

# Trabajo Práctico Escalabilidad

## Informe

Sistemas Distribuidos 1

75.74

2C 2023

### Integrantes

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

# Índice

<b>Introducción.....</b>	<b>3</b>
<b>Objetivo.....</b>	<b>3</b>
<b>Supuestos.....</b>	<b>3</b>
<b>Alcance.....</b>	<b>4</b>
<b>Archivos.....</b>	<b>5</b>
Aeropuertos.....	5
Vuelos.....	6
<b>Tecnologías de implementación.....</b>	<b>6</b>
<b>Ejecución.....</b>	<b>7</b>
<b>Protocolos de comunicación.....</b>	<b>8</b>
Mensajes.....	8
Por sockets.....	11
Por colas.....	12
<b>Escenarios.....</b>	<b>12</b>
Diagrama de Casos de Uso.....	12
<b>Vista Física.....</b>	<b>13</b>
<b>Grafo Acíclico Dirigido (DAG).....</b>	<b>13</b>
Diagrama de Robustez.....	14
Diagrama de Despliegue.....	16
<b>Vista de Procesos.....</b>	<b>17</b>
Diagrama de Actividades/Escenarios Concurrentes.....	17
Diagrama de Secuencia.....	18
<b>Vista de Desarrollo.....</b>	<b>19</b>
Diagrama de Paquetes.....	19
<b>Vista Lógica.....</b>	<b>20</b>
Diagrama de Clases.....	20
<b>Concurrencia/Paralelismo.....</b>	<b>22</b>
<b>Finalización.....</b>	<b>23</b>
<b>Pendientes.....</b>	<b>24</b>

## Introducción

El presente documento reúne la documentación de la solución al trabajo práctico de escalabilidad 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 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.

## 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.

Se deja entre paréntesis el responsable de cada uno.

- **Desarrollo de cliente.** Será el cliente que inicializa un pedido de procesamiento pasando los archivos de datos. *(Bruno)*
- **Desarrollo de servidor como punto de entrada.** Será el punto de entrada y salida del sistema. El cliente realiza las peticiones de procesamiento que son atendidas por este servidor. *(Franco)*
- **Desarrollo de middleware de comunicación por colas.** Será una abstracción para poder utilizar el framework RabbitMQ y que el mismo pueda ser reemplazado sin tener que modificar el sistema completo. *(Bruno y Franco)*
- **Desarrollo de servicio de procesamiento de datos.** Quita columnas e información que no va a ser necesaria para el sistema y genera cálculos que van a ser requeridos. *(Bruno)*
- **Desarrollo del completador de distancias:** Accede al dataset de aeropuertos obteniendo la latitud y longitud para calcular la distancia en caso de que la misma falte en las líneas del dataset entrante. Ya sea la directa o total. *(Bruno y Franco)*
- **Desarrollo del filtrado por cantidad de escalas.**<sup>1</sup> Filtra las filas de acuerdo a la cantidad de escalas. En nuestro caso son 3 escalas (Variable de entorno que podría ser configurada). *(Franco)*
- **Desarrollo del filtrado por distancias.** En este caso se busca verificar que la distancia total sea mayor a la distancia directa cuatro veces (Este multiplicador podría ser configurable por variables de entorno). *(Franco)*
- **Desarrollo del reductor de dimensiones.** Su foco es reducir la cantidad de datos finales a persistir para el punto 1 (Ya que los datos entrantes sirven también para el punto 3 y son sobrantes para este caso). *[Este caso también aplica para el reductor del punto 2, con sus propias columnas de dimensión, por lo que deberá de ser genérica la implementación]* *(Bruno)*

---

<sup>1</sup> Este servicio y el de filtrado de distancias se van a plantear como un servicio genérico de filtrado para reutilizar código.

- **Desarrollo de servicios para guardado de datos de cada punto junto con su recuperación.** Cada uno guarda los datos correspondientes a su punto. (*2 Savers cada uno*)
- **Desarrollo del servicio de obtención de resultados del punto 4 al obtener una señal de finalización de entrada de datos.** Poseyendo los datos de origen-destino persistidos hasta el momento de señalización de fin al punto 4, y los datos generales, se comparará cada precio de vuelo individual por trayecto contra la media general de precios, y se calculará la media y máxima de cada “origen-destino” con los vuelos que cumplan con el filtro para obtener el resultado final. (*Bruno y Franco*)

## Archivos

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.

### Ejecución

Se provee un `docker-compose-dev.yaml` que tiene todas las configuraciones requeridas para ejecutar el sistema 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.

## 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)

### Mensajes

Nuestros mensajes se puede representar de la siguiente forma:

```
type Message struct {
    TypeMessage int
    DynMaps      []DynamicMap
}
```

Donde `TypeMessage` 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

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.

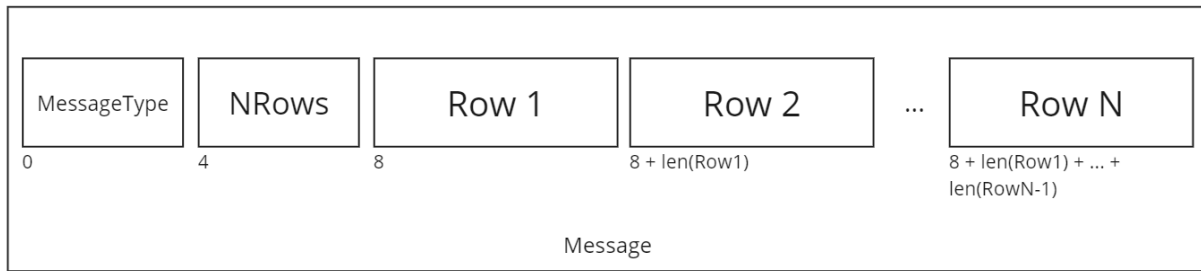
Al ser un array de mapas, se abre la posibilidad a realizar envíos de mensajes donde tengan chunks o batches. La cantidad de mapas a enviar es configurable en las correspondientes aplicaciones que pueden iniciar el stream. Como valor por defecto se tomó 300.



