

Trabajo Práctico
Tolerancia a Fallas
Informe

Sistemas Distribuidos 1

75.74

2C 2023

Integrantes

- Batastini, Franco Nicolás - 103775
- Grassano, Bruno - 103855

Índice

Introducción.....	3
Objetivo.....	3
Supuestos.....	4
Alcance.....	5
Archivos de entrada.....	6
Aeropuertos.....	6
Vuelos.....	7
Tecnologías de implementación.....	7
Ejecución.....	8
Scripts.....	9
Script Compose Generator.....	9
Script Container Killer.....	9
Protocolos de comunicación.....	10
Mensajes.....	10
Cliente con Servidor.....	13
Con RabbitMQ.....	14
Healthchecker.....	14
Escenarios.....	16
Diagrama de Casos de Uso.....	16
Vista Física.....	17
Grafo Acíclico Dirigido (DAG).....	17
Diagrama de Robustez.....	18
Diagrama de Despliegue.....	20
Vista de Procesos.....	21
Diagrama de Actividades/Escenarios Concurrentes.....	21
Diagrama de Secuencia.....	22
Vista de Desarrollo.....	26
Diagrama de Paquetes.....	26
Vista Lógica.....	28
Diagrama de Clases.....	28
Concurrencia/Paralelismo.....	31
Finalización.....	32
Checkpointing.....	33
Detección de duplicados.....	36
Pendientes.....	38

Introducción

El presente documento reúne la documentación de la solución al trabajo práctico de tolerancia a fallas de la materia Sistemas Distribuidos I. Este consiste en realizar un sistema llamado Flights Optimizer.

El Flights Optimizer es un sistema distribuido preparado para realizar consultas sobre los datos que provengan de registros de vuelos que cumplan con ciertas condiciones. Los resultados de estas consultas son devueltas al cliente para que el mismo pueda tomar las decisiones que considere oportunas.

Objetivo

El objetivo del presente trabajo práctico es realizar un sistema distribuido escalable y tolerante a fallas capaz de analizar 6 meses de registros de precios de vuelos de avión para proponer mejoras en la oferta a clientes.

Este sistema debe de responder a las siguientes consultas:

- ID, trayecto, precio y escalas de vuelos de 3 escalas o más.
- ID, trayecto y distancia total de vuelos cuya distancia total sea mayor a cuatro veces la distancia directa entre puntos origen-destino.
- ID, trayecto, escalas y duración de los 2 vuelos más rápidos para todo trayecto con algún vuelo de 3 escalas o más.
- El precio avg y max por trayecto de los vuelos con precio mayor a la media general de precios.

Supuestos

Para la realización del trabajo se tomaron los siguientes supuestos:

- La entrada de datos es simulada a través de un cliente al cual se le indica un archivo csv de vuelos, y otro archivo csv de aeropuertos para realizar algunas operaciones. Los archivos son leídos y enviados al sistema para ser procesados. Una vez enviados el cliente pedirá los resultados y los irá imprimiendo por pantalla a medida que lleguen.
- Los archivos a ser utilizados cumplen con ciertos requisitos, tales como nombres o formatos. Se deja especificado más adelante en el documento.
- Para los cálculos de distancia se tomó el método de cálculo Haversine.
- El sistema no es elástico, no se van a agregar nuevos nodos una vez que ya empezó a ejecutarse.
- Los clientes no pierden la conexión.
- Los nodos se pueden caer pero eventualmente se van a levantar.
- RabbitMQ no puede caerse.

Alcance

Los hitos que comprenden el alcance del presente trabajo práctico son los enumerados en esta sección. Se fue reduciendo a tareas y haciendo seguimiento de las mismas a través de GitHub Projects.

- **Adecuación del sistema actual para múltiples clientes.** El trabajo de escalabilidad si bien soportaba que un cliente pueda continuar una vez finalizado el actual, no tenía soporte para que haya varios concurrentes. Se deben adecuar los servicios actuales para darle soporte a esta funcionalidad.
- **Realizar al sistema tolerante a fallas.** Este ítem incluye cinco grandes tareas:
 - **Desarrollo de Servicio de Health Check y Heartbeat.** Para detectar las caídas de los servicios se deben de empezar a notificar a los Health Checker (nuevo servicio a desarrollar) mediante Heartbeats. Estos servicios elegirán un líder (Mediante mecanismo bully) el cual será el encargado de levantar a los servicios caídos.
 - **Detección de duplicados.** Para asegurar el correcto funcionamiento del sistema, el mismo debe ser capaz de detectar duplicados y así preservar la validez de los resultados provistos.
 - **Replicación de servicios.** En el trabajo anterior había servicios donde se centralizaba la información, por ejemplo los savers para los resultados finales, estos deben de poder replicarse para que el sistema siga funcionando si se cae alguno de ellos.
 - **Persistencia de datos relevantes.** Para ser tolerante a fallas, si un servicio se cae, debe de poder recuperar su estado anterior al ser reiniciado. Es por esto que estos datos deben de ser persistidos mediante algún algoritmo, en nuestro caso decidimos utilizar checkpointing.
 - **Continuación de la ejecución del cliente ante la caída del servidor.** Si se cae el servidor, el cliente debe de poder seguir ejecutando su consulta cuando se levante nuevamente.

Archivos de entrada

Para utilizar el cliente implementado, se debe de proveer ciertos archivos que deben de cumplir con los siguientes formatos

Aeropuertos

Este archivo es para tener las ubicaciones de donde se encuentra geográficamente un aeropuerto y así realizar cálculos de distancia en la consulta 3. Debe de ser de formato *csv* separado por ‘;’ (punto y coma).

Tiene que tener como mínimo las columnas:

- *Airport Code*: Código de 3 dígitos del aeropuerto - String
- *Latitude*: Latitud del aeropuerto - Float
- *Longitude*: Longitud del aeropuerto - Float

Se toma de base el archivo ubicado en [Airports Opendatahub \(kaggle.com\)](https://www.kaggle.com/datasets/airports)

Vuelos

Este archivo contiene la información de los vuelos a analizar. Debe de ser formato *csv* separado por ‘,’ (coma)

Tiene que tener como mínimo las columnas:

- *legId*: El ID del vuelo - String - Columna 0 del CSV
- *startingAirport*: Código de 3 dígitos del aeropuerto inicial - String - Columna 3 del CSV
- *destinationAirport*: Código de 3 dígitos del aeropuerto destino - String - Columna 4 del CSV
- *travelDuration*: Duración del vuelo - String con formato *PT2H29M* - Columna 6 del CSV
- *totalFare*: Costo del vuelo - Float - Columna 12 del CSV
- *totalTravelDistance*: Distancia total recorrida en millas - Float - Columna 14 del CSV
- *segmentsArrivalAirportCode*: Los aeropuertos (codigo de 3 digitos) a los cuales se llega, separados por ‘||’ (doble pipe) - String - Columna 19
- *segmentsAirlineName*: Nombre de aerolíneas por cada segmento, separados por ‘||’ (doble pipe) - String - Columna 21

Se toma de base el archivo ubicado en [Flight Prices \(kaggle.com\)](https://www.kaggle.com/datasets/flight-prices) y [Flight Prices 2M \(kaggle.com\)](https://www.kaggle.com/datasets/flight-prices-2m)

Tecnologías de implementación

Para la realización del trabajo se tomaron las siguientes decisiones respecto a que tecnología usar en la implementación.

- Entre los lenguajes de programación permitidos para realizar el trabajo práctico, se optó por realizar los servicios en Go debido a la buena performance y manejo de *goroutines* que tiene.
- En el trabajo se tuvo que comunicar servicios a través de un motor de colas. Decidimos utilizar RabbitMQ debido a la facilidad en su uso.
- Para el armado de scripts, tales como un generador de docker-compose o finalizador de tareas, decidimos utilizar Python.

Ejecución

Se provee un `docker-compose-dev.yaml` que tiene todas las configuraciones requeridas para ejecutar el sistema (Si se quiere personalizar ver la sección Script Compose Generator) y un Makefile para simplificar la ejecución de los comandos. Se puede ejecutar como `make <target>`

El Makefile tiene las acciones:

- `docker-image`: Bildea las imágenes a ser utilizadas tanto en el servidor como en el cliente. Este target es utilizado por `docker-compose-up`
- `docker-compose-up`: Inicializa el ambiente de desarrollo (bildear docker images del servidor y cliente, inicializar la red a utilizar por docker, etc.) y arranca los containers de las aplicaciones que componen el proyecto.
- `docker-compose-down`: Realiza un `docker-compose stop` para detener los containers asociados al compose y luego realiza un `docker-compose down` para destruir todos los recursos asociados al proyecto que fueron inicializados
- `docker-compose-logs`: Permite ver los logs actuales del proyecto.
- `test`: Ejecuta los tests de la aplicación
- `build`: Realiza el build de los servicios localmente. Es necesario Go 1.21 por lo menos

Una cuestión a tener en cuenta, es que se deberán de configurar los archivos a utilizar por el cliente. Por defecto el `docker-compose` los busca de la carpeta `/data`, pero es posible modificar el `docker-compose` para que los busque en otro directorio.

Scripts

Se proveen una serie de scripts adicionales para facilitar ciertas acciones

Script Compose Generator

Este script permite generar un docker compose a partir de un template ya definido. Fue implementado con Python y Jinja.

Ejecución: `python compose_generator.py`

Una vez ejecutado el script va a pedir la cantidad de contenedores de cada servicio. Se busca que facilite el proceso de configurar el sistema al estar escalando o replicando.

La salida se va a encontrar en la carpeta `generated`.

Script Container Killer

Para facilitar las pruebas de estar finalizando los servicios con SIGKILL, armamos este script que les envía la señal.

Ejecución: `python killer.py`

Al ejecutarlo pide los nombres de los servicios y hace el llamado a la API de docker.

Protocolos de comunicación

Dado que vamos a estar comunicando diferentes aplicaciones, resulta necesaria la definición de protocolos. Decidimos utilizar un protocolo binario siguiendo un formato similar a TLV (Tipo Largo Valor) en la comunicación entre cliente, servidor, healthchecker y con RabbitMQ.

Mensajes

Nuestros mensajes se puede representar de la siguiente forma:

```
type Message struct {
    MessageType int
    ClientId     string
    messageId    uint
    RowId        uint16
    DynMaps      []DynamicMap
}
```

Donde `MessageType` es un entero que toma los siguientes valores:

- `Airports = 0` - Indica que el mensaje está transmitiendo información del archivo de aeropuertos.
- `EOFAirports = 1` - Indica que se terminó el archivo de aeropuertos
- `FlightRows = 2` - Indica que el mensaje está transmitiendo información del archivo de vuelos, ya sean datos para procesar o resultados al final
- `EOFFlightRows = 3` - Indica que se terminó el archivo de vuelos
- `GetResults = 4` - Indica que se están pidiendo los resultados
- `Later = 5` - Indica que se tiene que esperar por los resultados
- `EOFGetter = 6` - Indica que se terminó de enviar los resultados
- `FinalAvg = 7` - Indica que se calculó el promedio final en la consulta 4
- `HeartBeat = 8` - Indica que es un mensaje de heartbeat para señalar que el servicio está vivo.
- `EofAck = 9` - Indica que el servidor recibió y pasó a RabbitMQ el EOF de un cliente.

