.UBAfiuba
FACULTAD DE INGENIERÍA

Bike Riders Analyzer

Alta Disponibilidad y Tolerancia a Fallos

[75.74] Sistemas Distribuidos I
1C 2023

Apellido y Nombre	Padrón	Email
Fernández Caruso Santiago Pablo	105267	sfernandezc@fi.uba.ar
Iragui Ignacio	105110	iiragui@fi.uba.ar

Índice

1. Introducción

El presente informe reune la documentacion y distintas vistas de arquitectura del trabajo practico. El objetivo del mismo es implementar un sistema multicomputing que sea capaz de procesar un dataset de viajes en bicicleta en diferentes ciudades del mundo, logrando obtener ciertas metricas de dichos viajes. Ademas, el sistema debe ser tolerante a fallos y brindar alta disponibilidad.

2. Vista Lógica

2.1. DAG

Se presenta el grafo acíclico dirigido el cual se utilizó para modelar las consultas que debe poder responder el sistema.

Figura 1: Direct Acyclic Graphs

2.2. Atomic Write

A continuación se presenta un pseudocódigo del guardado de archivos con seguridad de persistencia ante caídas.

```
1
2  def atomic_write(filename, data, encoding):
3      file_dir, file_name = split_file(filename)
4
5      temp_name = f"{file_dir}/temp.txt"
6      with open(temp_name, "w") as temp_file:
7          temp_file.write(data)
8
9      current_time = str(time.time())
10
11     new_file = f"{current_time}_{file_name}"
12     new_name = f"{file_dir}/{new_file}"
13
14     os.rename(temp_name, new_name)
15     for file in os.listdir(file_dir):
16         if file != new_file and file.find(file_name) > 0:
17             real_file = f"{file_dir}/{file}"
18             os.remove(real_file)
```

Listing 1: escritura de archivos

De esta manera uno siempre tendrá disponible una versión ya que no borramos los archivos "viejos" hasta que no se ejecuta el *rename* (en la linea 14). De caerse el nodo antes de dicha linea persistirá el estado anterior, un *timestamp* menor (calculado previamente en la linea 9). De caerse luego de esta linea en el peor de los casos habrá dos versiones guardadas y simplemente deberá escogerse aquella que tenga mayor *timestamp* en su nombre. Esto también es posible gracias a que el *rename* en *python* es una operación atómica [1], en caso contrario, podría caerse en el medio y dejarnos en un estado desconocido.

3. Vista Física

3.1. Diagrama de Robustez

En estos diagramas podemos ver los nodos del sistema y como se comunican entre si. La regla general es tomar mensajes de una cola, procesarlos, y enviarlos a otro nodo mediante otra cola.

También se puede ver cuales nodos son escalables y cuales no, buscamos hacer nodos "genéricos" que se puedan parametrizar mediante variables de entorno, los nodos que se pueden escalar son los $filters$, los $joiners$, el calculador de distancia, el modificador de fechas y el $parser$. Los $groupers$ son únicos ya que deben ir realizando una agregación cada vez que reciben un dato para poder calcular una métrica. También son únicos el $accepter$, el $statuscontroller$ y el $EofManager$.

Todas las comunicaciones se realizan con colas de $RabbitMQ$, algunas de estas colas son $workqueues$ y otras $publisher/subscriberqueues$.

Para facilitar la lectura de los diagramas decidimos dividirlos en distintos diagramas.

3.1.1. Diagrama General

En primer lugar vemos el diagrama general, se muestran los nodos principales que interactúan en todas las queries: *accepter*, *parsers* y el *controlstatus*.

El *accepter* se encarga de recibir todos los *requests* de los clientes, los *parsers* de parsear los datos enviados, y el *statuscontroller* es quien espera por la respuesta de cada query.

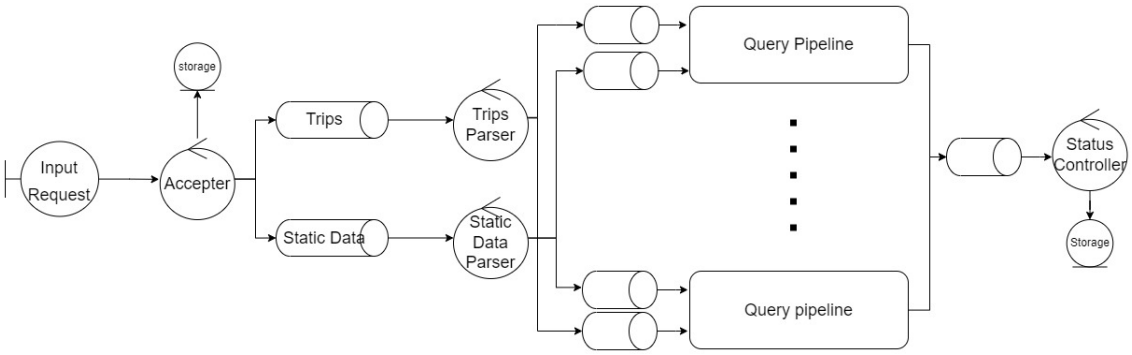

Figura 2: Diagrama General de Robustez

Se pueden agregar tantos *Query Pipelines* como sea necesario. La estructura general de los *Query Pipelines* es la que vemos en el siguiente diagrama:

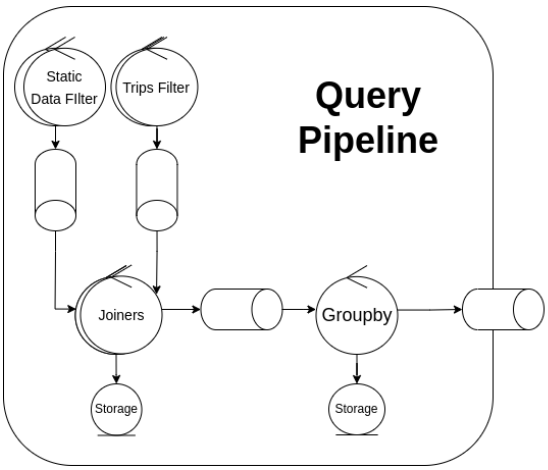

Figura 3: Query Pipeline base

Los *query pipelines* constan, por lo general, de un filtro para datos estáticos y otro para los viajes, que luego se unen en un *joiner* para finalmente ser agregados en el *groupby*. Sin embargo, de ser necesario se pueden agregar nodos extra en cualquier lugar del pipeline para poder realizar procesamientos que sean necesarios sobre los datos.

