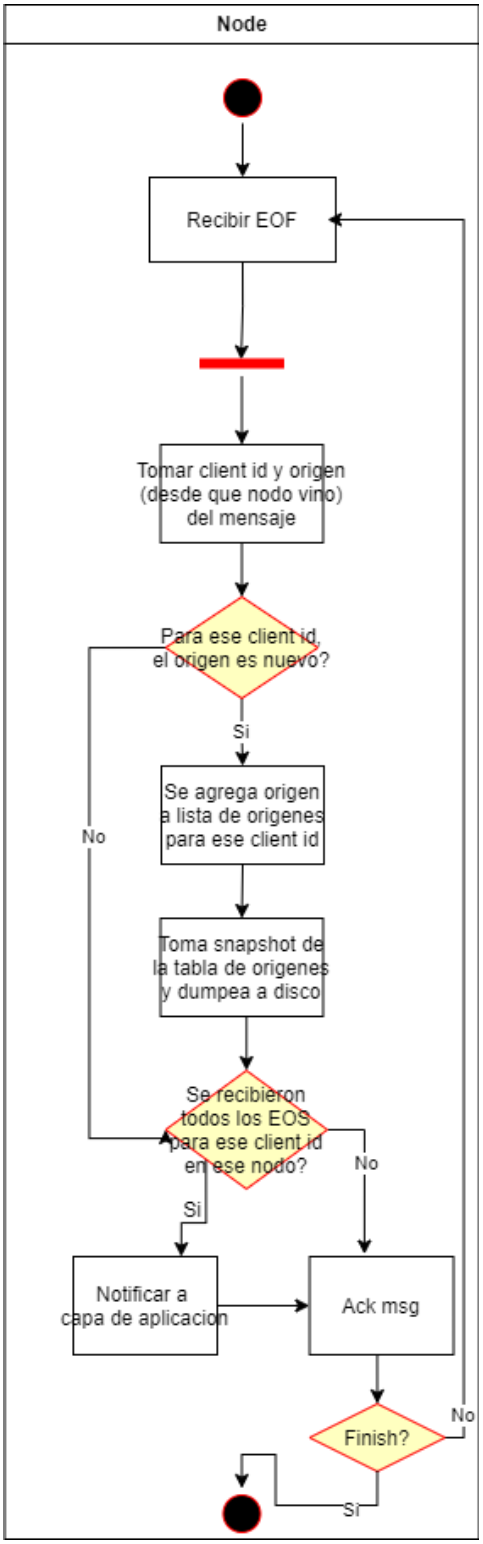

Figura 8.3: Diagrama de Logs para EOFs

Aqui se presenta el comportamiento de TODOS los nodos para loggear mensajes de EOFs. Esto sucede en la capa de comunicacion.

Una vez que se decide realizar un flush de la data en memoria a disco, necesitamos garantizar que el archivo donde estamos persistiendo la data no se corrompa durante la escritura, por ejemplo por la caida del nodo justo durante dicha escritura. Lo que hacemos entonces es escribir toda la data que vamos a persistir en un archivo temporal, distinto del archivo concreto (fijo) del Log. Una vez escrita la data en el archivo temporal, realizamos un rename de este archivo y le ponemos el nombre del archivo fijo, que sera justamente reeamplazado por el temporal. Lo importante de este rename es que es atomico, se hace o no se hace, por lo cual no existe la posibilidad de que el archivo quede corrupto.

Si el nodo se cayera antes de renombrar el archivo temporal, nos quedamos con el viejo archivo fijo y cuando el nodo vuelva a levantarse, simplemente no tendra la data que iba a persistirse (como tampoco hizo el ack, la data volvera a aparecer en la cola de rabbit). Si el nodo se cae despues de renombrar el archivo temporal, la data habra sido persistida, por lo cual sera leída al levantarse el nodo.

Es de destacar que este sistema de persistencia con renames tolera fallas hasta user space, no tolera fallas del sistema operativo.