`ClientId` es un UUID como string que representa a un cliente del sistema.

`MessageId` es un entero para representar el id de un mensaje, es un id secuencial.

`RowId` es otro entero que representa un id secundario a nivel fila.

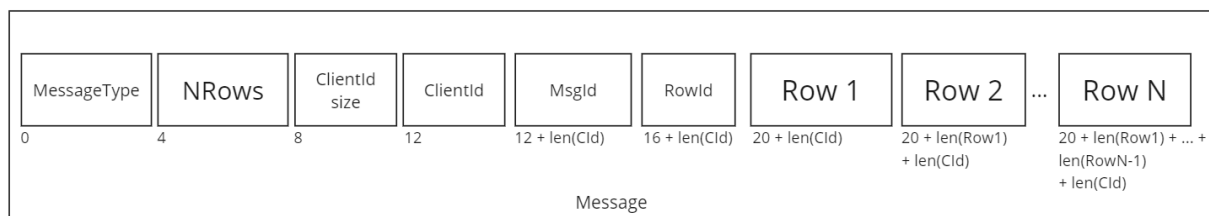
Donde `DynMaps` es un array dinámico de mapas. Estos mapas contienen la información que se va a estar transmitiendo en nuestro sistema, por ejemplo los datos de los archivos y cálculos adicionales.

Al ser un array de mapas, se abre la posibilidad a realizar envíos de mensajes donde tengan chunks o batches. La cantidad de mapas a enviar es configurable en las correspondientes aplicaciones que pueden iniciar el stream. Como valor por defecto se tomó 300.

Serialización del mensaje

Debido a que podemos tener múltiples mapas en nuestro mensaje, y que a su vez el mapa puede tener varios ítems clave-valor, es necesario plantear un mecanismo versátil para serializar y deserializar el mensaje que nos dé la suficiente flexibilidad para manejar los diferentes tipos de datos que tenemos.

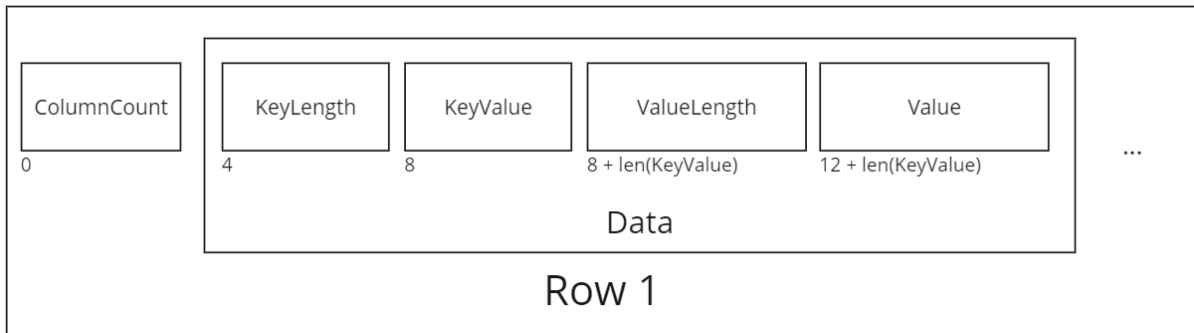
El mensaje entero en bytes se va a representar de la siguiente manera:



- En la imagen se puede ver que primero se serializa el tipo de mensaje, en nuestro caso el entero a 4 bytes en formato Big Endian.
- Le sigue otro entero de 4 bytes en Big Endian que tiene la cantidad de filas de datos (nuestros mapas). En caso de que no haya datos será cero.
- Le sigue el tamaño del ID del cliente junto con el ID. Este es un UUID. Se deja el tamaño variable para tener más flexibilidad si se quiere cambiar.
- Le siguen el `MessageId` y `RowId`, ambos son enteros de 4 bytes en Big Endian.
- Después vienen las filas de datos.

Notar los valores de offset de cada campo, durante la serialización hay que llevar un registro de cada movimiento dentro de los bytes para que el mensaje tenga sentido.

Una fila de datos, se representa de la siguiente manera:



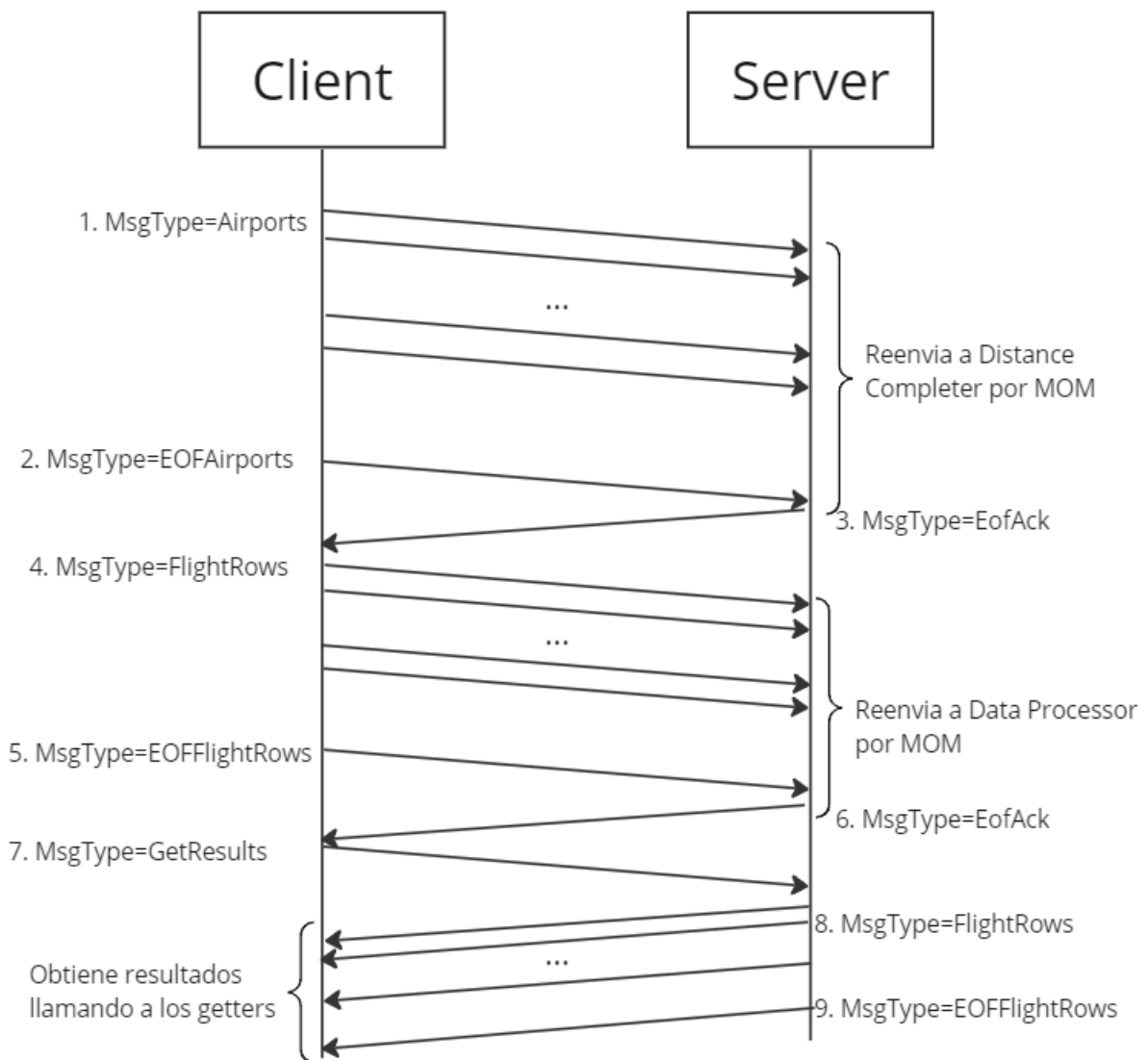
- Primero viene la cantidad de columnas (Data) que tiene la fila. Se va a estar repitiendo esta cantidad de veces la estructura Data
- KeyLength es el largo que tiene la clave de nuestro mapa dinámico. Es un entero de 4 bytes en Big Endian
- KeyValue es la clave del dato, tiene el largo de KeyLength. Es un String encodeado en UTF-8
- ValueLength es el largo del valor. Es un entero de 4 bytes en Big Endian
- Value es el dato. Se envía y maneja como bytes en el mapa. En caso de requerir el dato, depende de la aplicación pedirlo en el formato correcto. Por ejemplo, si el valor es un número entero, se pide al mapa la columna como `GetAsInt(column)` y se deserializa interpretandolo como Big Endian, si es un Float como Float de 32 bits y String como UTF-8 - *Nota: Al leer del archivo se obtiene todo en formato String, estos datos son convertidos/casteados al tipo de dato que corresponda para insertarlo en el mapa.*

Cliente con Servidor

Para comunicar la estructura de los mensajes por los sockets de nuestro sistema, es necesario enviar primero la cantidad de bytes a leer. Esto debe de ir en 4 bytes en formato Big Endian.

Recibida la cantidad de bytes a leer del socket, se procede a leer esa cantidad de bytes del socket para luego deserializar a un mensaje.

El flujo de mensajes queda de la siguiente forma:



Nota: El servidor es el que llama a los getters

En caso de que el cliente pierda conexión con el servidor, se reconectará y enviará el último mensaje enviado nuevamente para asegurarse que fue recibido y procesado correctamente.

Con RabbitMQ

La comunicación por colas con RabbitMQ se simplificó debido a que no se tiene que estar comunicando a Rabbit cuanto se va a enviar. En este caso solamente serializamos el mensaje al enviar y se deserializa cuando se recibe.

La particularidad de esta comunicación es que el ACK (o NACK) de un mensaje a RabbitMQ está diferido, se hace luego de enviado el mensaje a la siguiente etapa y de un checkpoint.

Healthchecker

Servicios con Health Checker

Todos los servicios del sistema envían a un Health Checker un heartbeat de forma periódica (cada 5 segundos) para que estos sepan que están activos. Esta comunicación es mediante TCP para asegurarnos el correcto envío del mensaje.

El heartbeat enviado contiene el nombre del servicio notificador dentro del DynMap (estructura explicada anteriormente) en una columna “*name*”.

Elección de líder de los Health Checkers

Debido a que buscamos que el sistema sea tolerante a fallas, si solo tuviésemos un Health Checker y este se cae el sistema dejaría de funcionar eventualmente, es por esto que este servicio debe de poder replicarse.

En esta replicación, debemos asegurarnos que solamente uno de los Health Checkers reinicia el servicio que no responde. Para esto, implementamos un algoritmo de líder de tipo Bully.