3.1.2. Diagrama Particular - Query3

Como se menciona en la sección anterior, los *pipelines* admiten agregar nodos *custom* en cualquier lugar del flujo.

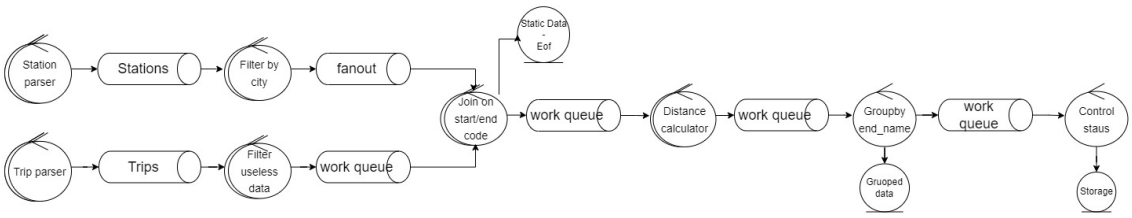

Figura 4: Query 3

Para el caso de la Query 3 vemos que respeta la estructura general pero agrega un nodo ***Distance calculator*** que se encarga de calcular distancias entre dos puntos.

3.1.3. Monitores & Eof Manager

Por ultimo hablaremos de nodos particulares, el *Eof Manager* y los Monitores. La diferencia de estos nodos con el resto es que son nodos de *"Control"*

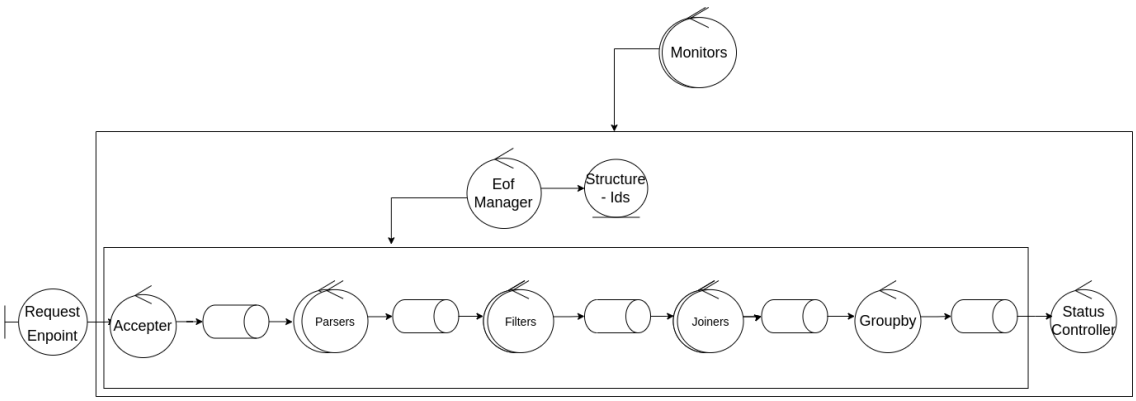

Figura 5: Monitores & Eof Manager

El *Eof Manager* se comunica con todos los nodos del sistema, exceptuando al *status controller* y los monitores, y su tarea es administrar y enviar los mensajes de *EOF* y *Clean* cuando sea necesario.

Los monitores se comunican con todos los nodos y su tarea es enviar y recibir *heartbeats* para chequear el estado del sistema, cuando detectan un nodo caído lo reinician utilizando *docker-in-docker*.
Hay 3 replicas de los monitores donde se elige un líder y es quien realizara la tarea de enviar y recibir heartbeats.

3.2. Diagrama de Despliegue

Para no repetir información que ya fue provista en el diagrama de robustez, el diagrama se despliegue se simplificó en uno mas genérico que busca mostrar los diferentes tipos de nodos y con cuales se comunican. Es un diagrama relativamente sencillo ya que todos los nodos se comunican

a través de *RabbitMQ* por lo que se puede desplegar cada nodo por separado, logrando un sistema *multicomputing*.

