Movies Analysis

Integrantes

- 107923 - Martín Juan Cwikla - mcwikla@fi.uba.ar
- 108634 - Maximiliano Nicolas Otero Silvera - motero@fi.uba.ar
- 110119 - Máximo Gismondi - magismondi@fi.uba.ar

Materia

- Facultad de Ingeniería de la Universidad de Buenos Aires
- Sistemas Distribuidos I [75.74/TA050]
- Cátedra Roca
- 1er. cuatrimestre 2025

Corrector

- Manuel Reberendo - mreberendo@fi.uba.ar

Entrega

- Fecha de entrega: 26/06/2025

Tabla de contenidos

Alcance

El objetivo de este proyecto es diseñar y desarrollar un sistema distribuido el cual procesa diversos archivos enviados por el cliente para otorgarle múltiples resultados en base a la información brindada. La información que puede ser enviada por el cliente son reseñas de películas, sus ratings y casting.

Arquitectura del Software

Este sistema implementa una arquitectura cliente - servidor con las siguientes responsabilidades:

Cliente

Envía los archivos sin modificar al servidor utilizando la ClientLibrary la cual es parte del middleware.

Servidor

El servidor recibe a través de TCP los archivos enviados por el cliente, genera las queries y las reparte entre diversos workers mediante RabbitMQ. Cada servidor puede atender múltiples clientes a la vez ya que dedica un proceso a cada nuevo cliente.

Middleware

Este componente actúa como intermediario entre el cliente, el servidor y las distintas entidades del sistema, desacoplando la lógica de comunicación. Implementa un sistema de mensajería basado en colas que permite distribuir consultas y conjuntos de datos (datasets) entre los distintos módulos del sistema.

Metas de la arquitectura

SIGTERM handling

- Se debe proveer graceful quit frente a señales SIGTERM

Escalabilidad

- El sistema debe estar optimizado para entornos multicomputadoras y además debe soportar el incremento de los elementos de cómputo para escalar los volúmenes de información a procesar

Load balancing

- El middleware debe distribuir la carga de manera inteligente con tal de no saturar un único worker mientras los demás se encuentran starving

Limitaciones de la arquitectura

Queries

- Solo se puede procesar las 5 queries esperadas

Dataset

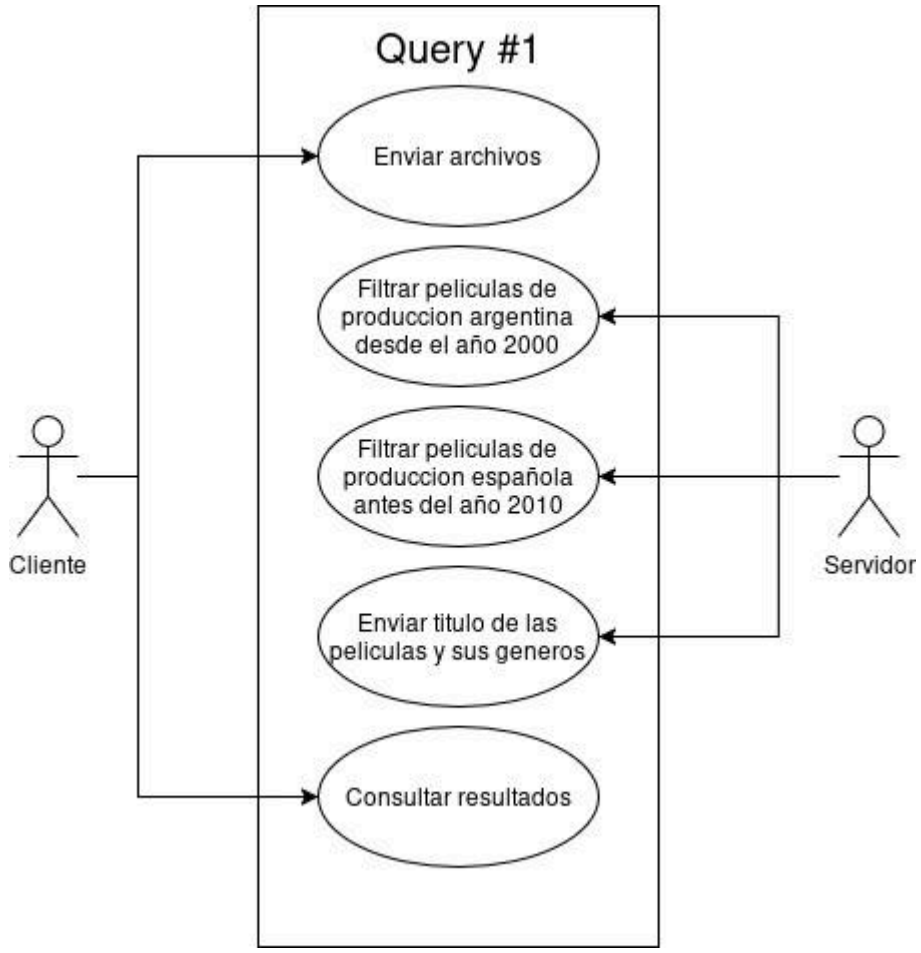
- Se espera una cierta estructura en el formato de cada dataset tanto en el inicio de la request como en cada etapa de una query en particular