Esta elección tiene como protocolo de transporte UDP y el siguiente tipo de mensaje de 2 bytes.

```
type UDPPacket struct {  
    PacketType uint8  
    NodeID      uint8  
}
```

El primer byte es el tipo de mensaje y el segundo el id del Health Checker. Los tipos de mensaje son:

- `ACK = 0` Para indicar la recepción de un mensaje.
- `Election = 1` Para señalar que se inicia una elección.
- `Coordinator = 2` Para indicar que hay un nuevo líder.
- `HealthCheck = 3` Usado por las réplicas del líder para asegurarse que este sigue activo y no hay que iniciar una elección.

Si alguna replica no recibe el ACK del líder cierta cantidad de veces al enviar el HealthCheck iniciará una nueva elección.

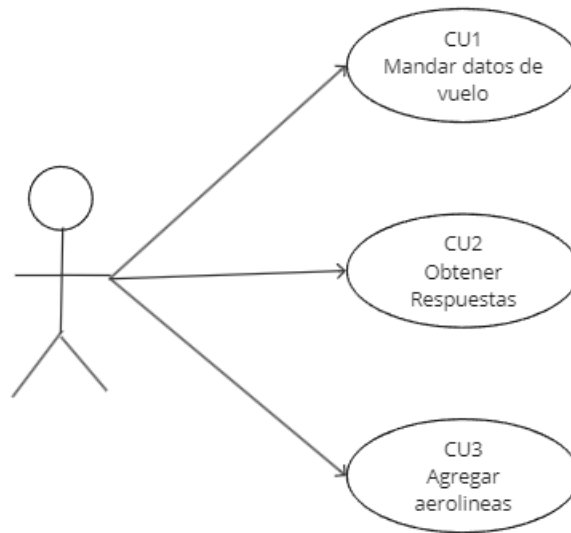
Health Checker con servicios

Si el Health Checker líder detecta que un servicio no responde por más de un tiempo configurado en segundos (este chequeo periodico de los servicios es también configurable), lo reinicia mediante la API de *docker-in-docker* y le da un tiempo para que se levante nuevamente.

Escenarios

Diagrama de Casos de Uso

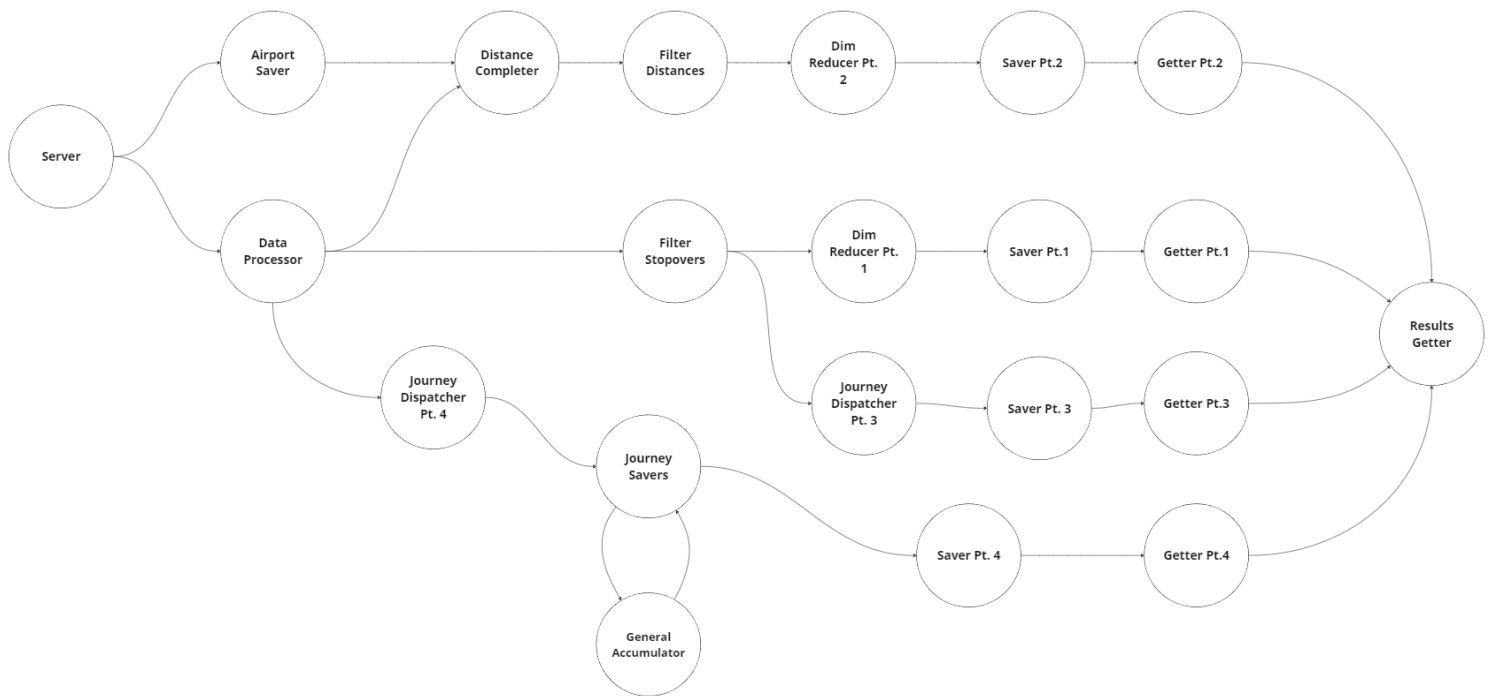
Nuestro sistema tiene los siguientes casos de uso.




- CU1 Mandar datos de vuelo: Es el envío de las filas de vuelos para su procesamiento.
- CU2 Obtener respuestas: Es la obtención de los resultados del procesamiento de las filas.
- CU3 Agregar aerolíneas: Es el envío de los datos de los aeropuertos al sistema para su consulta posterior durante el procesamiento.

Vista Física

Grafo Acíclico Dirigido (DAG)



Se puede ver en mejor calidad en  DAG.png

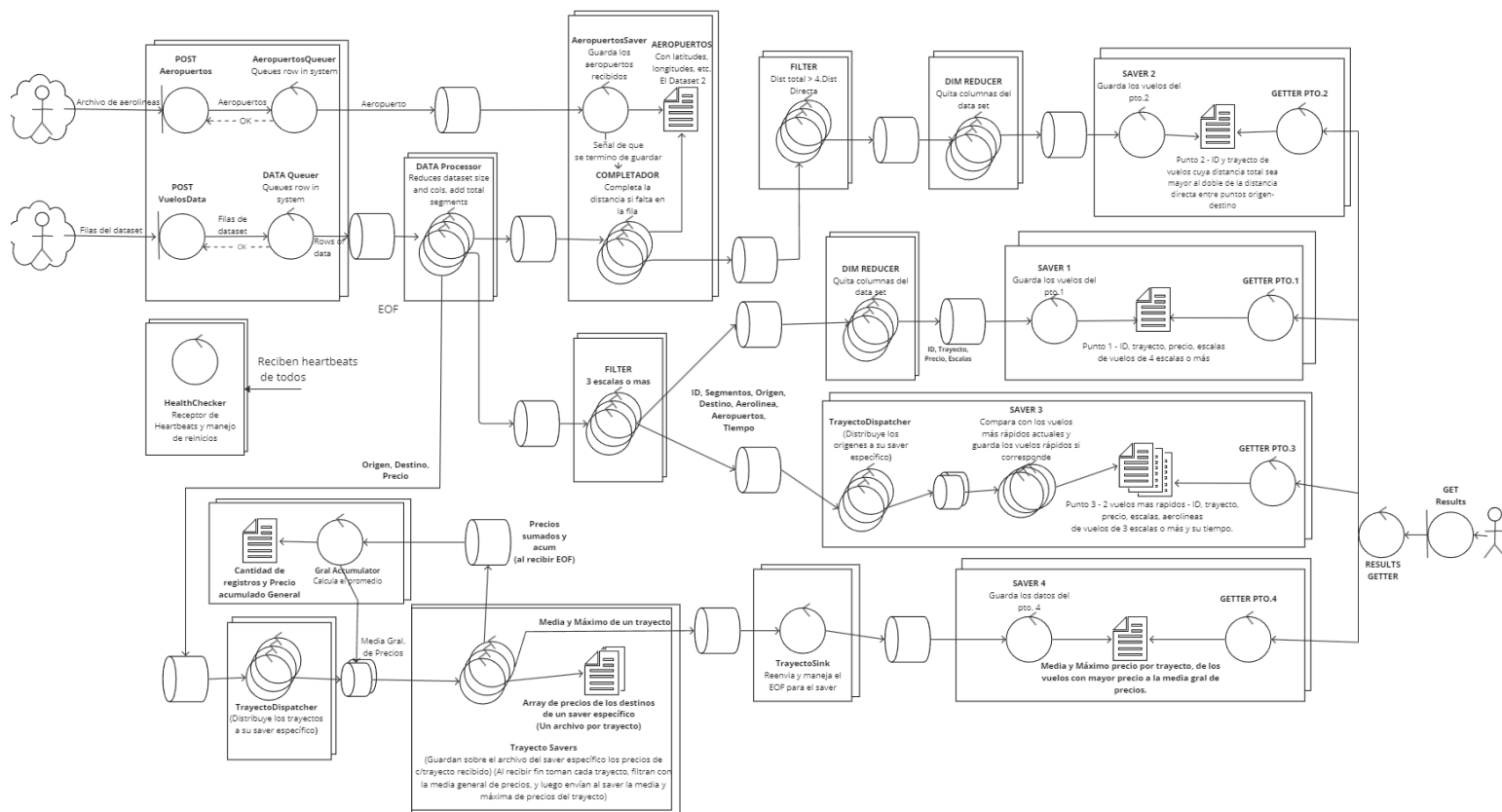
En la figura podemos ver el DAG del sistema. Analizamos sus nodos:


- **Server:** Recibe los datos del cliente y los reenvía al resto del sistema. Es el punto de entrada.
- **Airport Saver:** Guarda los datos del archivo de aeropuertos en el servicio completador de distancias.
- **Data Processor:** Hace un procesamiento inicial de los datos. Por ejemplo elimina columnas que no se usen si se le mandan, calcula cantidad de escalas, y arma las rutas completas. Estos mensajes los distribuye al resto del sistema.
- **Distance Completer:** Completa la distancia directa y total de los vuelos. Esta distancia va a ser usada como filtro más adelante.
- **Filter Distances:** Filtra filas dependiendo de la distancia.
- **Dim Reducer (Pt 1 y 2):** Reduce la cantidad de columnas que se reciben a solamente las necesarias. Este paso es para que los Savers tengan solo lo necesario para guardar.
- **Filter Stopovers:** Filtra filas dependiendo de la cantidad de escalas.
- **Journey Dispatcher (Pt 3 y 4):** Redirige datos a controladores particulares de un trayecto. Esto lo hace tomando un hash de origen y destino.