### 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.

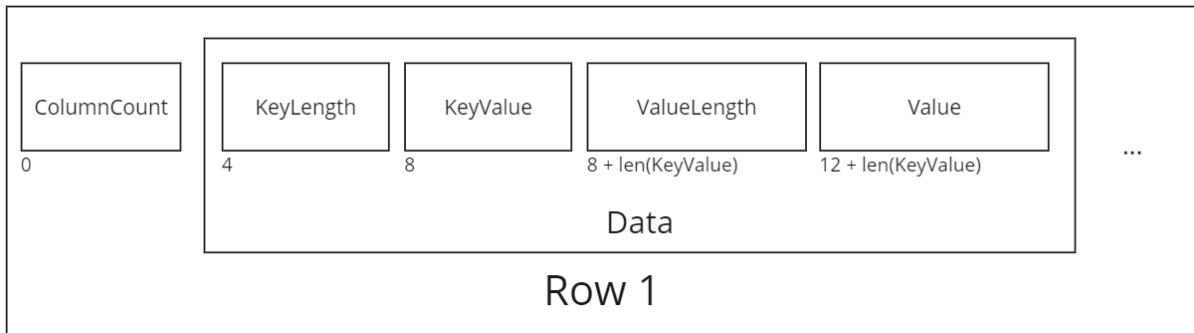
El mensaje entero en bytes se va a representar de la siguiente manera:



- 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.
- 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:



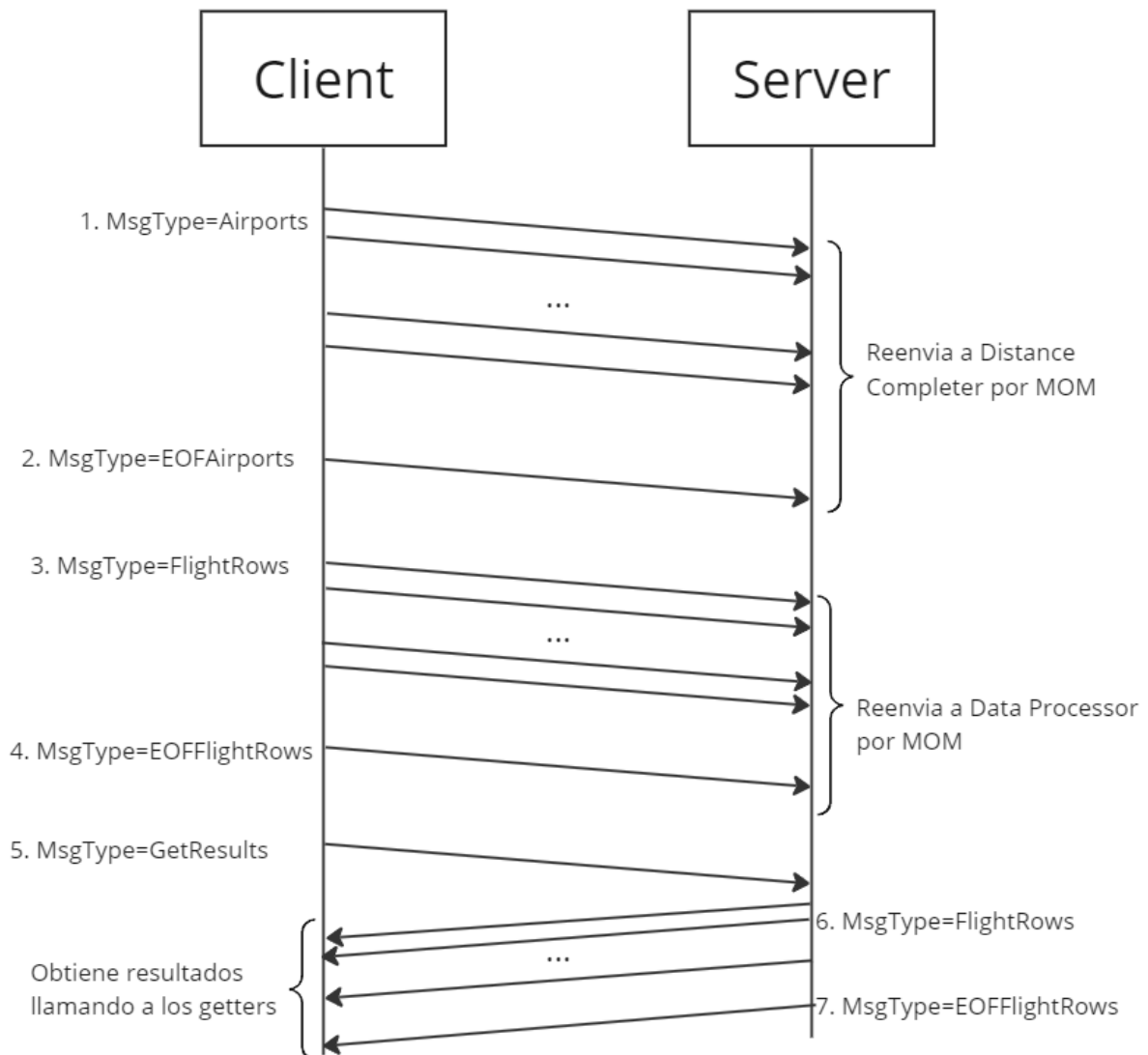
- Primero viene la cantidad de columnas (Data) que tiene la fila. Se va a estar repitiendo esta cantidad de veces la estructura Data
- KeyLength es el largo que tiene la clave de nuestro mapa dinámico. Es un entero de 4 bytes en Big Endian
- KeyValue es la clave del dato, tiene el largo de KeyLength. 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.*