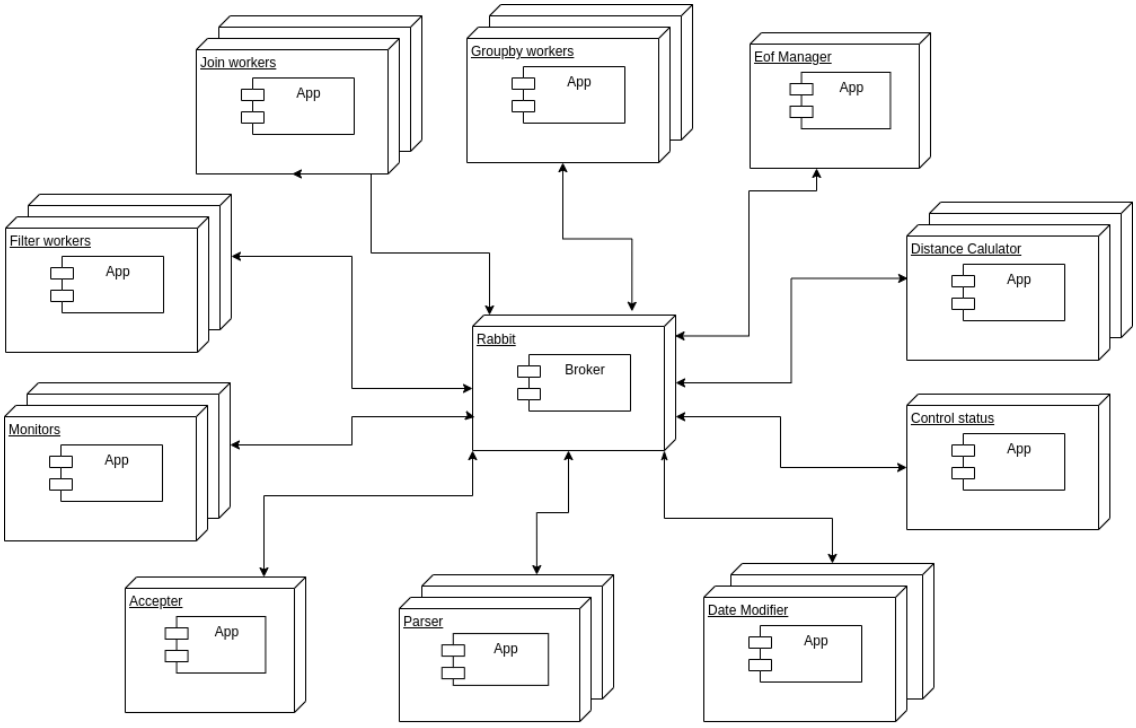

Figura 6: Diagrama de despliegue del sistema

4. Vista de Escenarios

4.1. Casos de Uso

El diagrama de casos de usos intenta representar el problema a resolver y como el usuario interactúa con el sistema.

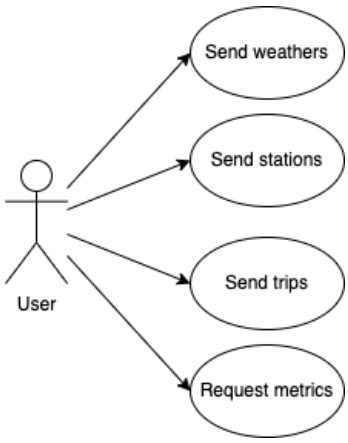

Figura 7: Diagrama de Casos de Uso

5. Vista de Procesos

5.1. Diagramas de Actividad

5.1.1. Query 2

A continuación se puede ver el diagrama de actividades correspondiente a la consulta numero 2: *Obtener los nombres de las estaciones que al menos duplicaron la cuantiad de viajes iniciados en ellas entre 2016 y 2017.*

Las lineas punteadas representan la comunicación mediante las colas de *RabbitMQ*. El diagrama omite ciertos nodos para facilitar la lectura.

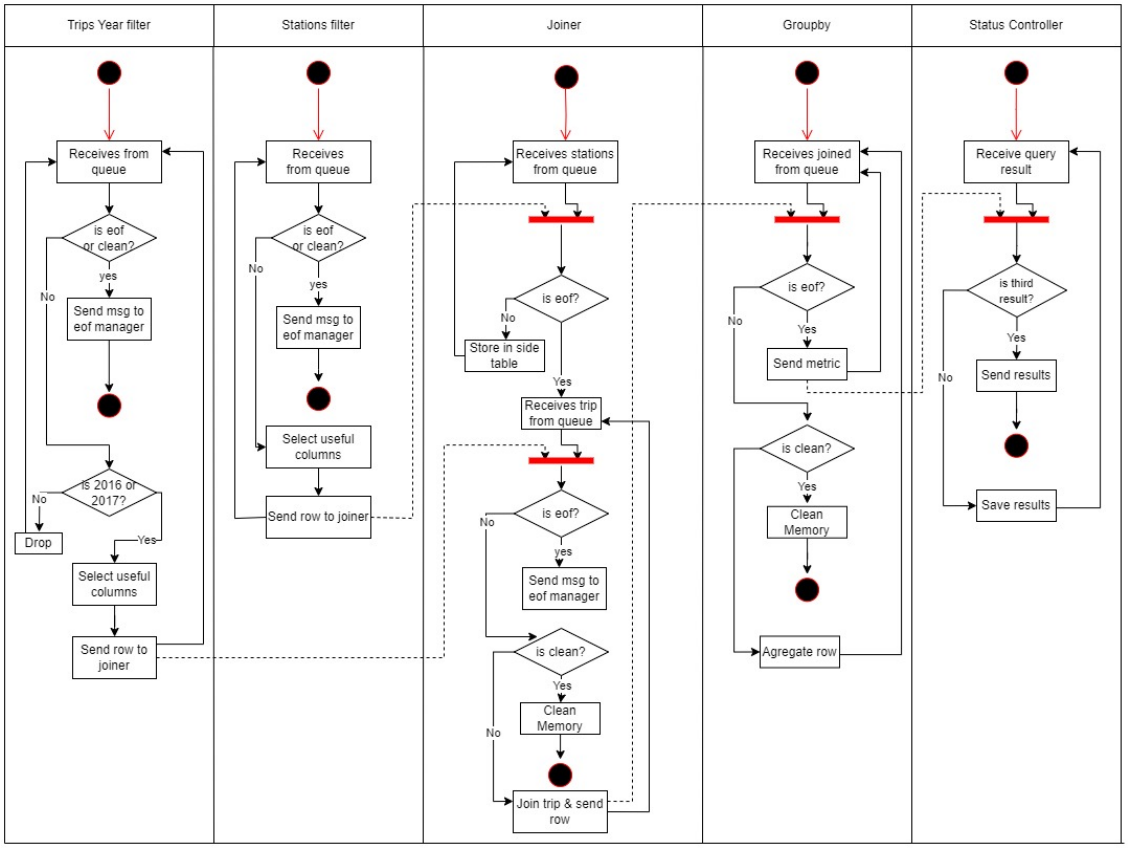

Figura 8: Diagrama de Actividad para consulta 2

Las demás consultas se resuelven de forma muy similar. Los mensajes de *EOF* se tratan utilizando un componente llamado *Eof Manager* el cual veremos mas detallado en los diagramas de secuencia.