Trabajo Práctico Escalabilidad

- Journey Savers: Van sumando, acumulando, y guardando los datos de cada trayecto para ser procesados cuando se tenga el promedio final.
- General Accumulator: Realiza el cálculo del promedio cuando se recibió que se proceso la última fila. Si bien en el grafo se ve que se tiene un loop, debido a la secuencialidad de los mensajes no se puede entrar en un deadlock.
- Saver (Pt 1, 2, 3 y 4): Guardan los resultados de las consultas.
- Getter (Pt 1, 2, 3 y 4): Recuperan los resultados de las consultas.
- Results Getter: Obtiene todos los resultados de las consultas. Se encuentra en el servidor.

Diagrama de Robustez

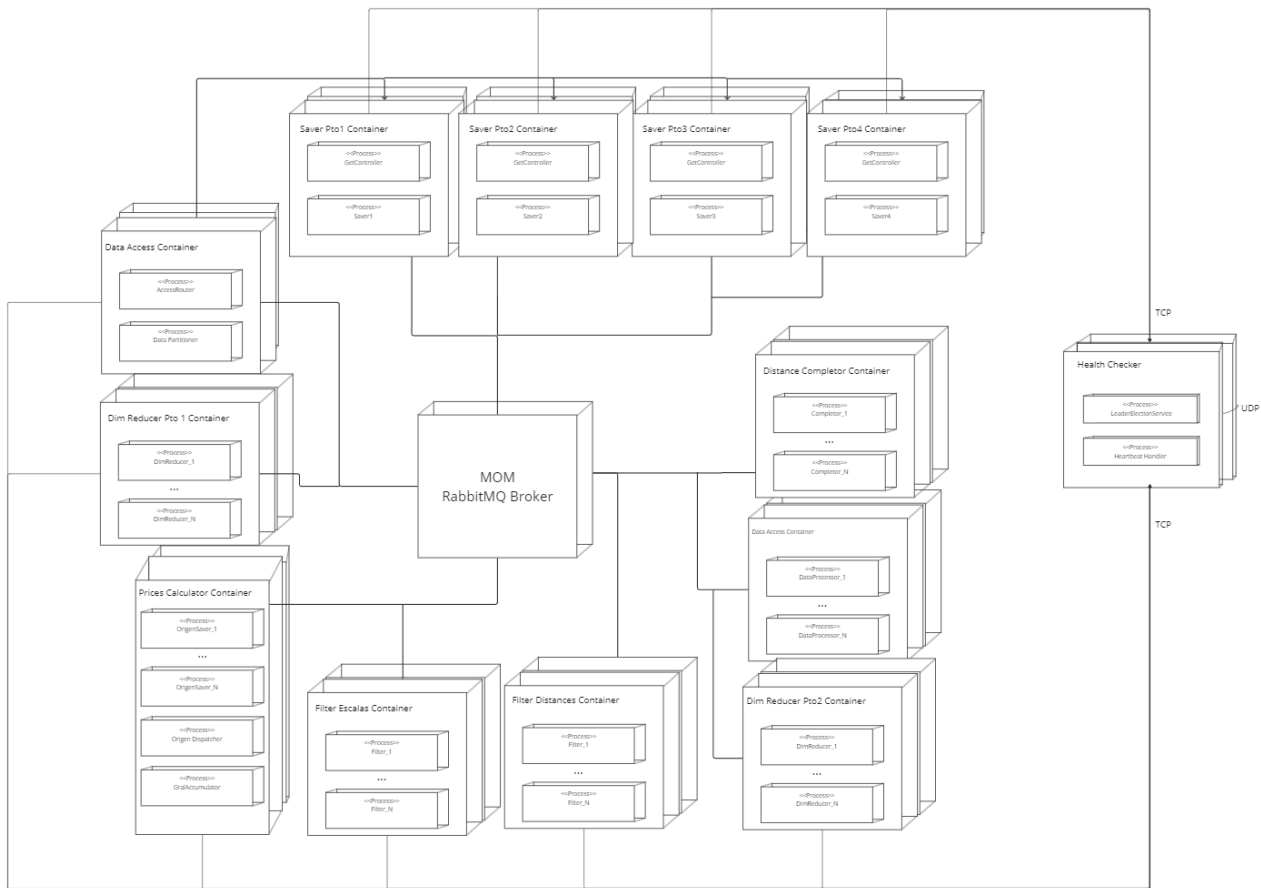


Se puede ver en mejor calidad en  Robustez.png

En el diagrama podemos ver los 16 servicios y los puntos de acceso que tiene el cliente.

- Los servicios General Accumulator, Trayecto Sink, los Savers, y los HealthCheckers son replicables.
 - En el caso de los HealthCheckers, estos se replican y deciden entre sí un líder para ser el que reinicie a los servicios caídos.
 - Los demás replican el estado al recibir la misma entrada.
- El resto de los servicios son escalables tanto a nivel contenedor como controladores internos ya que no tienen un estado.
 - Se pueden agregar contenedores que consuman y produzcan con las mismas colas.
 - Se puede configurar cuántos controladores internos tiene cada servicio mediante variables de entorno. Se tiene como mínimo 1 y máximo 32. Por defecto se tienen 4.

Diagrama de Despliegue



Se puede ver en mejor resolución en [Despliegue.png](#)

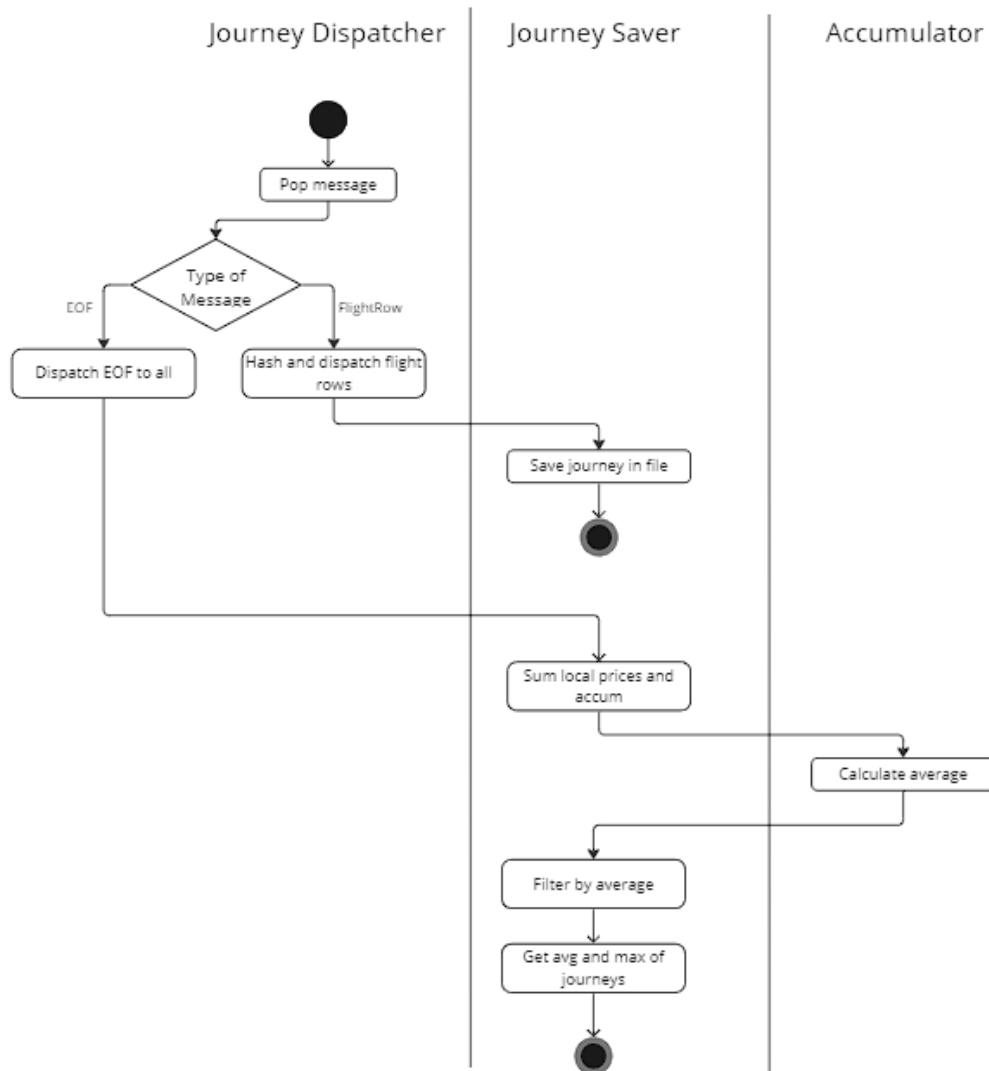
En el diagrama podemos ver cómo se relacionan los contenedores. Podemos destacar:

- Todos apuntan al MOM de RabbitMQ, por lo que el Middleware se vuelve clave para el funcionamiento del sistema.
- El servidor (Data Access Container) se comunica adicionalmente con los Savers para obtener los resultados a través de TCP.
- Todos los servicios se comunican con el Health Checker para determinar su estado de salud periódicamente vía TCP, y poder ser reiniciados vía docker-in-docker.
- Los Health Checkers se comunican entre sí para la elección del líder mediante UDP.

Vista de Procesos

Diagrama de Actividades/Escenarios Concurrentes

Actividades en la consulta 4

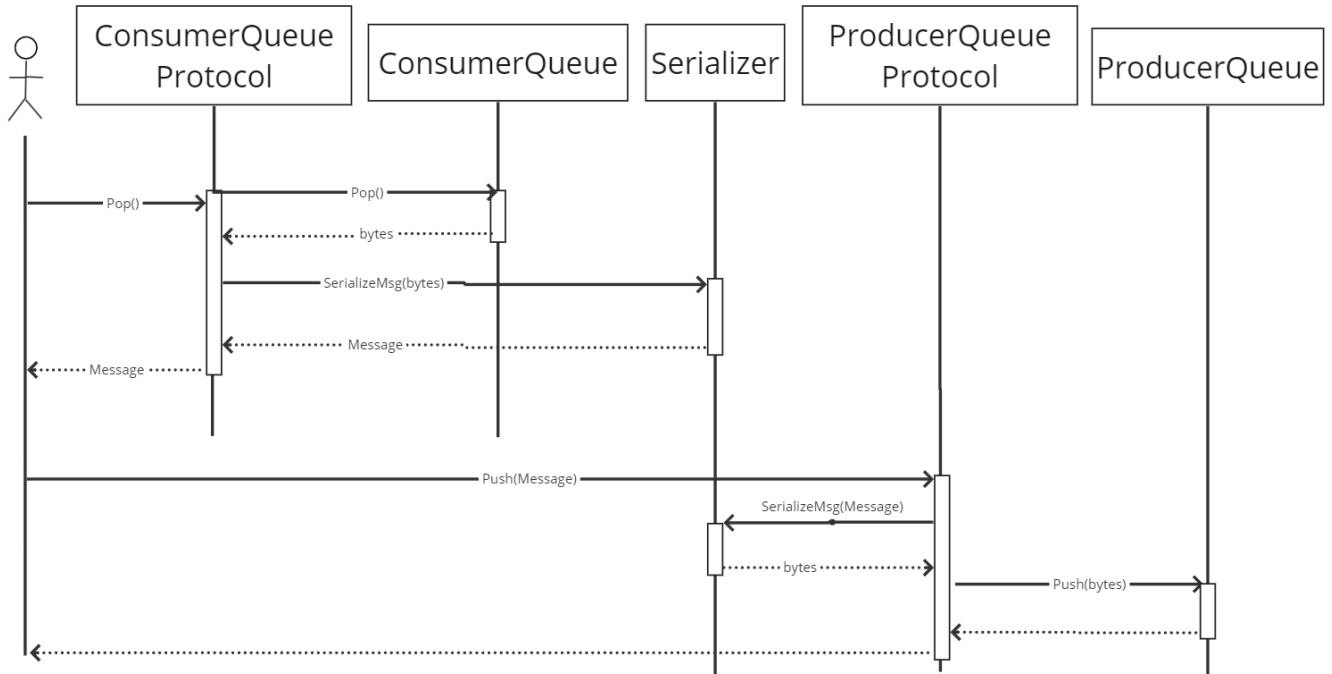


En el diagrama podemos observar la secuencia de actividades que se hacen para obtener los resultados de la consulta 4.