8.3. Elección de lider

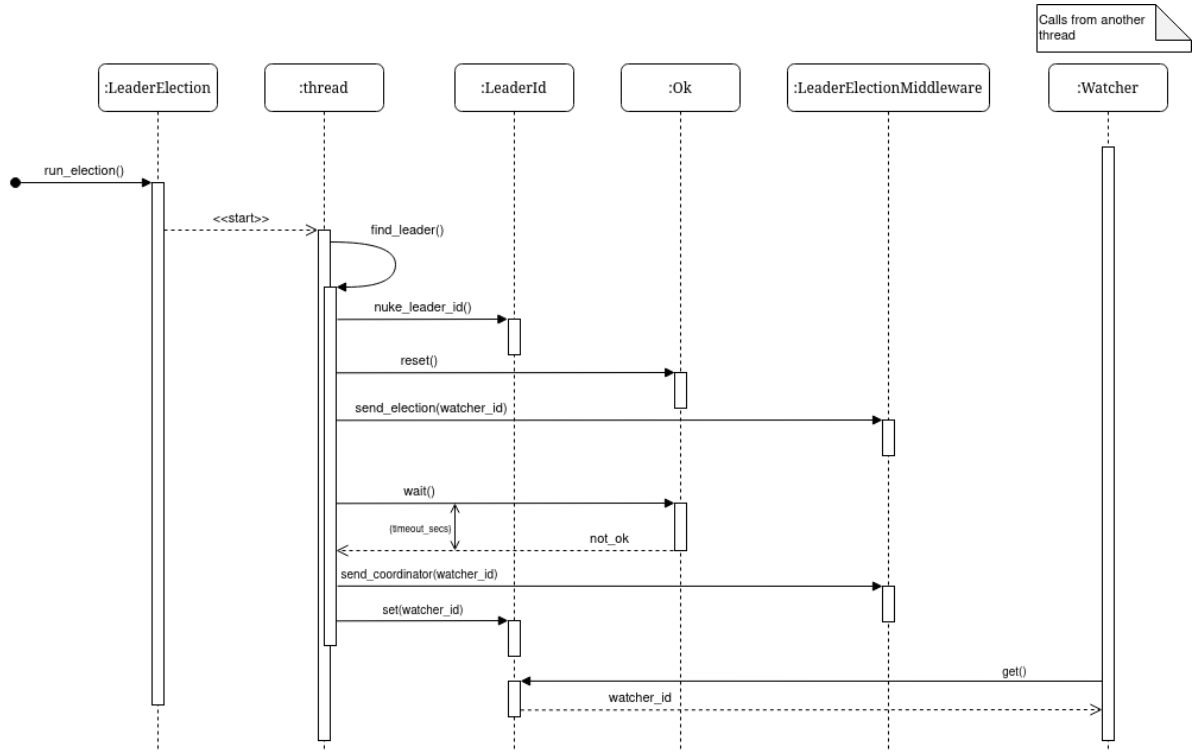

Figura 8.4: Proceso de elección de lider

En el proceso de elección de lider, cuando se genera un trigger de elección de lider, se lanza un hilo que se encargará de enviar el mensaje ELECTION a cada watcher con mayor id al propio (siguiendo el algoritmo de Bully). Este hilo luego esperará un tiempo a que al menos

alguien le responda. Esta espera la hace sobre una instancia de Ok (objeto que sincroniza justamente esta condición, haber recibido al menos una respuesta al mensaje ELECTION).

El sistema tolerará que se caigan todos excepto un Watcher. De caerse el último Watcher disponible en el mismo, el resto de los nodos no tendrán la posibilidad de ser recuperados en caso de fallas que incurran en caidas de servicio de los mismos. En este caso, deberá hacerse el restart de por lo menos un Watcher de manera manual a modo de recuperar al sistema. Este parámetro es configurable.