## Por sockets

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*

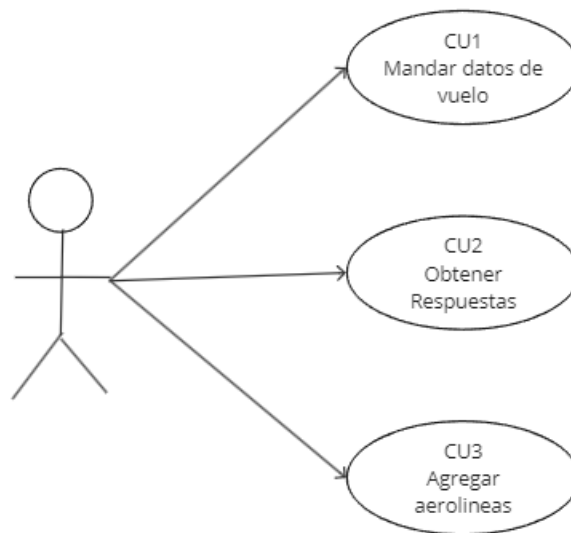
### Por colas

La comunicación por colas 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.

### Escenarios

#### Diagrama de Casos de Uso

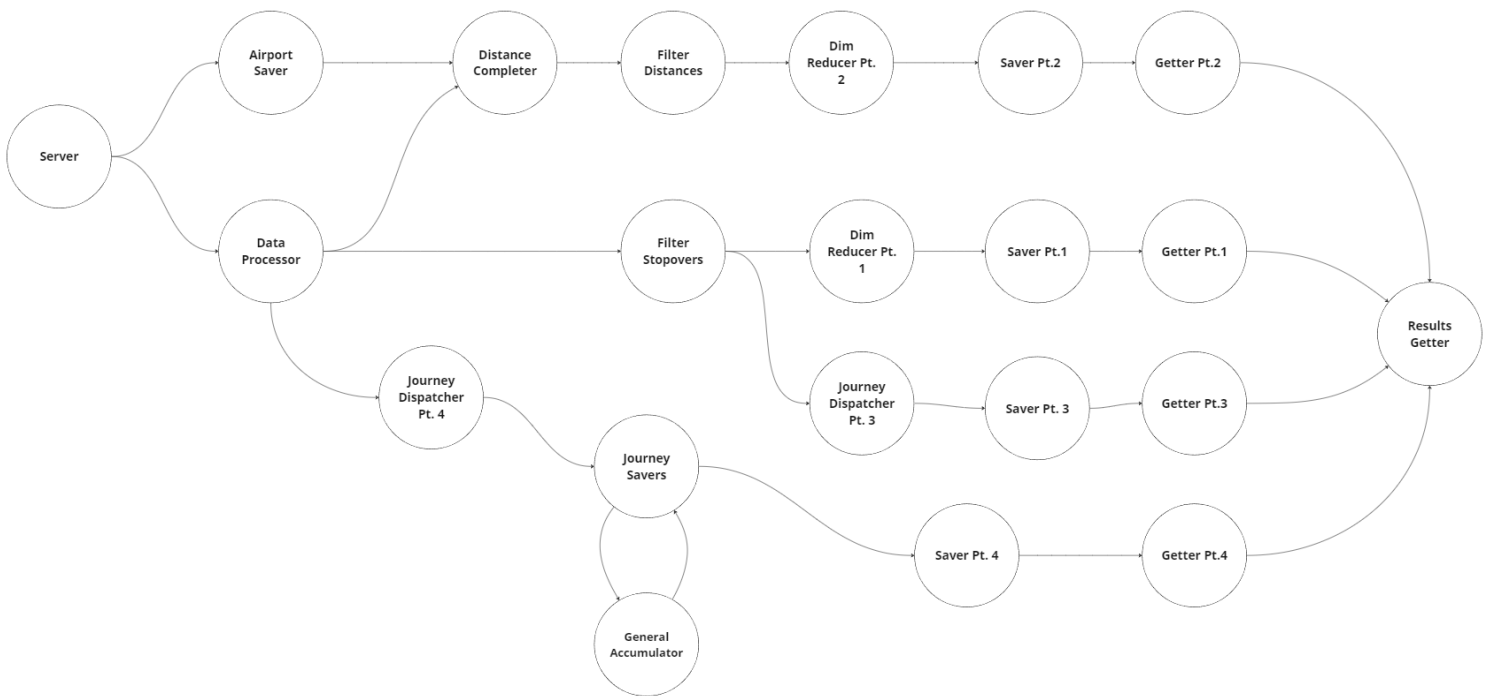
Nuestro sistema tiene los siguientes casos de uso.