- Se repite la parte de los mensajes de FlightRows hasta obtener el mensaje de EOF, generando una iteración del presente diagrama de actividades por cada mensaje recibido.
- Al obtener el mensaje de EOF se realiza el cálculo de lo acumulado localmente y se envía al acumulador.
- El acumulador calcula el promedio con lo de todos los Journey Savers.
- Se envía el promedio para filtrar y obtener los valores buscados.

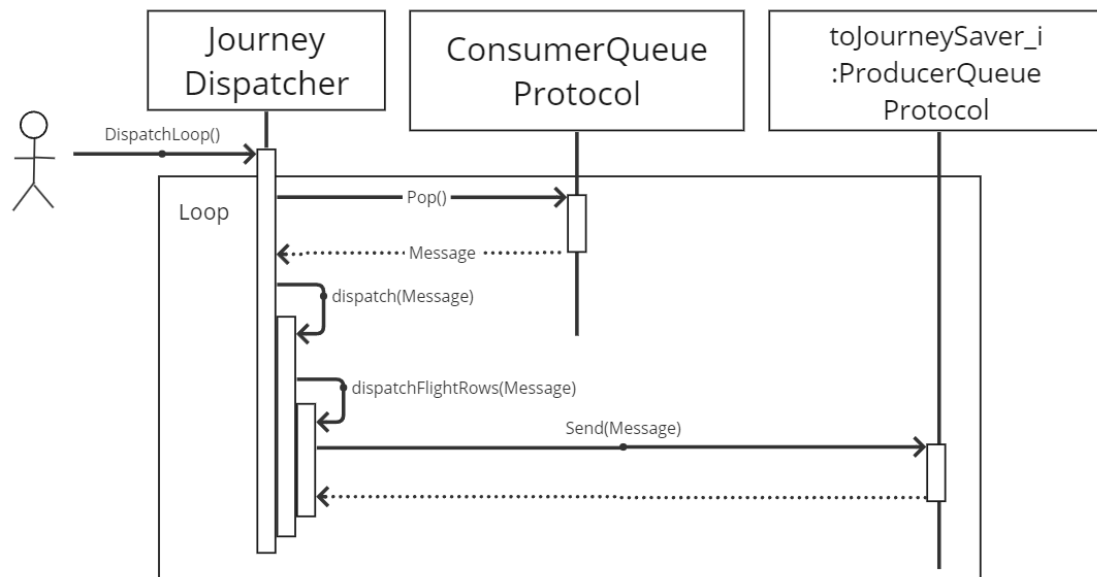
Diagrama de Secuencia

Interacción con las colas

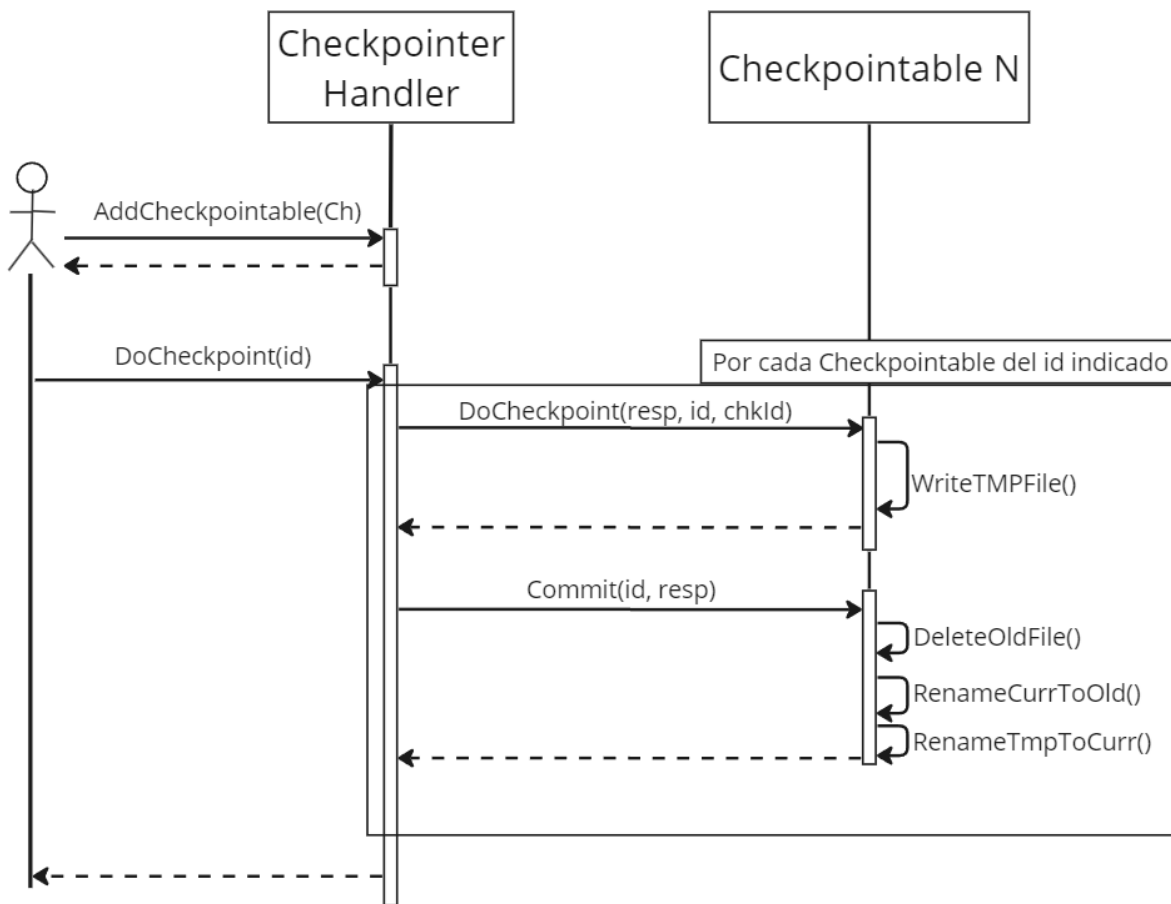


En el diagrama se puede observar la interacción que tienen la mayoría de los servicios con las colas, piden un mensaje, deserializan los bytes para devolver la estructura creada y el cliente lo procesa. Una vez procesado envía un mensaje, es serializado, y se envían los bytes.

Loop del Journey Dispatcher

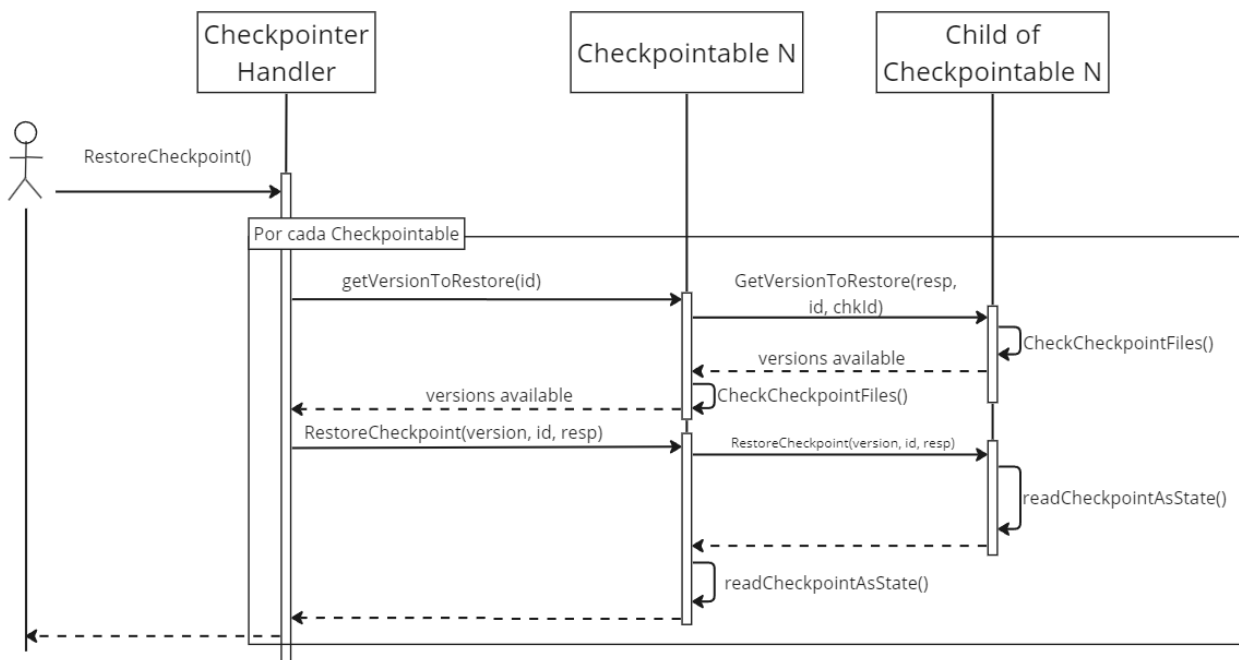


En este diagrama se puede observar el loop que maneja JourneyDispatcher para enviar los mensajes a los JourneySaver individuales en la consulta 4. En este caso el mensaje es de datos de vuelos y será hashado el trayecto para enviarlo a una cola única.

Realización de un Checkpoint

En este diagrama se observa como se realiza un checkpoint según nuestra interfaz de *checkpointables* (Vista más adelante en el presente informe). La idea consiste en persistir el estado a un archivo temporal que luego será renombrado y reemplazará lo que son los checkpoints anteriores, para que en caso de fallas, un *Checkpointable* pueda restaurar su estado anterior sin problemas.

El diagrama muestra un objeto sin hijos *Checkpointable*, si los tuviera repetiría la secuencia al delegarle las llamadas.