8.4. Hashing para routear a los agregadores

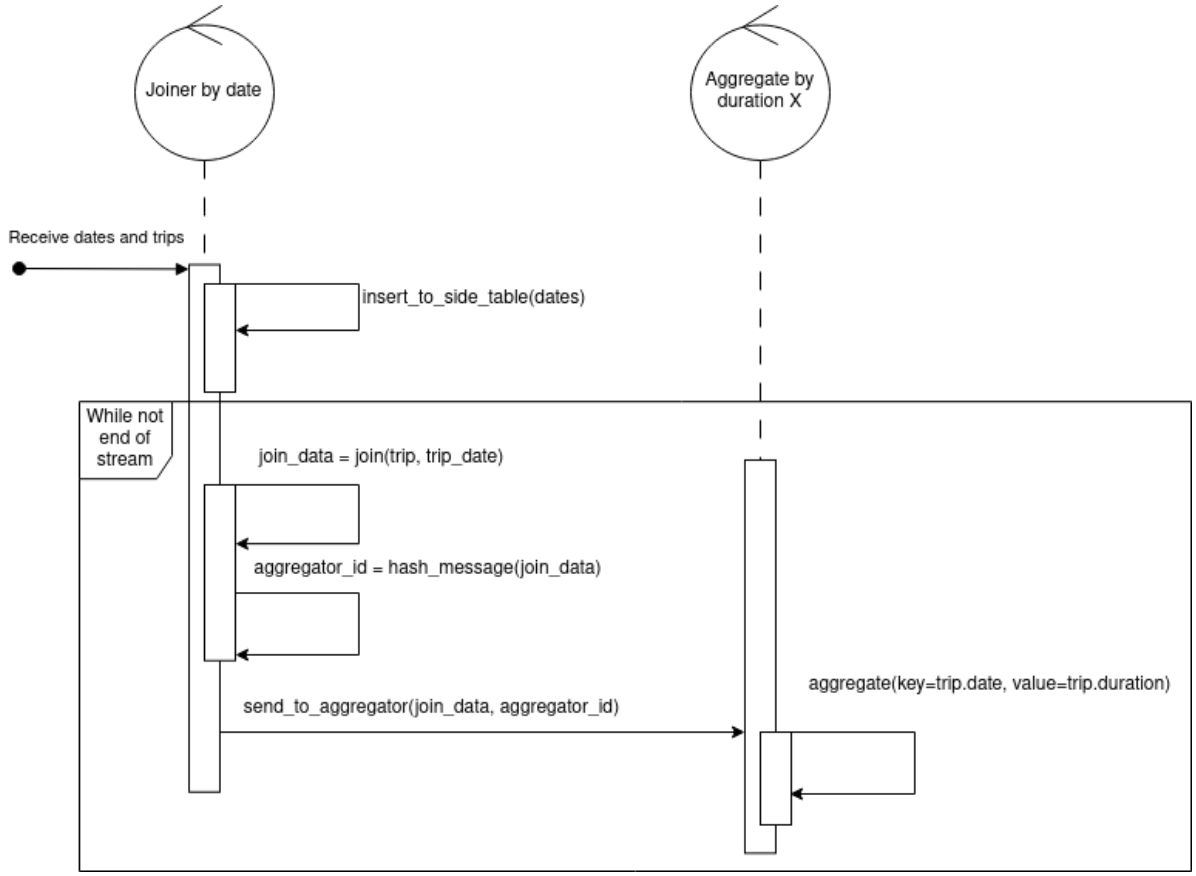

Figura 8.5: Diagrama de secuencia, hashing de claves para agregadores

Una de las caracteristicas del sistema es que se separa la etapa de junta de la etapa de agregación. Esto fue pensado asi de manera tal de evitar que la etapa de join sea un cuello de botella. Ahora, como en particular la etapa de agregación puede ser replicada, para estar agregando las particiones de datos bien, se necesita algún método que permita agregar estas particiones de manera tal de no generar datos inconsistentes entre corridas. Esto es posible gracias a que al momento de enviar la data joineada, los Joiners se encargan de hashear las claves para que los Aggregators reciban según ese hash. Esto permitirá además distribuir la carga de manera equitativa, contando con que el hash es consistente y tiene un output pseudo-random. Vemos el caso de la primera consulta, en donde el Joiner routea datos según la clave

de fecha. Los datos que sean de fechas iguales terminarán siendo agregados por el mismo Aggregator.