Escenarios

Casos de uso

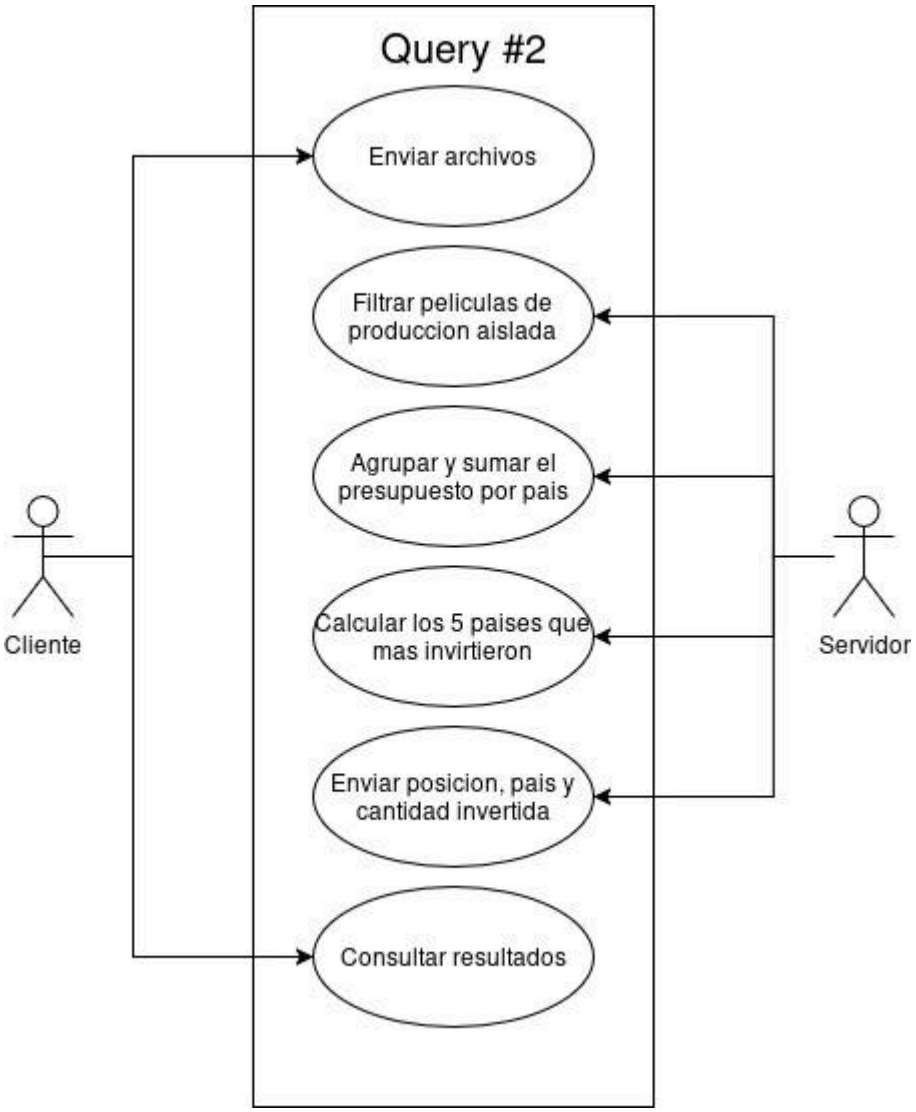
Query 1

- **Descripción**
  - Obtener películas y sus géneros de los años 2000 con producción Argentina y Española.
- **Casos alternativos**
  - Return lista vacía
    - Si no hay películas que cumplan las condiciones se retorna una lista vacía
  - Dataset con error
    - Si alguna de las líneas del dataset contiene un formato no esperado, la línea se ignora. Si todas las líneas son erróneas, se retorna una lista vacía.