Restauración de un Checkpoint

Para la restauración de checkpoints se puede visualizar como existe una primer etapa en donde se consultan los archivos propensos a ser restaurados para lograr una versión común entre los distintos checkpointables.

Posteriormente, con la versión de checkpoint definida, se procede a restaurar el checkpoint específico en cada uno de los checkpointables leyendo dicho archivo y restaurando el estado interno de cada checkpointable. Una vez finalizado este proceso, los checkpointables poseen un estado válido y finaliza la restauración de los checkpoints.

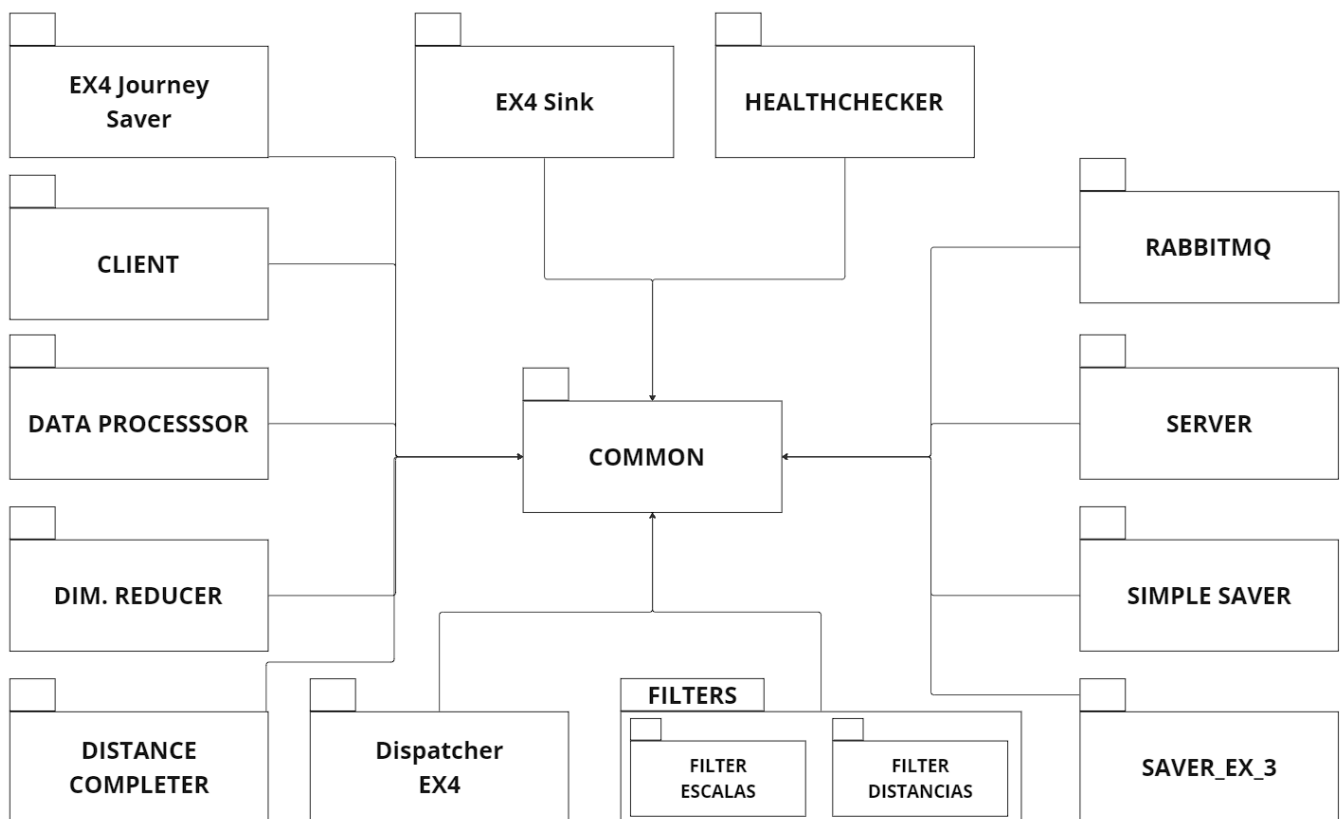
En el diagrama se muestra un Checkpointable con un hijo (el caso de las colas consumidoras con sus duplicados), si no tuviese un hijo esas llamadas ya las resuelve el mismo objeto (por ejemplo el Avg Accumulator del Ejercicio 4)

Vista de Desarrollo

Diagrama de Paquetes

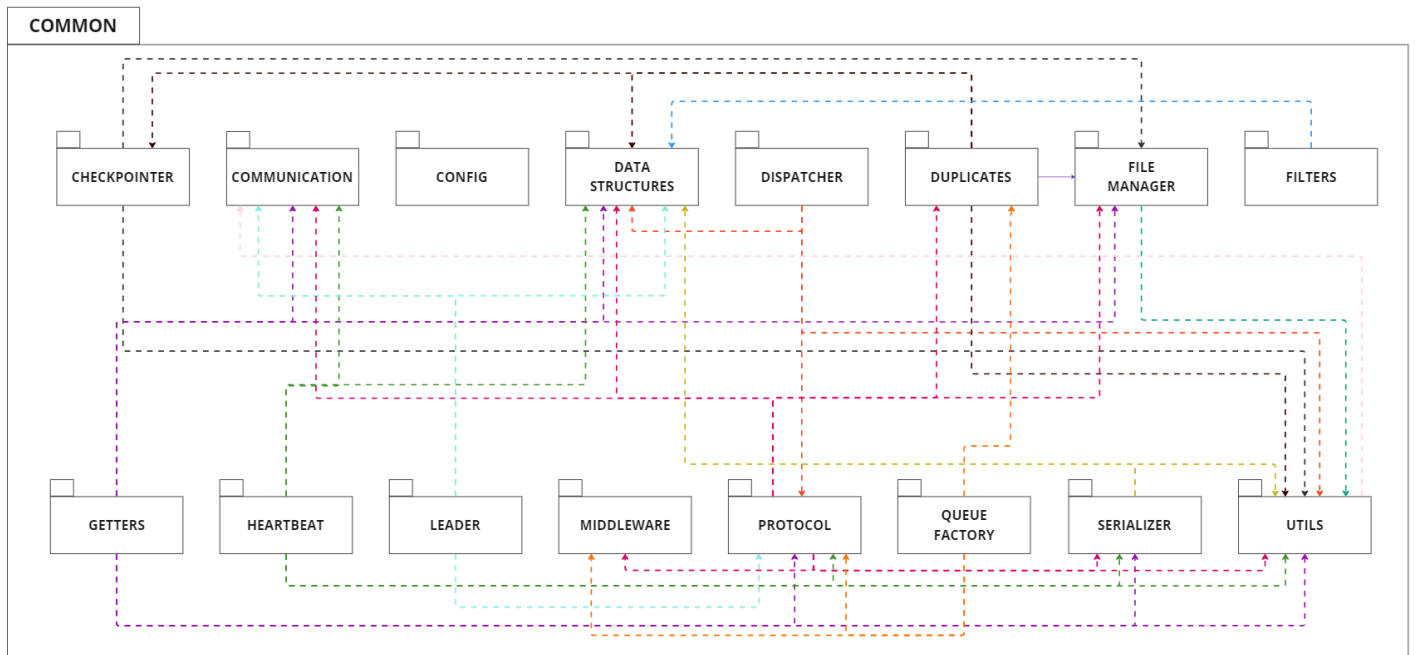
En el presente trabajo práctico vamos a encontrar que principalmente existe un paquete para cada servicio contenedorizado, ya que estos deben correr bajo su propio main, y tendrán cierto código propietario de dicho servicio (Mínimo en comparación con la escala del código compartido). Por otra parte, existe un paquete común que unifica los distintos structs con los cuales hemos trabajado a lo largo del proyecto Flights Optimizer, que se denomina como “common”.

Las interacciones entre paquetes pueden verse diagramadas en el siguiente esquema:



Se puede ver en mejor resolución en [Paquetes_Gral.png](#)

Y en específico para la carpeta common (Que es donde existen más paquetes internos):



Se puede ver en mejor resolución en [Paquetes_Common.png](#)

Como puede apreciarse, la estructura del paquete common es la más desarrollada, mientras que los paquetes de los servicios consumen principalmente de common para su correcto funcionamiento.

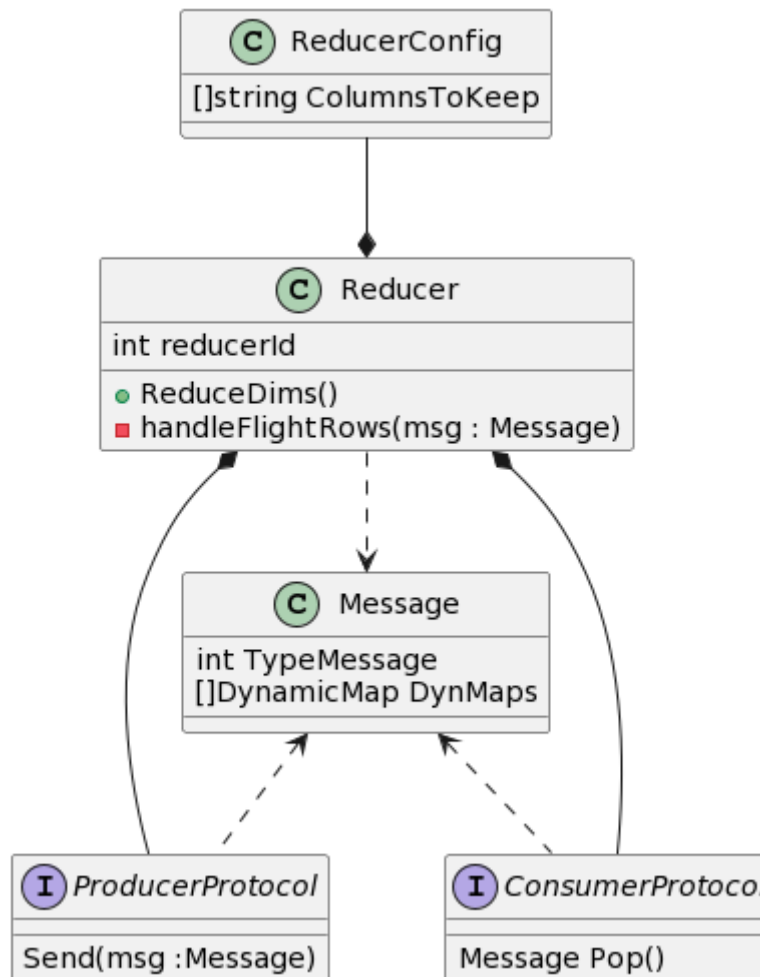
Vista Lógica

Diagrama de Clases

Para desacoplar los diferentes niveles de abstracción que manejan los servicios, utilizamos un modelo de capas junto con varias interfaces. Esto dio como resultado un modelo que tiene un alto nivel de abstracción y permite realizar pruebas unitarias sin que nos afecten las implementaciones de más bajo nivel.