- 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

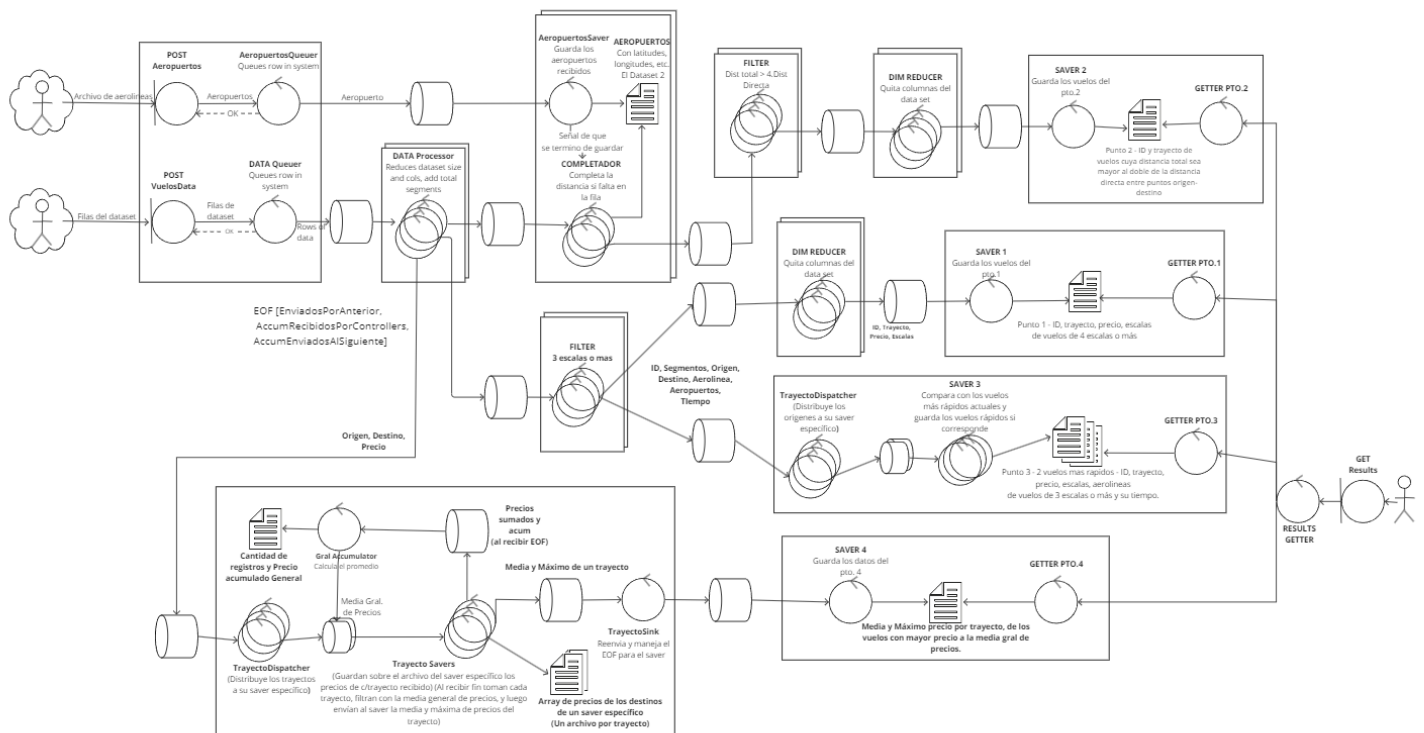
En la figura podemos ver el DAG del sistema. Analizamos sus nodos:

- **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

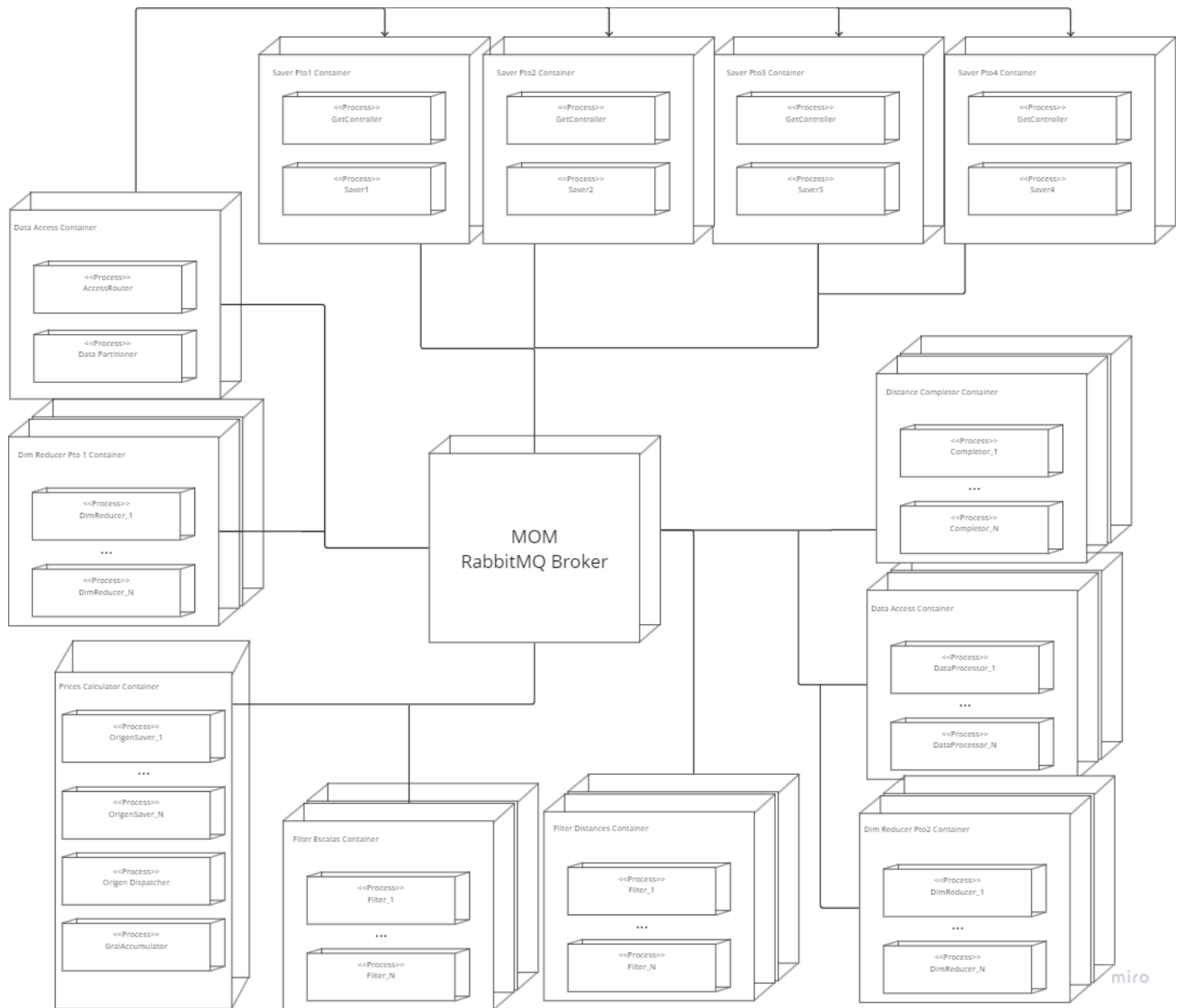


Se puede ver en mejor calidad en  Robustez.jpg

En el diagrama podemos ver los 11 servicios y los puntos de acceso que tiene el cliente.