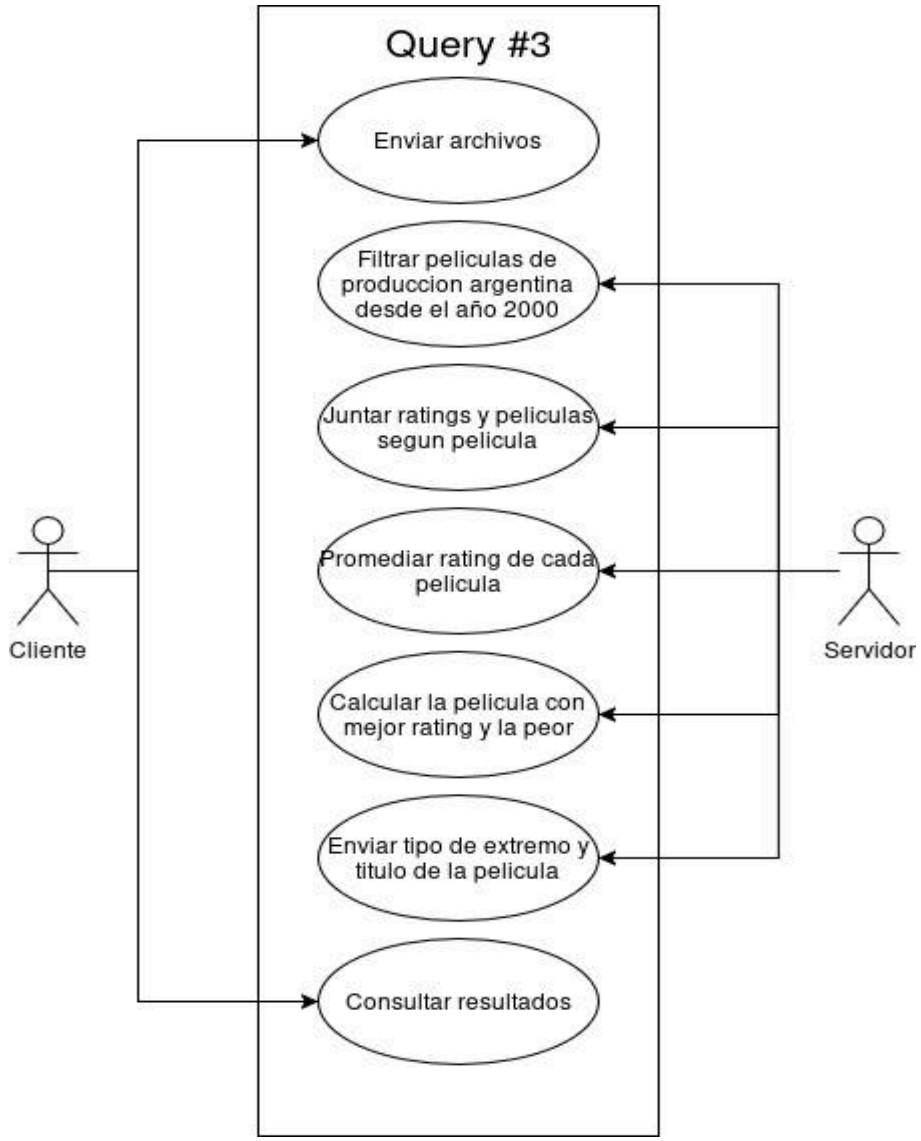
Query 2

- **Descripción**
  - Obtener el top 5 de países que más dinero han invertido en producciones sin colaborar con otros países.
- **Casos alternativos**
  - Return top N, 0 <= N < 5
    - Si no hay suficientes países como para conformar un top 5 pero si un top N (0 <= N < 5), se retorna un top de N países
  - Dataset con error
    - Si alguna de las líneas del dataset contiene un formato no esperado, la línea se ignora. Si todas las líneas son erróneas, se retorna una lista vacía.