8.5. Watchdog y Healthchecks

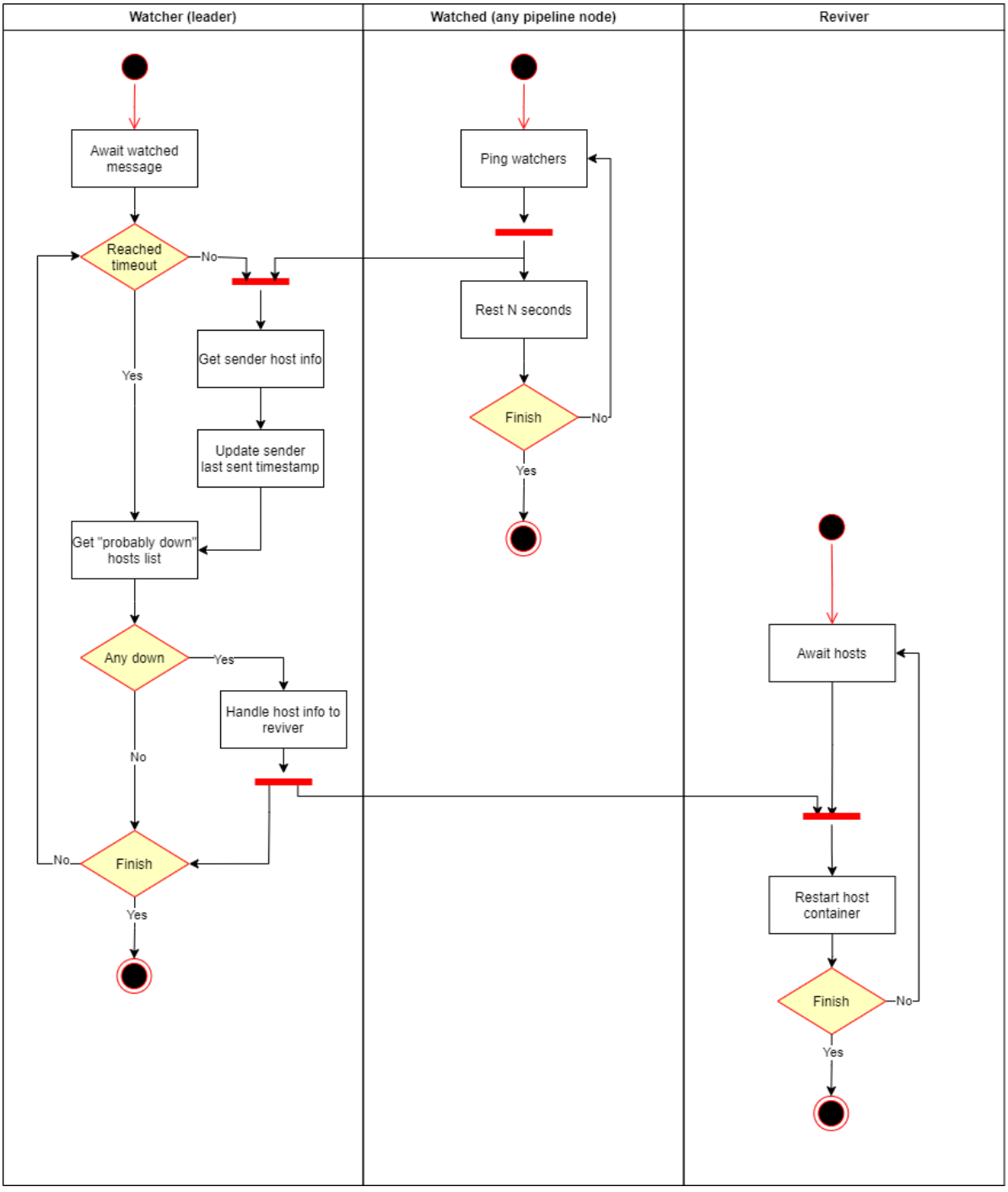

Figura 8.6: Flujo de mensajes entre nodo Watched y Watcher líder

En el sistema, los nodos Watched envían al líder un mensaje de un byte que representa un "ping". El líder Watcher espera mensajes sobre un socket udp durante una cantidad fija de tiempo. De recibir algun mensaje, registra el tiempo local para el nodo emisor. Independientemente de haber recibido o no un ping, checkea la lista de todos los hosts para ver por cada la última vez que enviaron un ping. Si alguno de esos no mandó un ping por cierta cantidad de tiempo (configurable), entonces se lo considera caido. Lógicamente que puede ser detectado como falso positivo, razón por la cual con una cantidad de tiempo razonable podría detectarse un nodo caido con alta probabilidad, pero al mismo tiempo podría demorar mucho el progreso de las queries. Una vez que el Watcher arma el listado de nodos caidos, se los envía a un worker para que los reinicie, utilizando la api de Docker.

La secuencia de mensajes entre estos objetos puede verse con mayor detalle en el siguiente diagrama.