- El servicio del ejercicio 4 y los Savers no son fácilmente escalables debido a las operaciones que realizan.
  - El ejercicio 4 requiere de que se haya recorrido todo el archivo y recibir el EOF para poder hacer procesamiento, por lo que se tiene que estar manteniendo cierto estado para avanzar.  
Avanzado el desarrollo se nos fueron ocurriendo mejoras que se le podrían hacer:
    - a. Manejar cierta lógica de almacenamiento en memoria para diferir y reducir las escrituras al archivo del trayecto.
    - b. Crear múltiples JourneySavers que reciban de la misma cola de entrada y coordinarlos en el acceso a los archivos. En caso de que estén procesando distintos trayectos (por lo que están en distintos archivos) se mejoraría la performance.
  - Los savers escriben en un archivo único los resultados. Se podría hacer una mejora al diferir la escritura y manejar hasta cierto punto los datos en memoria.
- 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:

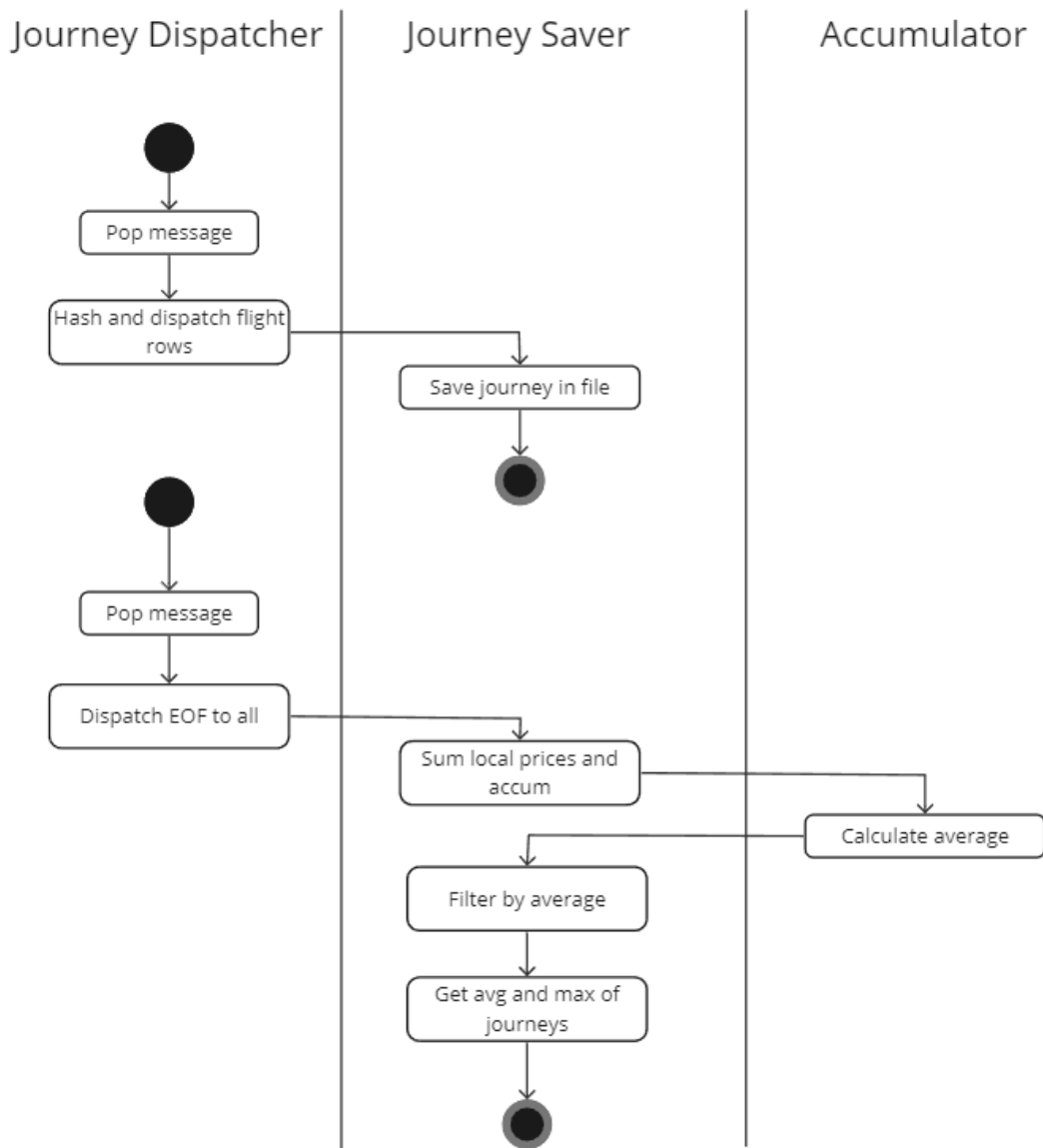
- Todos apuntan al MOM de RabbitMQ, por lo que el Middleware se vuelve clave para el funcionamiento del sistema.
- El servidor (Data Access Container) se comunica adicionalmente con los Savers para obtener los resultados.