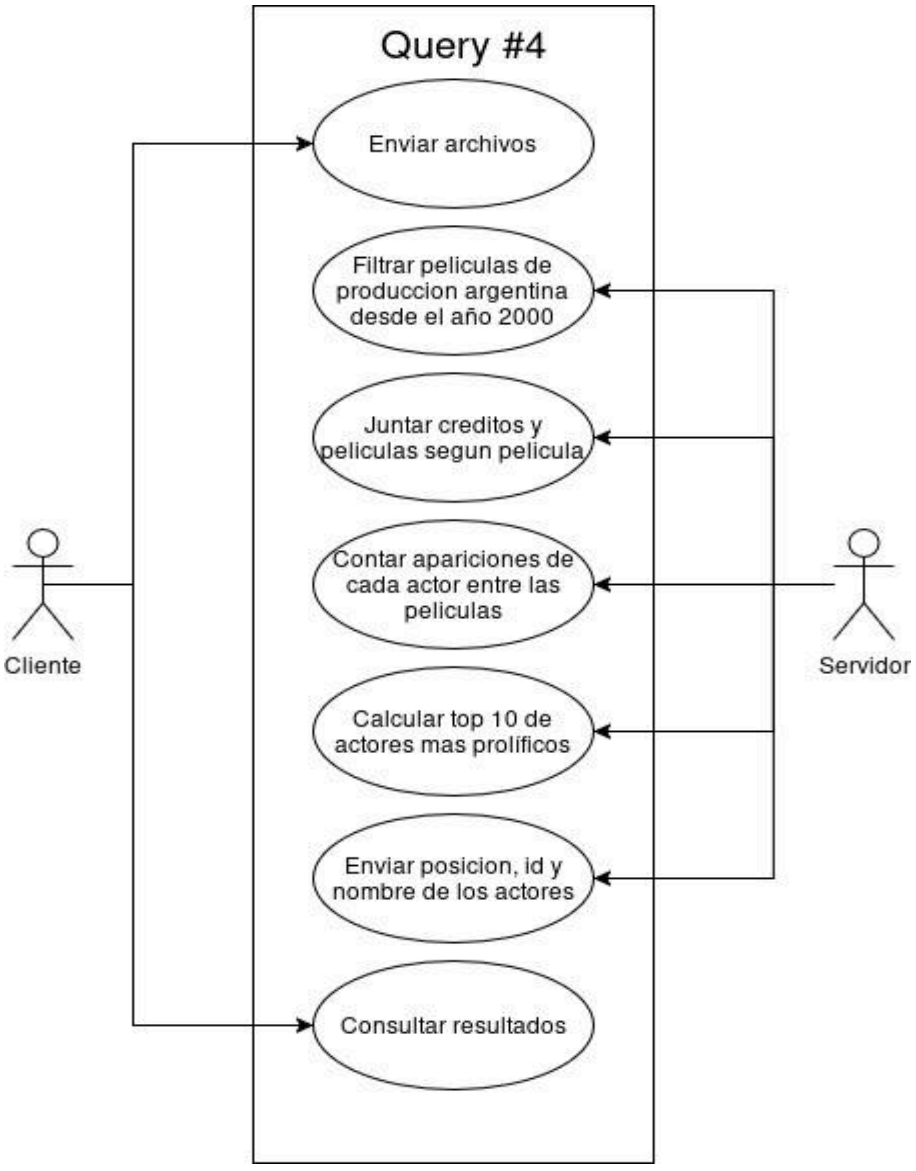
Query 3

- **Descripción**
  - Obtener película de producción Argentina estrenada a partir del 2000, con mayor y con menor promedio de rating.
- **Casos alternativos**
  - Return vacío
    - Si no hay ratings de películas que cumplan las condiciones se retorna MAX, null, null; MIN, null, null
  - Mismo return
    - Si una única película cumple las condiciones se retorna MAX, value “title-example”; MIN, value, “title-example”
  - Dataset con error
    - Si alguna de las líneas del dataset contiene un formato no esperado, la línea se ignora. Si todas las líneas son erróneas, se retorna una lista vacía.