En este caso decidimos representar solo el proceso asociado a un cliente. De forma que cuando se recibe un mensaje *clean* se corta el proceso pero en los nodos reales estos siguen corriendo ya que deben procesar otros clientes.

5.1.2. Elección de líder - Monitores

Para que el sistema sea tolerante a fallos decidimos implementar la clase ***Monitor***, encargada de detectar y levantar nodos caídos. La cantidad de monitores es parametrizable y de entre todas

las replicas se elige un líder que realizara las tareas mencionadas.

La elección de líder se lleva a cabo utilizando el algoritmo ***Bully***

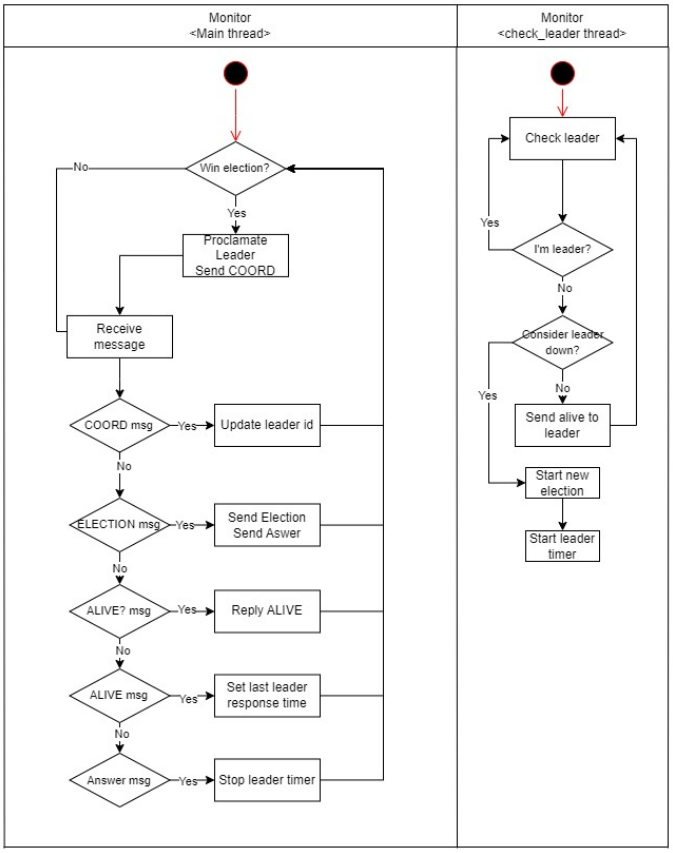

Figura 9: Monitores

En el diagrama se muestra la estructura básica de un monitor y cómo es el procesamiento. Hay un thread *main* escuchando y procesando mensajes constantemente, y otro thread *check_ leader* que se encarga de enviar mensajes ALIVE? al líder, y en caso de detectar que el líder está caído, inicia una nueva elección.

El thread *main* chequea si ganó la elección, de ser así, envía un mensaje COORDINATOR para anunciar su liderazgo. Luego, comienza a escuchar mensajes de sus pares. Ante un mensaje de COORDINATOR, se actualiza el ID del líder con el de quien envió el mensaje. Si el mensaje es de ELECTION, quiere decir que otro monitor con menor ID detectó al líder caído e inició una elección; en ese caso, envío el mensaje ELECTION a todas las réplicas más grandes y respondo con ANSWER a la que envió el ELECTION. A su vez, inicio el *timer* de elección de líder. Ante un mensaje de ALIVE?, que solo es recibido por el líder, responde con un ALIVE. Los no líderes recibirán el ALIVE del líder y actualizarán el último contacto que tuvieron con él. Cuando recibo un ANSWER, simplemente freno mi *timer* de elección de líder porque sé que hay una réplica mayor que yo en el sistema que está activa.

El *thread check_ leader* envía mensajes de ALIVE? al líder siempre y cuando haya otro líder. En caso de no encontrar un líder, inicia su *timer* de líder y comienza una nueva elección.

Ambos *threads* comparten ciertas variables, como los *timers* de elección de líder, para poder çomunicarsez saber cómo actuar.

5.1.3. Hearbeats

Una vez elegido el líder, es importante entender como este se encarga de saber que nodos están vivos y cuales no para poder decidir cuales levantar nuevamente. Para esto planteamos un diagrama de actividad que nos muestra los dos hilos que tiene un monitor relacionados al *heartbeat* y un tercer nodo que podría ser cualquier otro nodo comunicándose con el monitor.