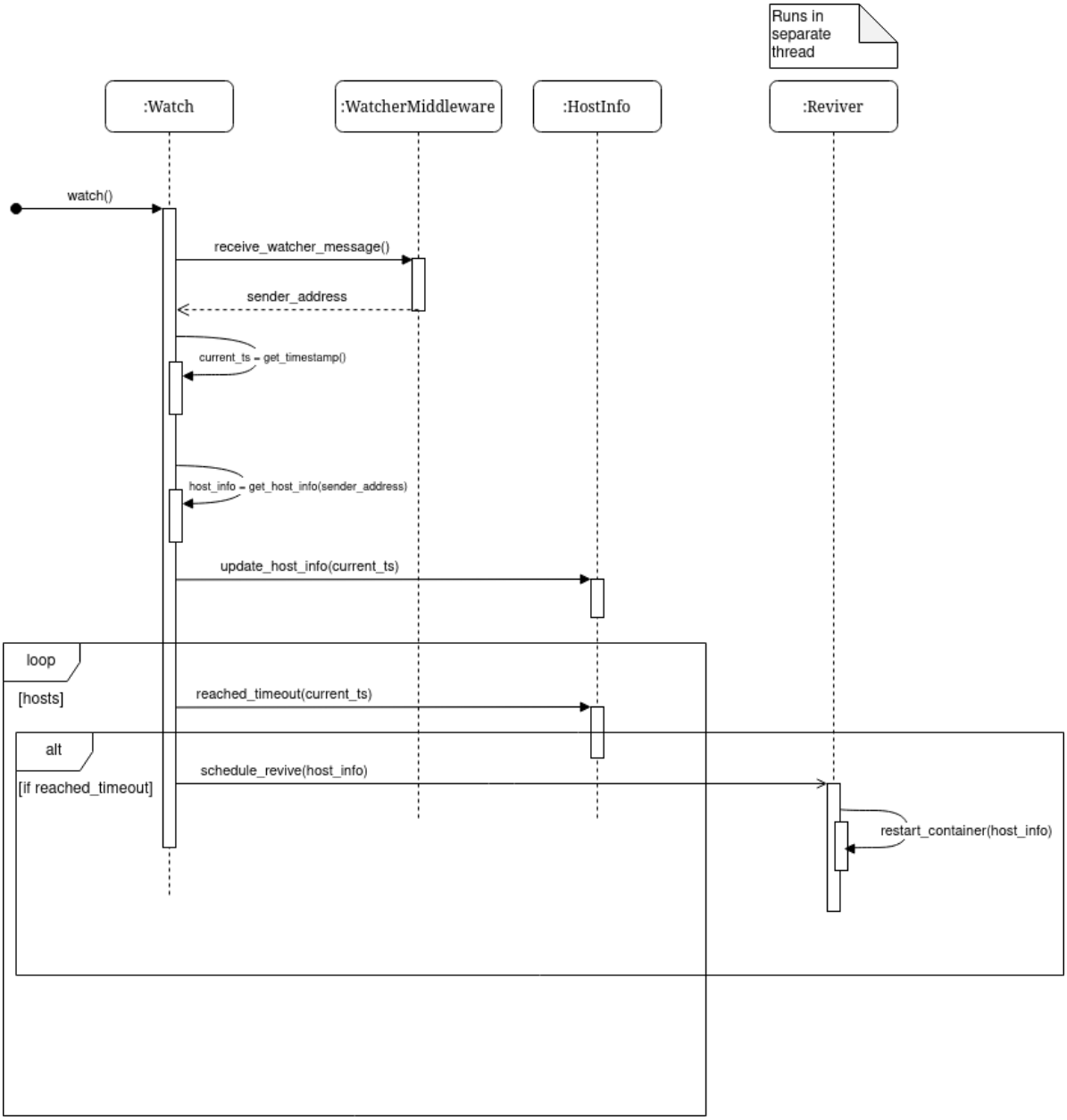

Figura 8.7: Interacción Watch-Reviver

8.6. Formato de los mensajes

A continuación se presenta un diagrama ilustrativo sobre los Mensajes en nuestro sistema, contando que campos lo conforman y para que sirve cada uno de ellos.