## Vista de Procesos

### Diagrama de Actividades/Escenarios Concurrentes

#### Actividades en la consulta 4

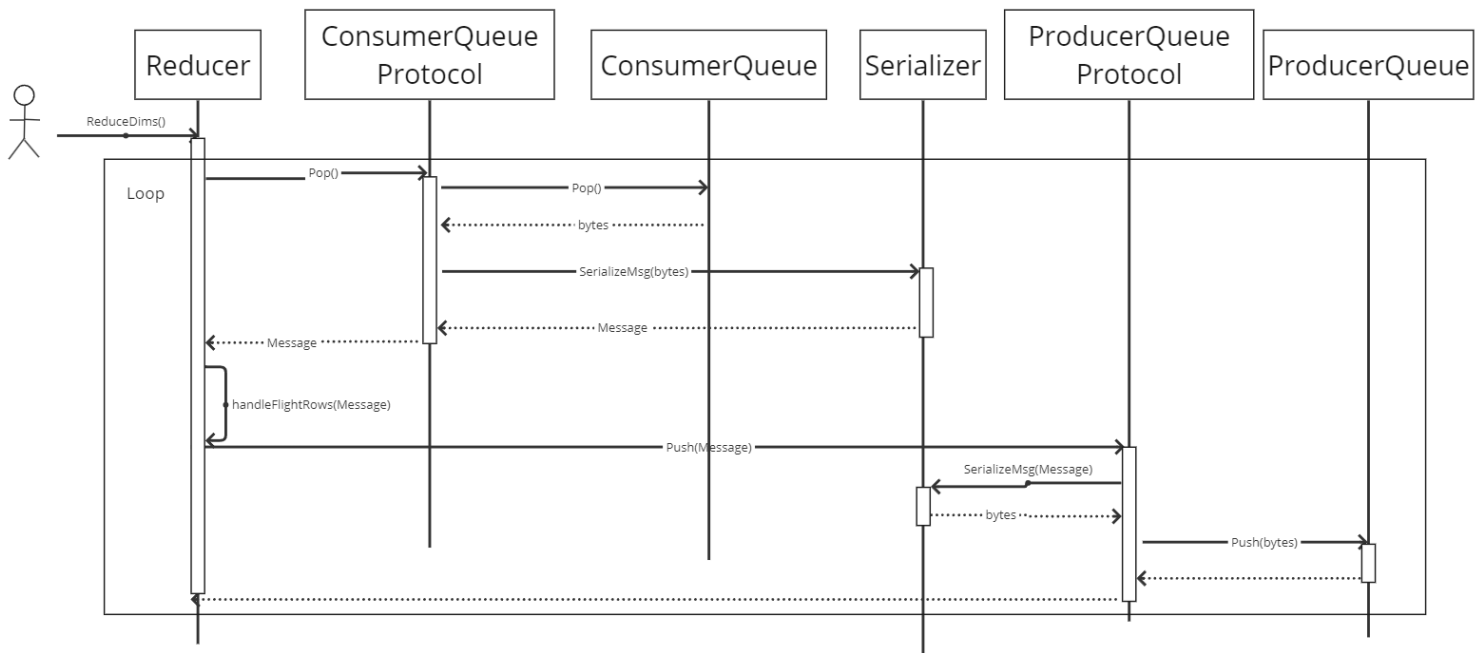


En el diagrama podemos observar la secuencia de actividades que se hacen para obtener los resultados de la consulta 4.

- Se repite la primera parte hasta 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 JourneySavers.
- Se envía el promedio para filtrar y obtener los valores buscados.

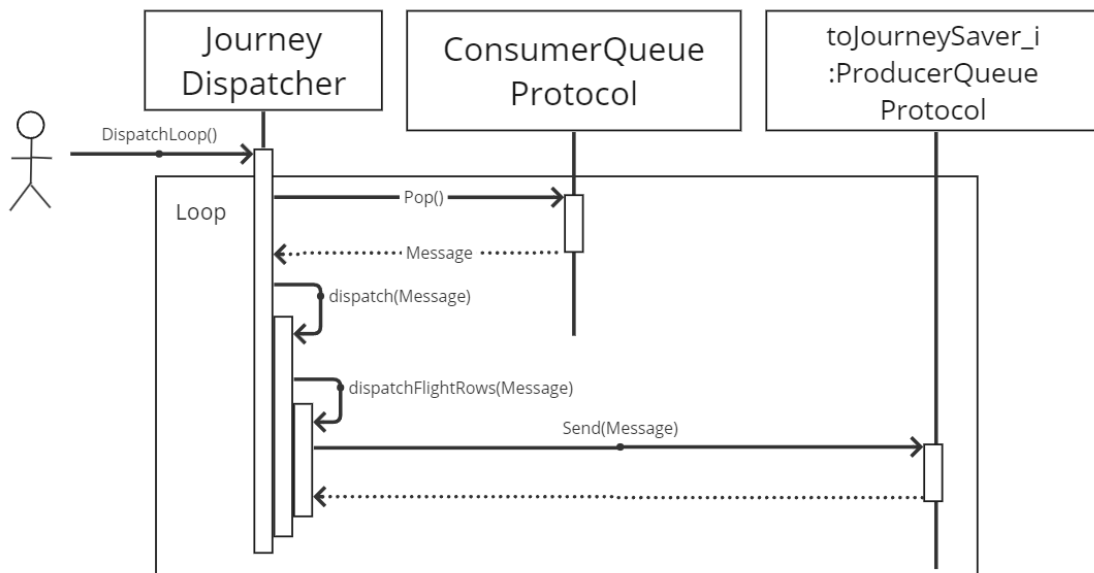
## Diagrama de Secuencia

### Loop general



En el diagrama se puede observar el loop del servicio de DimReducer, la mayoría de los servicios siguen una lógica similar donde solamente cambia la capa de negocio (en el diagrama: Reducer)