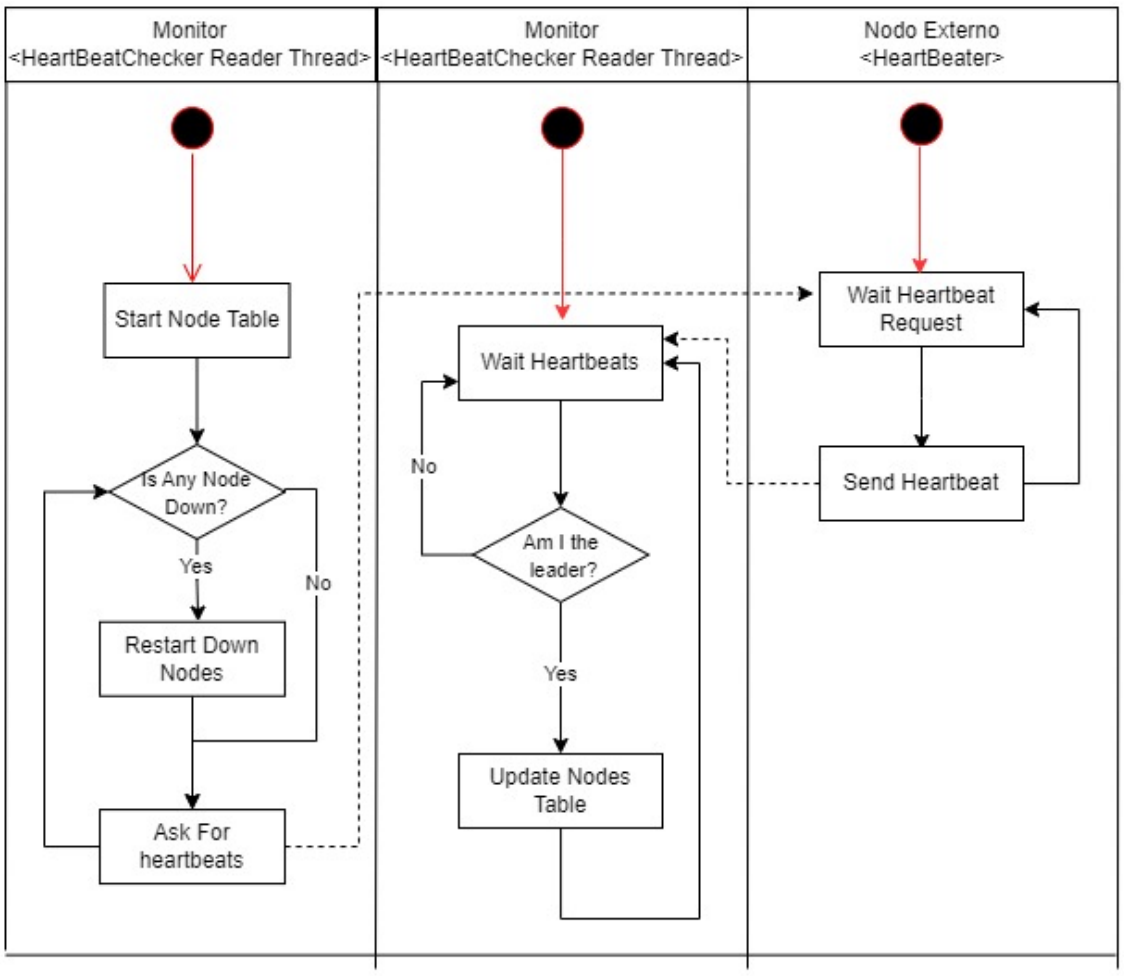

Figura 10: Envío de Heartbeats

En el diagrama se ve cómo solo aquel monitor que sea líder leerá los mensajes enviados. A su vez, se ve cómo el monitor es el que pide el envío de heartbeats, permitiéndole a los otros nodos no tener que implementar nuevos threads solo para el *heartbeat*, ya que esto se puede conectar a la misma conexión usada para otras actividades.

5.2. Diagramas de Secuencia

5.2.1. Manejo de múltiples clientes

El sistema es capaz de soportar múltiples clientes en paralelo.
El funcionamiento básico es que el hilo principal se encuentra constantemente escuchando por nuevas conexiones, cuando un nuevo cliente se quiere conectar se chequea que la cantidad de clientes en el sistema no sea mayor a una constante definida y en caso afirmativo es aceptado.

Una vez aceptada, la nueva conexión se ejecuta en un nuevo thread donde se mantiene la comunicación con el cliente hasta que finaliza.

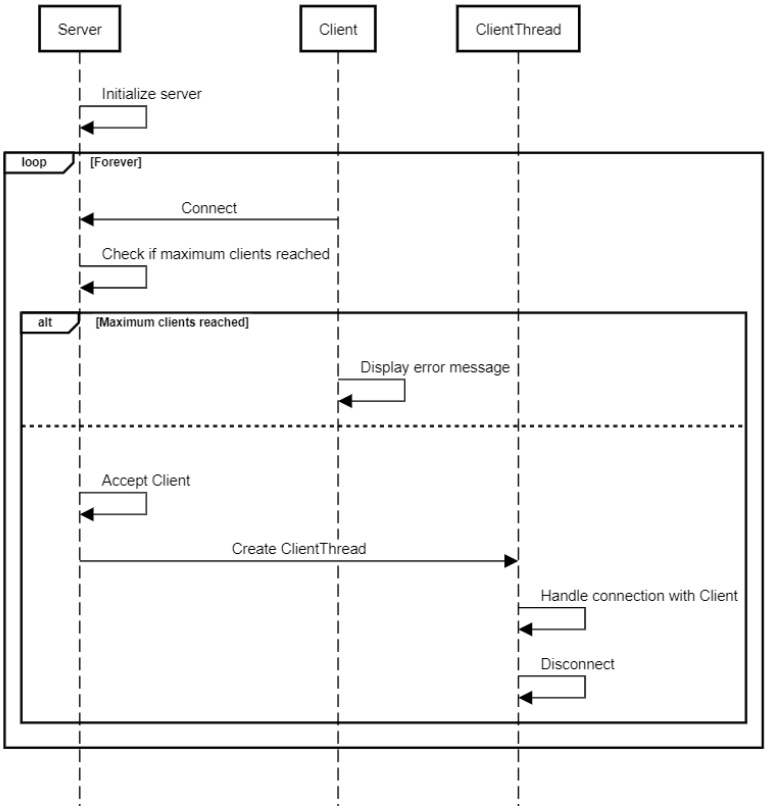

Figura 11: Manejo de múltiples clientes

5.2.2. Comunicación cliente servidor

La comunicación entre cliente y servidor es sumamente sencilla, consta de enviar los datos estáticos y luego los viajes.