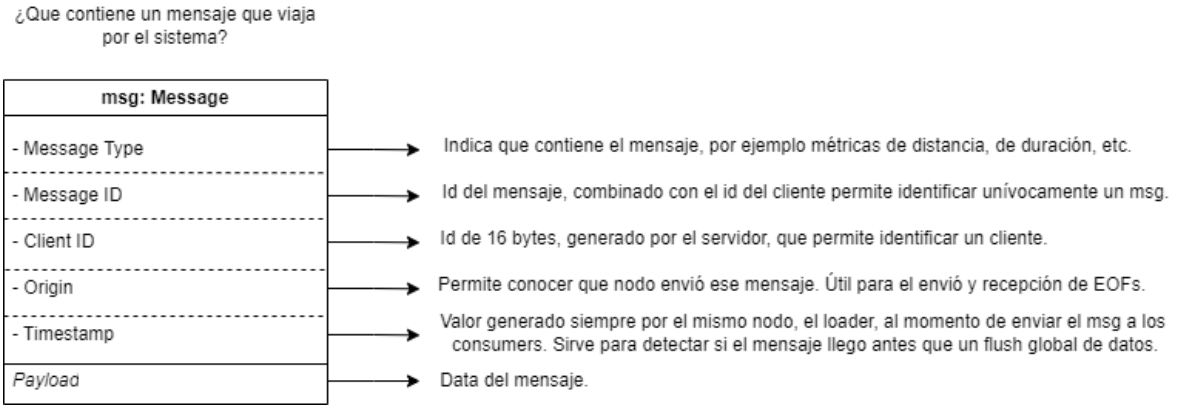

Figura 8.8: Mensajes en el sistema

Algo a destacar de los mensajes es la forma de distribución en el sistema. Supongamos el caso en que el usuario esta mandando trips. Nosotros tenemos filtros para, justamente, filtrar esos mensajes. Recordemos que tenemos varias replicas de un mismo filtro, cada una con su cola particular. Nostros queremos enviarle el mensaje unicamente a un filtro, entonces debemos decidir en cual cola de las N replicas insertar ese mensaje. Lo que hacemos es hashear el mensaje por su id considerando el numero de replicas, por lo cual, al ser el id siempre incremental (el id del mensaje NO es unico en el sistema, el id unico del mensaje es id del mensaje + id del cliente), se formaria un round robin para seleccionar a que filtro le llegara ese mensaje, y de esta forma balancear la carga de procesamiento.

Este mecanismo de hasheo se repite siempre que queramos propagar un dato en una unica cola, pero el tipo de nodo receptor este replicado, cada uno con una cola unica.

8.7. Formato de los Snapshots

Nuestro sistema, a la hora de loggear, no realiza Journaling sino que toma directamente un snapshot de la data que se desea persistir. A continuación una imagen ilustra que datos se guardan en los diferentes snapshots que tenemos.

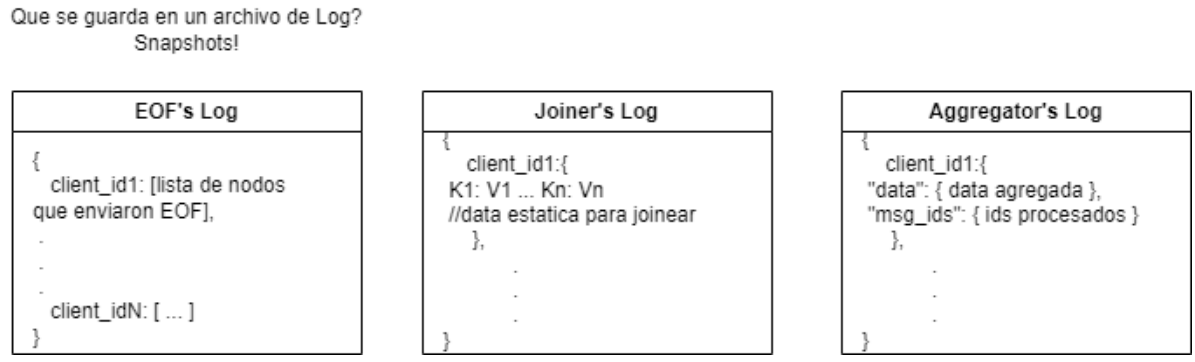

Figura 8.9: Mensajes en el sistema

El log de EOFs simplemente guarda, para cada client id, la lista de los origenes, es decir los nodos, que ya han enviado el EOF. Por ejemplo un filter que espera que 3 replicas de

weather consumer le envien el EOF almacenaria ["weather consumer 1", "weather consumer 2", "weather consumer 3"]. Esto se debe a la posibilidad de recibir EOFs duplicados. Todos los nodos del sistema tienen este log.

El log de los Joiners guarda para cada client id recibido la data estatica que usara para joinear. Guarda como clave el identificador que hara referencia a esa row en un trip, como data, los datos que seran joineados con ese trip.

Por ultimo, el log de los aggregators tiene para cada client id la data que ese aggregator ya fue consumiendo -agregando- y la lista de los messages ids ya procesados, esto ultimo a fin de evitar procesar mensajes duplicados, ya que aqui las operaciones no son idempotentes (como si lo son en un joiner). La data seria los resultados parciales de una query que va calculando para ese cliente.