### Loop del Journey Dispatcher



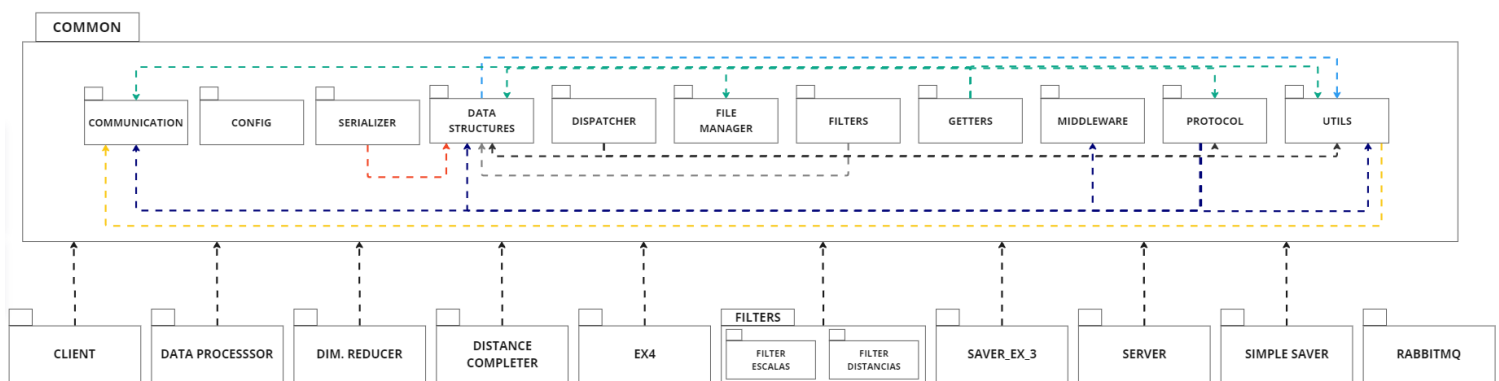
En este diagrama se puede observar el loop que maneja JourneyDispatcher para enviar los mensajes a los JourneySavers individuales en la consulta 4. En este caso el mensaje es de datos de vuelos y será hashado el trayecto para enviarlo a una cola única.

## 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.jpg](#)

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.

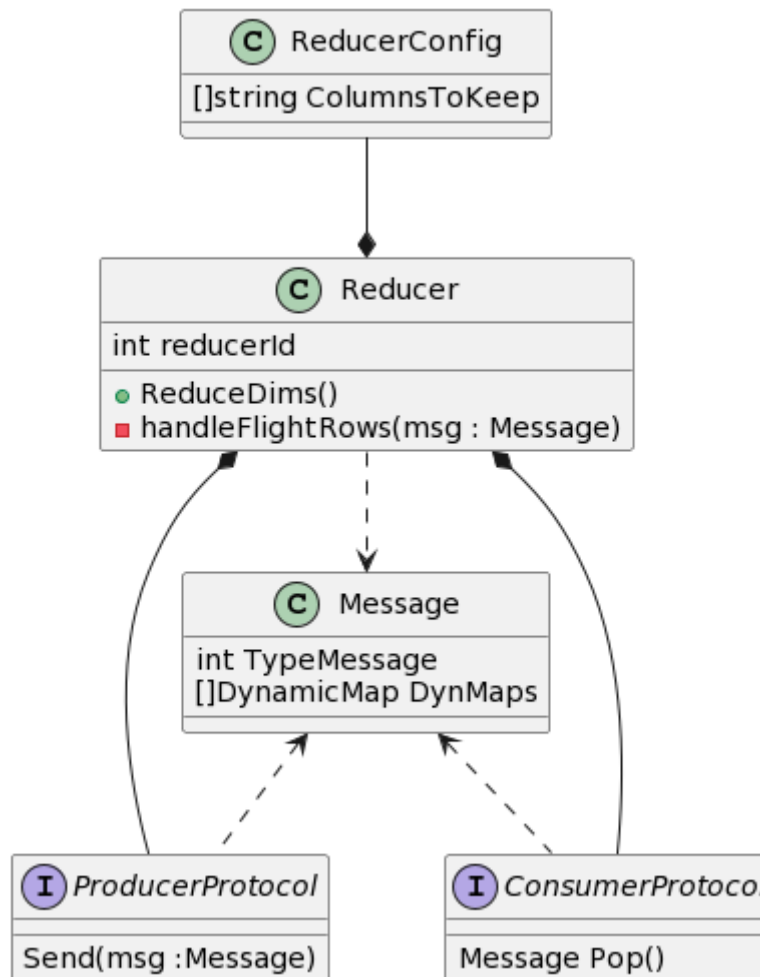
## Vista Lógica

### 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

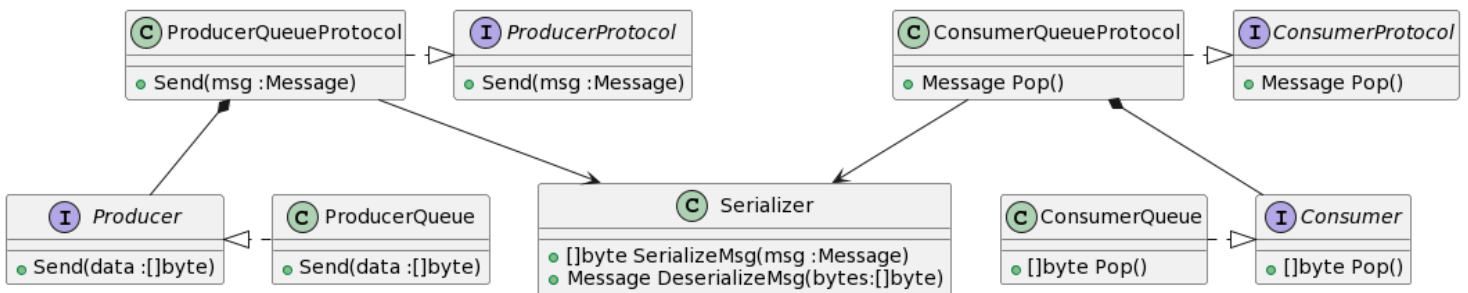
Se muestran algunos de los modelos de los servicios hechos.