El protocolo internamente maneja el envío de mensajes y *acknowledges*.

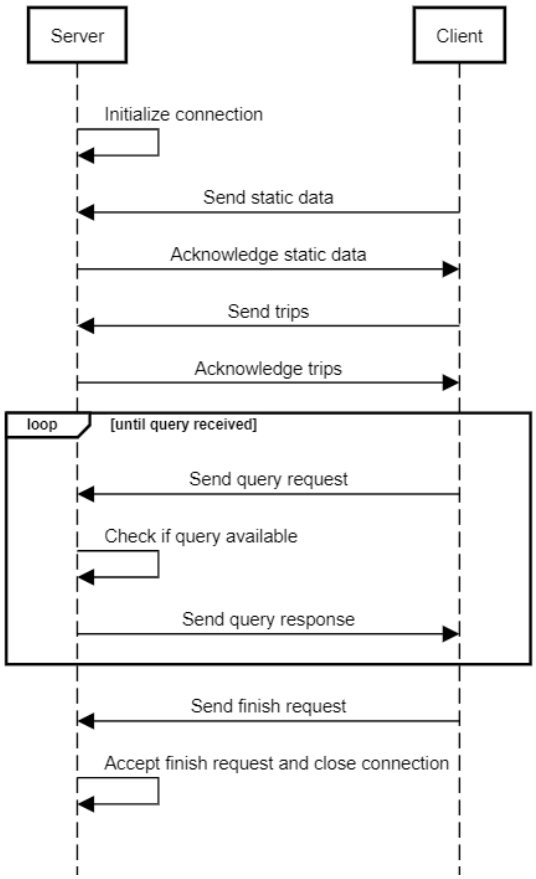

Figura 12: Comunicación cliente servidor

Una vez que el cliente envía todos los datos puede comenzar a pedirle las respuestas al sistema, si ya se procesaron todos los datos se enviaran las queries o de lo contrario un False, de esta forma el cliente puede pedir los resultados cuando le sea conveniente.

Por ultimo es necesario un mensaje de finish por parte del cliente para poder cerrar la conexión correctamente.

5.2.3. Envió de Datos

Se muestra un diagrama de secuencia de como es el proceso desde que el cliente envía los datos hasta que llegan al *joiner*.

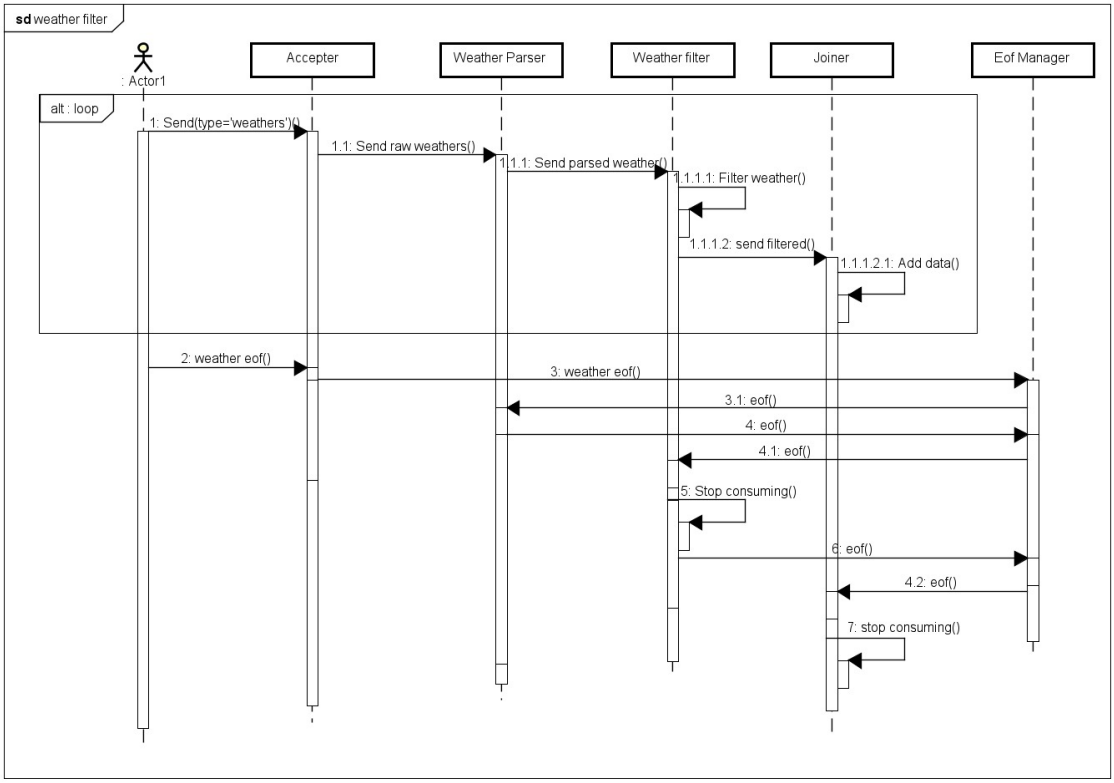

Figura 13: Secuencia filtrado de weathers

Se puede ver como todos los mensajes de *EOF* pasan primero por el *Eof Manager* antes de ir al componente correspondiente.