8.8. Flush global de datos

El Loader es un nodo vital en el sistema ya que conforma la puerta de entrada al servidor, es quien recibe la data del cliente, se la envia a los consumers y quien recibe los resultados para tambien enviarselos al cliente. Este nodo no esta replicado, es una unica instancia y su caida es problematica porque se pierde la conexion con los clientes.

La decisión tomada respecto al loader es que en caso de caida, los clientes entienden que la subida/procesamiento de datos fallo y que deben intentar reconectarse y empezar a subir datos de nuevo. El problema de esto es que habra quedado viajando por el sistema data de los clientes que sera resubida (o no, pero en fin, data vieja), y de alguna forma esta data debe borrarse.

Lo que hace el loader entonces es, al momento de iniciar su ejecución, enviar via exchange a todos los nodos un timestamp, un numero global y unico que indica el momento en que inicio la ejecución del loader. Los nodos reciben y persisten este valor, y ademas borran sus logs ya que entienden que al haberse caido el loader, toda la data que almacenaron quedo vieja y es innecesaria.

Luego, a cada paquete que el loader envia a los consumers le pondra un timestamp, el cual sera leido por cada uno de los nodos al recibir dicho paquete (esto en la capa de comunicación), y luego comparado con el ultimo timestamp global que recibieron del loader. Si el timestamp del mensaje es mayor al global, el paquete corresponde a un cliente en funcionamiento y se procesa normalmente. Si el timestamp es menor al global, se considera que el mensaje es viejo y se descarta sin procesar.

En la sección de Known Issues se pueden leer las posibles formas de obtener un timestamp global y las ventajas e inconvenientes de cada una de estas formas.

De esta forma el sistema garantiza que los mensajes obsoletos se eliminen tan pronto como sea posible, sin seguir procesandose inutilmente.

9. DAG

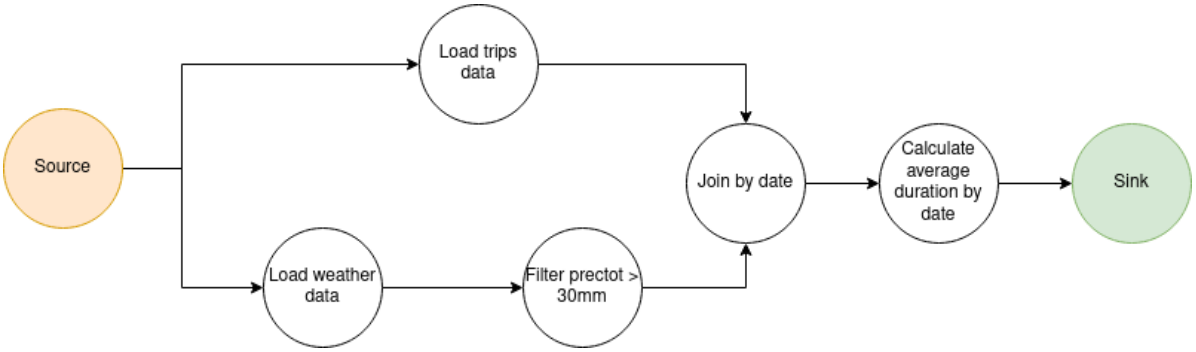

Figura 9.1: DAG, consulta de duración de viaje en dias lluviosos

En este DAG se puede apreciar como se da el flujo de información relevante a la primera consulta. Tanto la información de viajes como de clima fluye desde el source. Los datos del clima serán filtrados por su campo de precipitación (dando lugar a un Filter por ese campo como nodo) y llegarán a una etapa de Join, en donde se agruparán con los datos de viajes por fecha. A medida que se vayan agrupando, se calculará un rolling average de dicha duración de viaje, y con ese dato tendremos resuelta la consulta.