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.

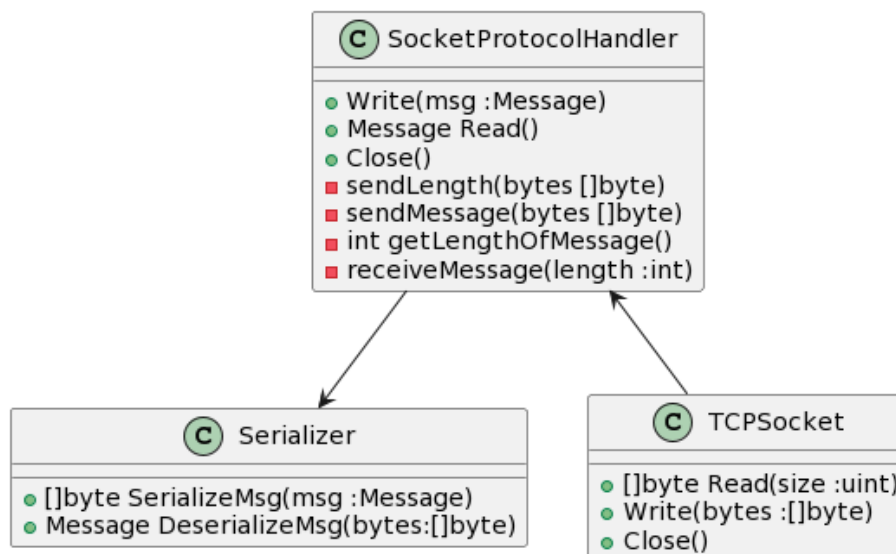
### Protocolo de colas



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.

## 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).

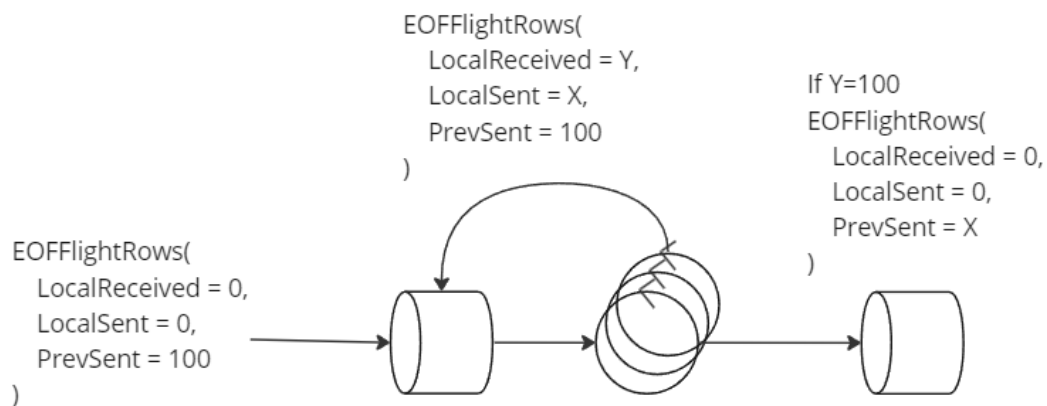
Ya al utilizar las colas de RabbitMQ podemos serializar el acceso a estos recursos y evitar condiciones de carrera, ya que solamente habría uno manejando la escritura a ese archivo.

Dentro de un mismo servicio, los puntos comunes que hay son el envío de alguna señal (por ejemplo de que se terminó de recibir el archivo de aeropuertos). Al estar utilizando Go podemos aprovechar las *goroutines* y adherir de forma sencilla al principio de '*Don't communicate by sharing memory; share memory by communicating*', con canales para comunicar esta señal.

## 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 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 cuantos se mandó a esa etapa con valores recibidos y enviados en cero.
2. Un controlador recibe el mensaje y sabe que ya no van a venir más datos, suma al mensaje lo que recibió (leyó de la cola) y lo que pasó a la siguiente etapa. Con esto lo vuelve a reencolar y pasará al siguiente consumidor en la asignación de RabbitMQ.
3. Otro controlador tomará el mensaje y repetirá el proceso hasta que sea el último, en cuyo caso el total de mensajes recibidos será igual al enviado por la etapa anterior, esto quiere decir que se leyó todo y se puede pasar a la siguiente etapa del EOF actualizando lo enviado.

### Pendientes

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

- Realizar mayor cantidad de tests unitarios de los componentes.
- Optimizaciones en puntos donde se trabajan con archivos para que manejen algunos recursos en memoria y dado un tiempo o cantidad de datos se escriban a disco.
- Quedan pendientes la realización de algunos refactors y limpieza de código.
- 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.
- Para mejorar la escalabilidad horizontal del handler del ejercicio 4 se requeriría desacoplar los Dispatchers del contenedor. Esto implicaría la utilización de algún Exchange en nuestro Middleware de Rabbit cuya Routing Key tenga que ver con el número del contenedor hacia el cuál enviar y que éstos suscriban con el Routing Key específico. De esa manera, los contenedores tendrán todos trayectos independientes internamente, y se podrá escalar la cantidad sin ningún inconveniente de inconsistencias de datos.