5.2.4. Recorrido de los trip

Por otro lado tenemos el recorrido de los *trips*, que es muy similar al anterior.

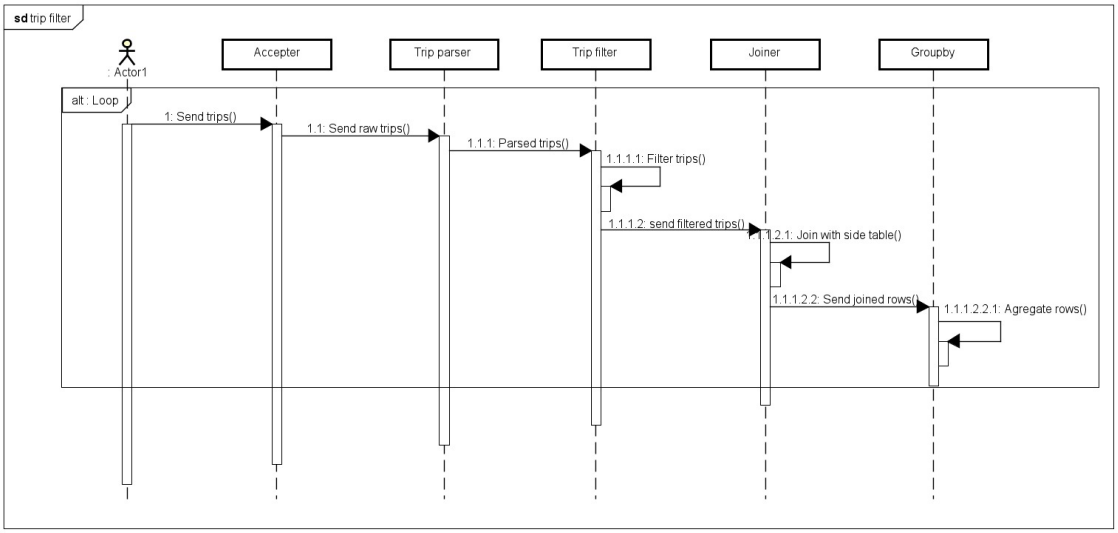

Figura 14: Secuencia filtrado de trips

5.2.5. Funcionamiento del Eof Manager

A continuación un diagrama del funcionamiento del *Eof Manager* más en detalle. En este caso tenemos un componente de *parser*, dos de filtrado y un *joiner*, por lo tanto el parser le envía datos a ambos *filters*, estos procesan los viajes y los envían al *joiner*.

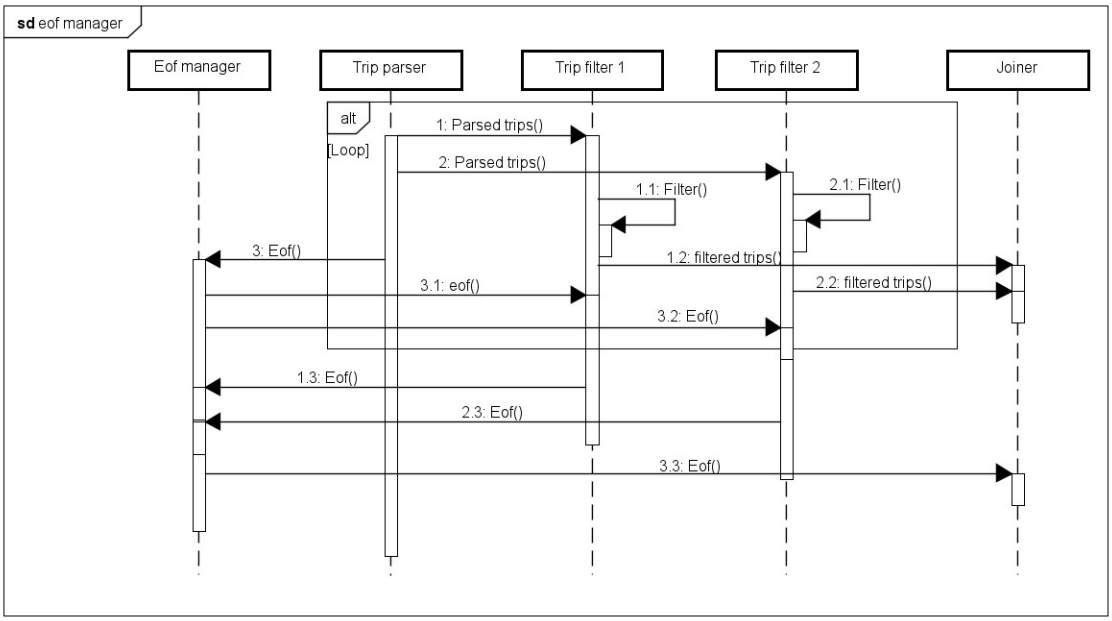

Figura 15: Secuencia funcionamiento de Eof Manager

Una vez que al *Trip Parser* le llega un EOF, deja de consumir mensajes y envía el EOF al *Eof Manager*, el Eof manager sabe que solo hay un *Trip Parser* por lo tanto sabe que ya puede enviar los EOF a la siguiente etapa. El *Eof Manager* también conoce que hay dos *filters* por lo cual encola dos mensajes de EOF en dicha cola. Cuando los *filter* procesan el EOF dejan de escuchar y envían EOF al *Eof Manager*.

Como el *Eof Manager* sabe que hay dos *trip filter*, espera por dos mensajes de EOF antes de enviar el EOF al joiner.