Aplicaciones

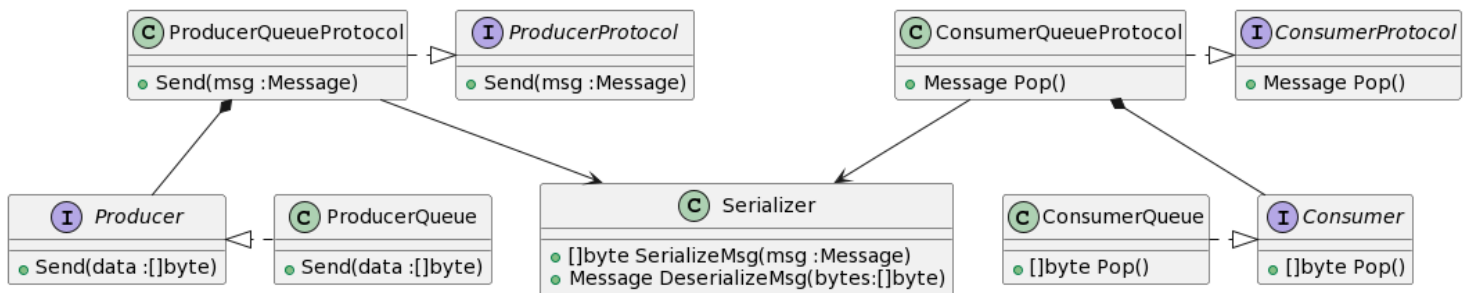
Se muestran algunos de los modelos de los servicios hechos.



En este diagrama se ve la capa de negocio del Dim Reducer, interactúa con el protocolo de consumidor para obtener un mensaje, lo maneja, y lo envía al productor. *Se omiten campos que no suman al diagrama.*

Los servicios de Data Processor y los Filter siguen este mismo modelo.

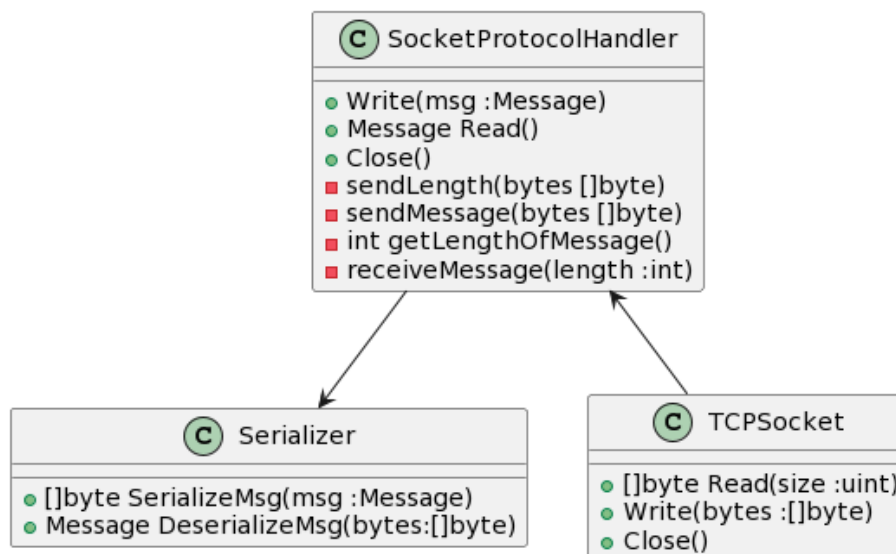
Protocolo de colas



En el diagrama podemos ver como está planteado el modelo de comunicación de colas. Este sistema está en todos los servicios desarrollados. *Se omiten campos que no suman al diagrama.*

- Se tiene la primera capa de abstracción que maneja mensajes (`ProducerProtocol` y `ConsumerProtocol`).
- Las implementaciones que convierten esos mensajes hacia y desde bytes
- La capa final (junto con su interfaz) que interactúa con Rabbit, ya solamente con bytes.

Protocolo con sockets

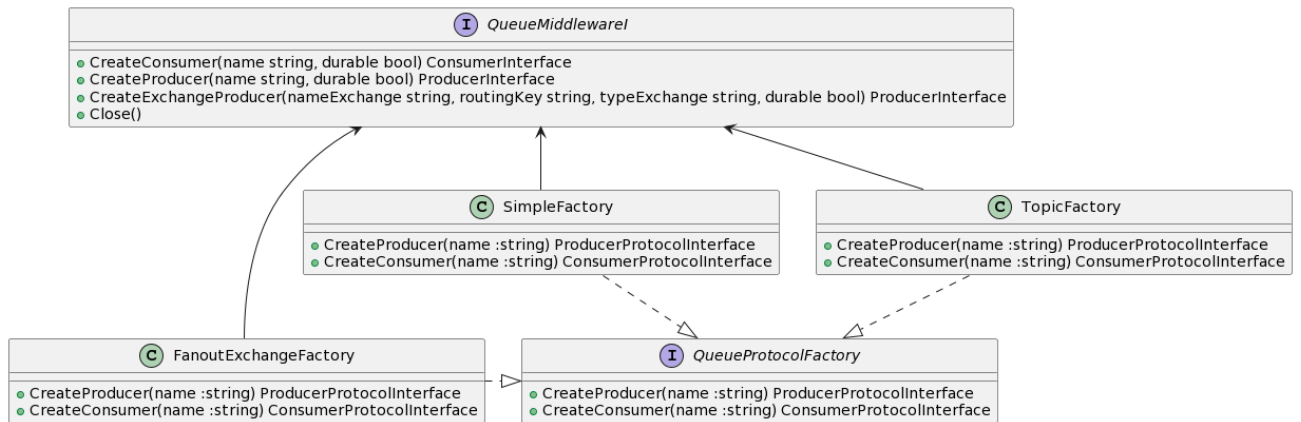


En el diagrama podemos ver como está planteado el modelo de comunicación mediante sockets en el caso del cliente, servidor, y getters. *Se omiten campos que no suman al diagrama.*

- `SocketProtocolHandler` es la clase que se encarga de manejar el protocolo, envía/lee el largo del mensaje primero y después el mensaje.
- `TCPSocket` maneja las lecturas y escrituras ya con bytes.

Creación de colas

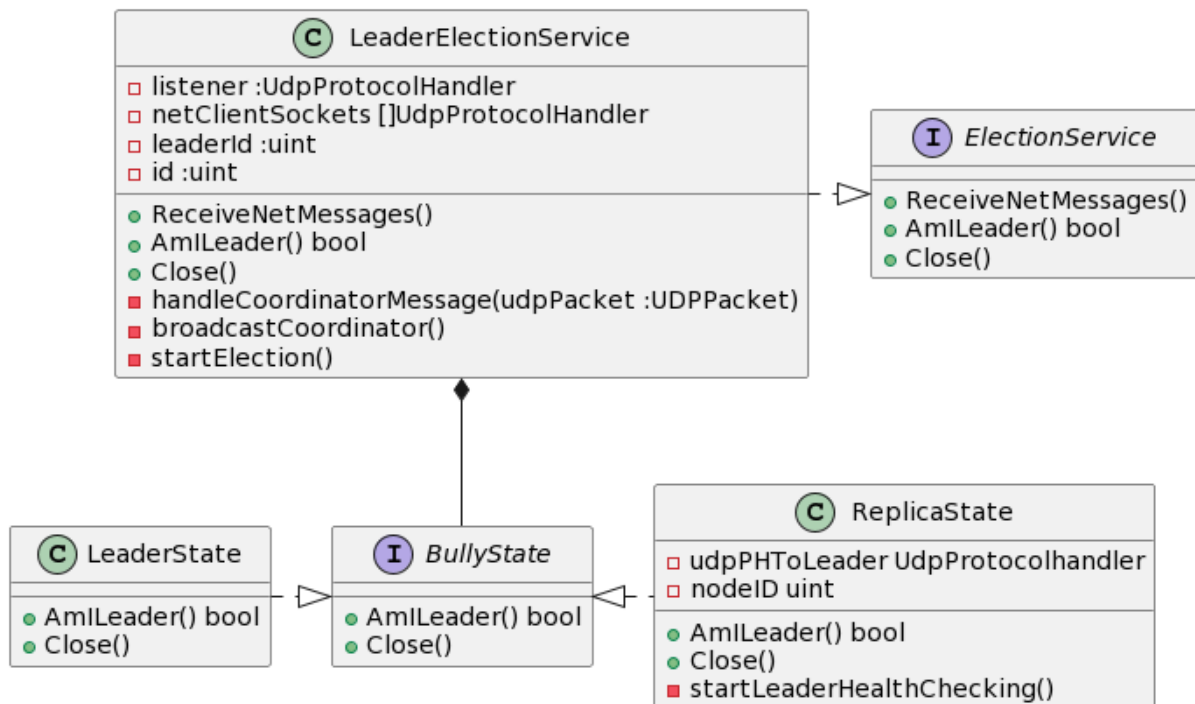
Para desacoplar la creación de las colas de los servicios decidimos implementar una Factory las cuales crean la cola junto con su protocolo.



Cada *factory* es instanciada dependiendo del servicio que la requiera.

Leader Service

Para el manejo del algoritmo de líder tenemos la siguiente estructura. Se muestran los métodos y atributos más relevantes solamente.



El método `'AmILeader()'` termina rompiendo el polimorfismo que se estaba teniendo al utilizar el state, no se llegó a realizar una solución en la cual se le delegue al estado la acción a realizar debido a limitaciones de tiempo.

Concurrencia/Paralelismo

El sistema tiene algunos recursos que hay que estar sincronizando el acceso, este es el caso de la escritura de un archivo (por ejemplo en los savers).

Ya al utilizar las colas de RabbitMQ podemos serializar el acceso a estos recursos y evitar condiciones de carrera, ya que solamente habría uno manejando la escritura a ese archivo.

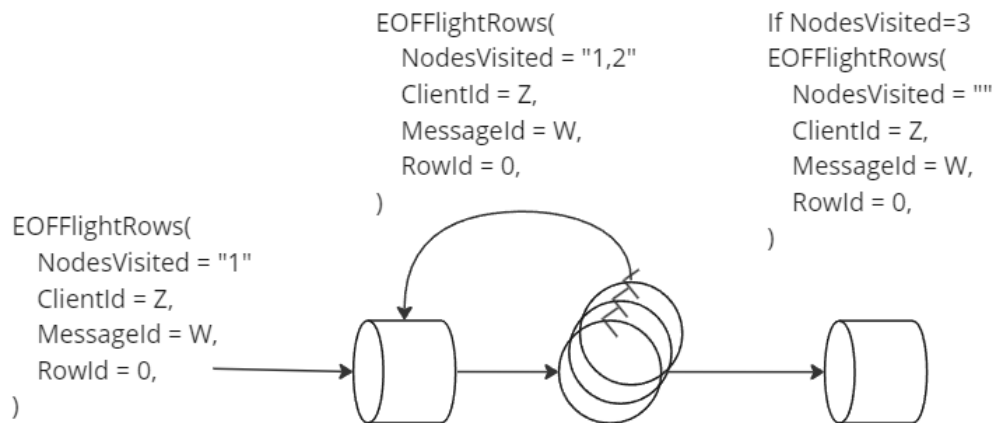
Dentro de un mismo servicio, por ejemplo para señalar que se terminaron los resultados de un cliente y que pueden ser devueltos, aprovechamos el file system para indicar este cambio de estado. Si se finaliza, movemos el archivo a una carpeta nueva del cliente.