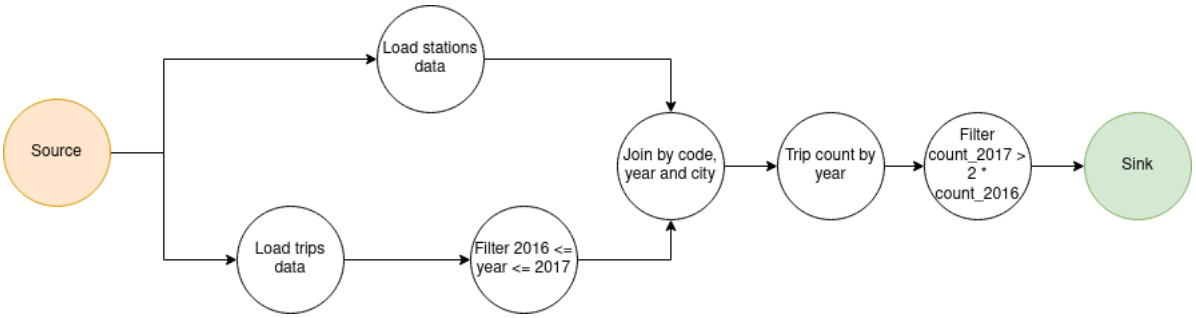

Figura 9.2: DAG que muestra el flujo de información para la consulta de viajes por año entre 2016 y 2017

En este caso, se consumiran datos tanto de estaciones como de viajes. Los datos de viajes serán filtrados por año, para luego ser agrupados con los datos de estaciones por el código de estacion de inicio, el año y la ciudad. A medida que los resultados de la junta se materialicen, serán delegados a la etapa de agregación, que en este caso tendrá un contador para 2016 como para 2017, y podrá entonces acumular esas cantidades para cada estacion de salida. Una vez agregados los datos, se filtrarán aquellos que hayan duplicado la cantidad de viajes en 2017, descartando aquellos que no hayan tenido algun viaje en 2016.

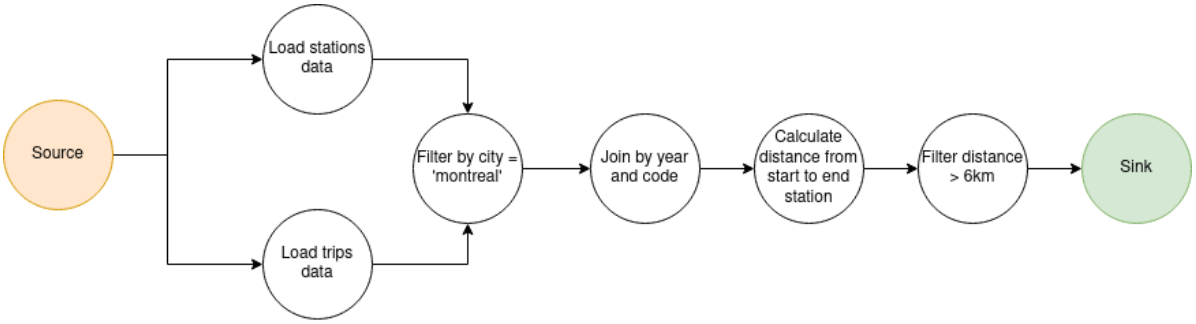

Figura 9.3: DAG que muestra el flujo de información para la consulta de estaciones de llegada con promedio de distancia recorrida mayor a 6km, para la ciudad de Montreal

De la misma manera que en el DAG anterior, se consumiran datos de viajes como de estaciones, que serán primero filtrados por ciudad (en este caso Montreal). Una vez filtrados, se agruparán por los códigos de estaciones y año (debe estar por la naturaleza de los datos) para entonces obtener coordenadas de origen y destino, además del nombre de la estación. Como paso siguiente, se calculará la distancia utilizando Haversine, para luego filtrar y quedarnos con aquellas estaciones cuya distancia promedio para llegar desde cualquier otra sea mayor a 6km.

10. Known issues

En algunos sistemas, puede que la función time.monotonic no sea consistente entre cores, y por ende, puede que desde distintos hilos se obtengan resultados no consistentes en cuanto a timestamps. Bike Rides Analyzer implementa entonces en el Loader un monitor de timestamps para cubrir estos casos, a costa de perder performance. Esto es de todas maneras configurable, se puede optar por utilizar la implementación del sistema operativo o la de Bike Rides Analyzer, a través de la variable booleana de configuracion TIME_MONOTONIC. De no garantizarse la consistencia, el sistema podría encontrarse con un caso borde en el que se flushean los datos y de manera concurrente se unen uno o mas clientes al sistema y empiezan a procesar información. Si bien se envía el mensaje de flush a los nodos previo a aceptar clientes, si el timestamp no garantiza orden del tipo happened-before, entonces podría darse el caso en que ciertos datos del cliente puedan ser desestimados. De todas maneras, la implementación propia garantiza el ordenamiento happened-before y puede considerarse un tradeoff entre performance y manejar este caso borde al configurar el sistema.