5.2.6. Persistencia de datos en nodos

Por último planteamos la estructura básica del guardado de memoria en nodos para persistir información ante la caída del mismo.

La idea fundamental es no enviar el acknowledge de mensajes hasta que estos hayan sido guardados en disco y establecemos que se debe bajar en ciertos puntos críticos, es decir, al recibir un mensaje de clean o uno de eof, o si ya tenemos más de $MESSAGE\_BATCH$ mensajes sin confirmar la recepción.

A su vez como esto puede generar que nosotros guardemos mensajes que luego recibiremos nuevamente por la falta de su acknowledge, por eso implementamos un filtrado de mensajes duplicados.

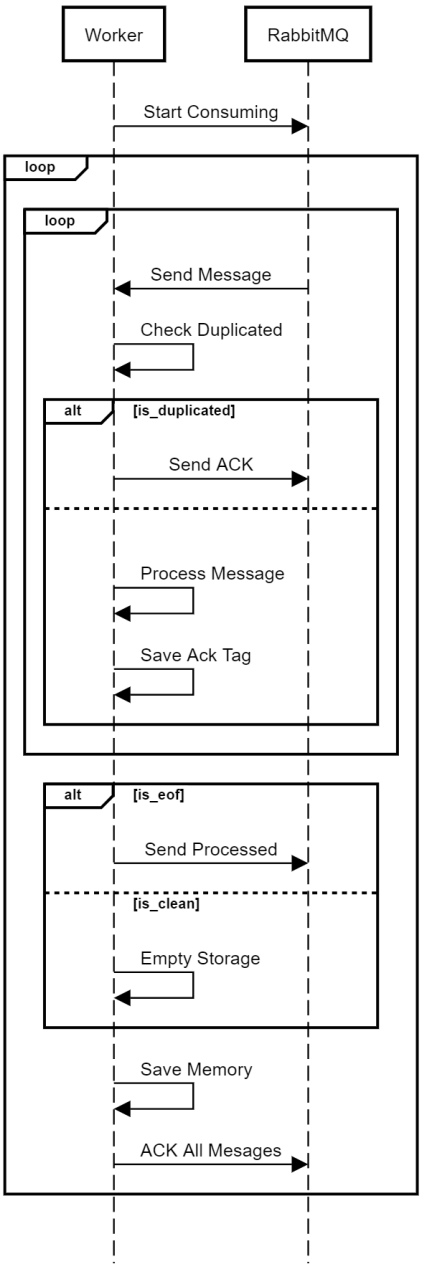

Figura 16: Persistencia de datos en nodos

6. Vista de Desarrollo

6.1. Diagrama de Paquetes

Para dicho diagrama se modelaron los diferentes módulos del sistema y como se relacionan unos con otros. Cada servicio es un paquete diferente que de despliega en un nodo diferente pero todos dependen del paquete *common* que contiene la lógica del *middleware* de comunicación, de *heartbeats* y de bajada a disco para aquellos que lo necesiten.

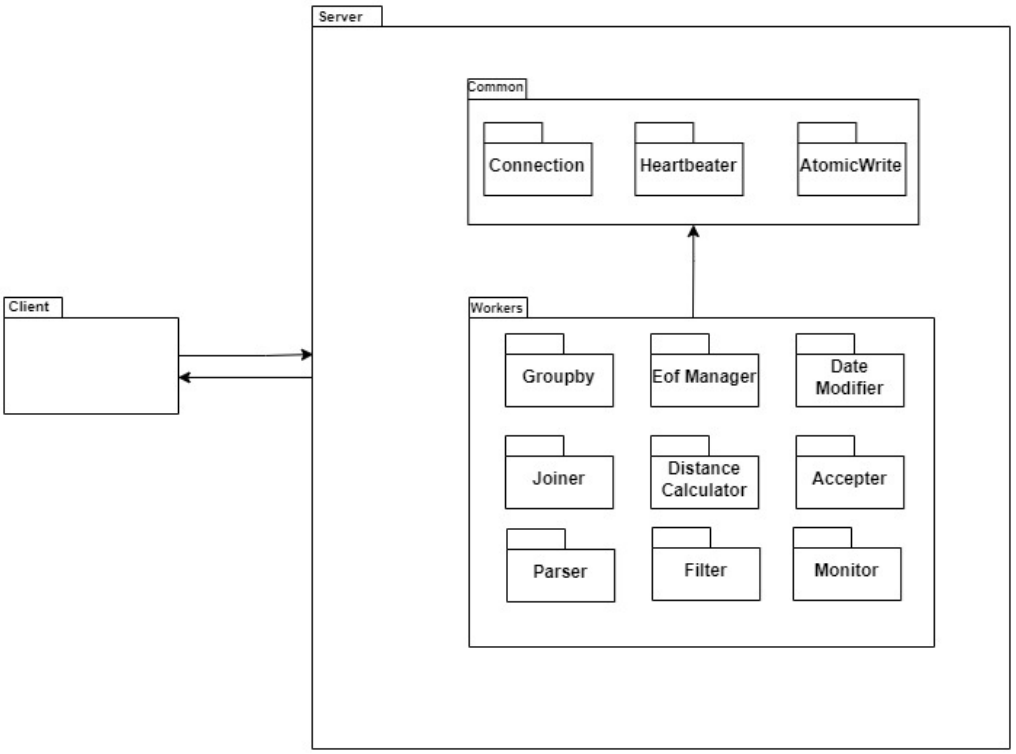

Figura 17: Diagrama de paquetes

7. Bibliografía

Referencias

[1] Documentación oficial de Python: https://docs.python.org/3/library/os.html#os.replace.