Nota: Físicamente no se mueve, solo es un renombre.

Finalización

Debido a que la consulta 4 requiere de que se recorran todos los datos de entrada, y que el cliente necesita la confirmación de que los resultados obtenidos son los definitivos en las demás consultas (no faltan procesar datos), se vuelve necesario implementar un mecanismo de cierre o finalización del procesamiento del cliente.

Para esto, decidimos implementar un esquema donde se vuelve a reencolar el mensaje de finalización (EOF) y se propaga a las siguientes etapas del pipeline cuando ya recorrió la actual. De esta manera, nos aseguramos de que todos los controladores reciban la señal.



1. En la imagen se ve que se envía un mensaje EOF indicando cuántos controladores se recorrieron.
2. Un controlador recibe el mensaje y sabe que ya no van a venir más datos de un cliente, suma al mensaje su ID. Con esto lo vuelve a reencolar y pasará al siguiente consumidor en la asignación de RabbitMQ. En caso de que ya haya recibido este mensaje (porque tiene el ID) lo devuelve a la cola.
3. Otro controlador tomará el mensaje y repetirá el proceso hasta que sea el último (recorrió todos los controladores), en cuyo caso la totalidad de un cliente ya ha sido procesado en esta etapa y se puede pasar a la siguiente etapa del EOF limpiando los IDs.

Checkpointing

Para que el sistema pueda ser tolerante a fallas, hemos utilizado el mecanismo de checkpointing para persistir el estado que requieren utilizar los contenedores presentes en el sistema.

Interfaz Checkpointable

El mecanismo de checkpointing consiste en general en 3 archivos que son utilizados para cada objeto que implementa nuestra interfaz de “Checkpointable”. La interfaz cumple con los siguientes métodos:

```
type Checkpointable interface {
    DoCheckpoint(response chan error, id int, chkVersion int)
    RestoreCheckpoint(chkToRestore int, id int, res chan error)
    GetCheckpointVersions(id int) [2]int
    Commit(id int, response chan error)
    Abort(id int, response chan error)
}
```

Checkpoint Handler

El checkpoint handler actúa como un conocedor de los checkpointables y coordina la escrituración de los checkpoints y la restauración de los mismos a nivel de nodos (Controladores) que existan dentro de un contenedor.

Cada checkpoint handler posee internamente los distintos checkpointables y con ellos logra coordinar los algoritmos de Do Checkpoint y Restore Checkpoint que se ven detallados en las posteriores subsecciones.

Tipos de Archivos

Los archivos que se utilizan entonces para realizar un checkpoint se pueden separar como:

- **OLD:** Es el archivo que corresponde al último checkpoint válido, actúa como backup del actual checkpoint en caso de que llegase a existir algún error al persistir el mismo.
- **CURRENT:** Es el checkpoint más actual, proveniente de un renombramiento del archivo TMP y es el archivo que en general suele usarse para restaurar el sistema salvo ciertos casos borde (Commits parciales en los

`checkpointables`, errores de escritura, invalidez en algún `checkpointable`, etc).

- **TMP:** Es donde se realiza el checkpoint a forma de escritura. Luego, al realizarse el commit del checkpoint se renombrará a ser el nuevo `current`. El TMP no tiene validez en sí mismo como un checkpoint para nuestro algoritmo, ya que únicamente actúa como buffer para evitar inconsistencias en los checkpointings. Tendrá validez a partir de que sea denominado como el nuevo `CURRENT`.

Formato de Archivos

Cada archivo consta del siguiente formato:

- Una primera línea de cabecera indicando el número de checkpoint (Identificador) que posee dicho archivo.
- El resto de líneas indicando el estado del `checkpointable` en cuestión, siendo el formato estipulado con una interfaz de tipo `writable` que utiliza el mismo, e interpretado con la función de `Restore Checkpoint` que implementa el `checkpointable`. Este formato depende de cada servicio, por lo general siguen un estilo `csv` con distintos separadores para cada parte del estado.

Algoritmo de generación de Checkpoint

Conociendo el formato de los archivos y cómo funcionan los 3 tipos de archivos diferentes podemos entonces explicar el algoritmo de checkpointing en cuestión:

- 1) Al recibir un mensaje de tipo `batch` este será procesado en su totalidad y enviado a la salida correspondiente en caso de requerirse.
- 2) Al finalizar el procesamiento se hace llamado a la función `Do Checkpoint` para crear el archivo TMP en cada uno de los `checkpointables` del nodo en cuestión.
- 3) Finalizado el `Do Checkpoint` del nodo se realiza un commit (o abort si se ocurriese):
 - a) Eliminación del archivo OLD (A esta altura ya no es el válido actual, se debería usar el `CURRENT` en caso de fallos)
 - b) Renombramiento de `CURRENT` a OLD (Sigue manteniendo validez como OLD)
 - c) Renombramiento de TMP a `CURRENT`.

- 4) Finalizado el COMMIT se realiza ACK del batch de mensajes obtenido (Evitando la relectura del duplicado).

Cabe aclarar que en caso de fallos durante la instancia de checkpointing el mensaje sería consumido nuevamente ya que actualmente la política es de “At Least Once” y no de “At Most Once”, y como el checkpoint en todo caso persiste la identificación de duplicados, si este finalizara y el ACK no se pudiera lanzar, sería detectado posteriormente el mensaje como duplicado en el consumidor y se descarta para evitar un reprocesamiento.

Algoritmo de Restauración de Checkpoint

Para restaurar el checkpoint el procedimiento es sencillo:

- 1) Para cada `checkpointable` se lee la primera línea de los archivos OLD y CURRENT en caso de existir, y se retornan los 2 identificadores de checkpoint válidos que estos posean.
- 2) Se comparan las versiones válidas y se toma la máxima válida de entre todos los `checkpointables` (Todos deben poseerla a nivel de nodo para que sea válida en su totalidad).
- 3) Con la versión válida de checkpoint se llama a la función `Restore Checkpoint` junto con el identificador del `checkpointable` para que sepa cuáles archivos se deben tomar para la restauración.
- 4) Cada `checkpointable` lee sus archivos específicos interpretando su estado recuperado y tomándolo dentro del estado interno del struct para la correcta restauración del objeto. Si algo dentro del archivo no se puede comprender debido a algún error, es descartada esa línea.
- 5) Finalizada la restauración se lanzan a correr las Go Routines usuales de procesamiento del contenedor en cuestión.

Detección de duplicados

Dado que estamos en un sistema distribuido que puede tener nodos caídos, se pueden llegar a presentar mensajes duplicados. Para manejarlos decidimos implementar un sistema de IDs y su registro para detectarlos y reducir su impacto.

Debemos destacar primero que este manejo se encuentra actualmente en todos los servicios, los que tienen estado (por ejemplo los savers y el ejercicio 4) y los que no (reducers, filtros) como una optimización para reducir los mensajes circulando por la red. Si quisiéramos podría estar solamente en los que tienen estado, ya que estos son los que sufren las consecuencias realmente.

Para detectar un mensaje duplicado, debemos de analizar el ID conjunto:

- El Client ID es un UUID que representa a un cliente
- El Message ID es un entero secuencial, a medida que el cliente envía un mensaje se va aumentando.
- El RowId es otro entero secuencial que está a nivel fila, inicia en 0 y cambia más adelante en el sistema. Hay que recordar que nuestros mensajes son por batch, por lo que un solo MessageID no nos es suficiente, al momento de procesar a nivel fila (por ejemplo en el dispatcher) se empieza a cambiar este valor.

Al recibir un mensaje, se llama a la estructura `DuplicatesHandler` que sigue la siguiente lógica:

```
//Structure ClientID -> MessageID -> RowID  
lastMessagesSeen map[string]map[uint]uint16
```

1. Si el Client ID no existe, no es duplicado
2. Si el Message ID es aún más viejo que el más viejo de los mensajes guardados (y tengo la lista de mensajes llena), es un duplicado. Se compara esto mediante su valor entero dado que es secuencial.
3. Si se encuentra el Message ID, pero el Row ID es mayor al detectado, no es duplicado.
4. Cualquier otro caso es un duplicado

Ya con ese resultado, el protocolo consumidor puede tomar los siguientes caminos:

- Si fue detectado como duplicado, se le hace ACK al mensaje para que sea descartado y se pide otro mensaje.
- Si no es duplicado, se agrega a la lista de mensajes vistos y se devuelve para procesarlo.

Un par más de cuestiones a tener en cuenta:

- La lista de mensajes vistos no almacena todos los mensajes, solamente guarda los últimos 1000 por cada cliente, se aprovecha la secuencialidad de los valores para la detección. Si se llega a llenar se reemplaza el más viejo.
- Esta lista de mensajes es persistida durante los checkpoints para asegurarnos que no se ven afectados los resultados ante una caída. Si se llega a caer se restaura el estado y sigue funcionando normalmente.

Pendientes

Mencionamos algunos pendientes o mejoras que quedaron durante la realización del trabajo práctico.

- Realizar mayor cantidad de tests unitarios de los componentes.
- Quedan pendientes la realización de refactors y limpieza de código. Algunas partes del código seguramente puedan ser simplificadas o eliminadas completamente ya que se tuvieron que realizar cambios en varias secciones.
- Al ser un trabajo con mucha cantidad de código con el tiempo se fue tornando más difícil ubicarse en donde se estaba parado, alcanza con ver el diagrama de paquetes y su nivel de horizontalidad que tiene, quedaria como pendiente revisar si se puede hacer algún reordenamiento de los paquetes para simplificarlos.
- Mejorar la lectura de los archivos, especialmente en el lado del cliente para que se mande a leer por chunks más grandes en vez de línea por línea. Se leería un chunk del archivo y se parsearía para el envío.
- Al estar haciendo checkpointing por cada mensaje (considerando que en cada uno vienen varias filas) se agrega un overhead al estar escribiendo a disco para ser lo mayor posible tolerante a fallas. A mayor cantidad de filas enviadas en el batch, menor va a ser este overhead. Se plantea como una mejora realizar el checkpointing cada cierta cantidad de mensajes en vez de que sea de a uno.

El impacto de esta mejora ayudaría más que nada a la performance de la consulta 4. Este ejercicio se ve afectado porque los dispatcher envían de a una fila a un journey saver (ya que hacen el hash del trayecto) y se termina teniendo un mayor overhead.

Otra alternativa para el ejercicio 4 es que los dispatcher acumulen cierta cantidad de mensajes para cada saver y enviar un mensaje con el batch acumulado.

- Modificar o adaptar el Haversine y pasar a usar float 64 para evitar errores de redondeo al filtrar por distancia en el ejercicio 2.