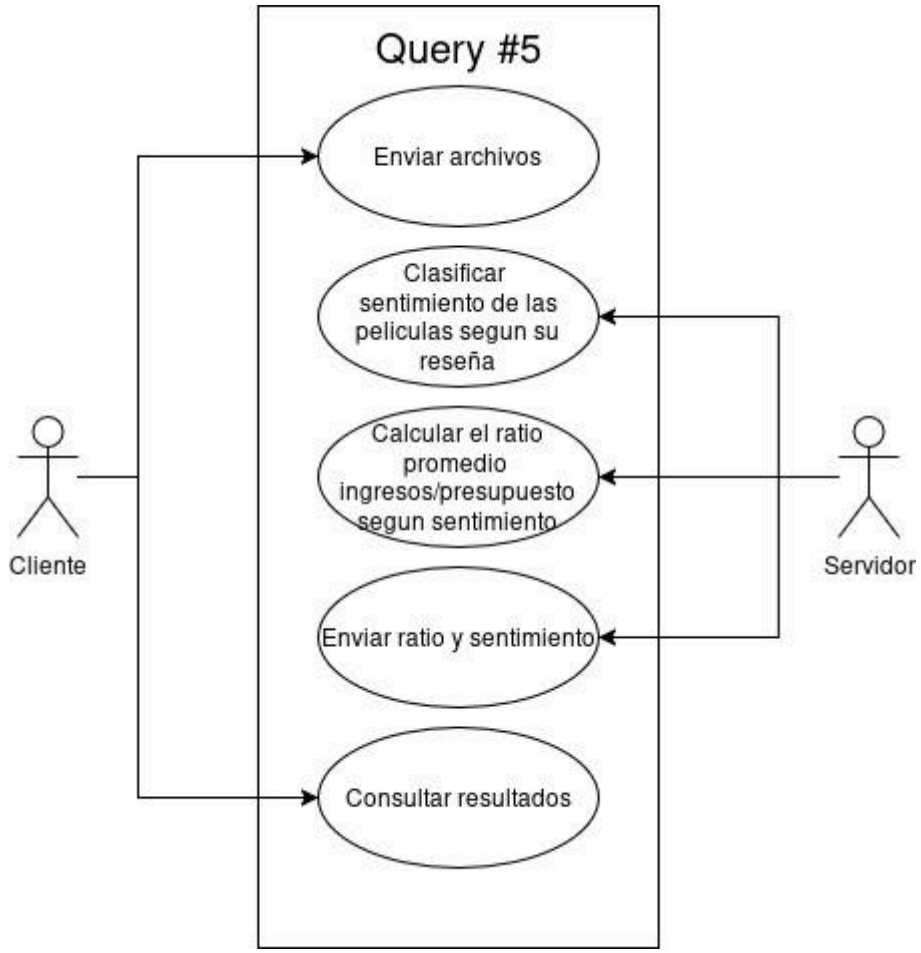
Query 4

- **Descripción**
  - Obtener el top 10 de actores con mayor participación en películas de producción Argentina con fecha de estreno posterior al 2000.
- **Casos alternativos**
  - Return top N, 0 <= N < 5
    - Si no hay suficientes participaciones de actores en películas como para conformar un top 5 pero si un top N (0 <= N < 5), se retorna un top de N países
  - Dataset con error
    - Si alguna de las líneas del dataset contiene un formato no esperado, la línea se ignora. Si todas las líneas son erróneas, se retorna una lista vacía.



Query 5

- **Descripción**
  - Obtener el promedio de la tasa ingreso/presupuesto de películas con overview de sentimiento positivo vs. sentimiento negativo.
- **Casos alternativos**
  - Return vacío
    - Si no hay películas se retorna POSITIVE, null; NEGATIVE, null
  - Sentimientos homogéneos
    - Si al clasificar las reseñas de las películas se clasifica un único sentimiento para todas, se retorna el promedio de ese sentimiento y null en el otro caso (e.g. POSITIVE, value; NEGATIVE, null)
  - Dataset con error
    - Si alguna de las líneas del dataset contiene un formato no esperado, la línea se ignora. Si todas las líneas son erróneas, se retorna una lista vacía.



Vista lógica

Diagrama de Clases

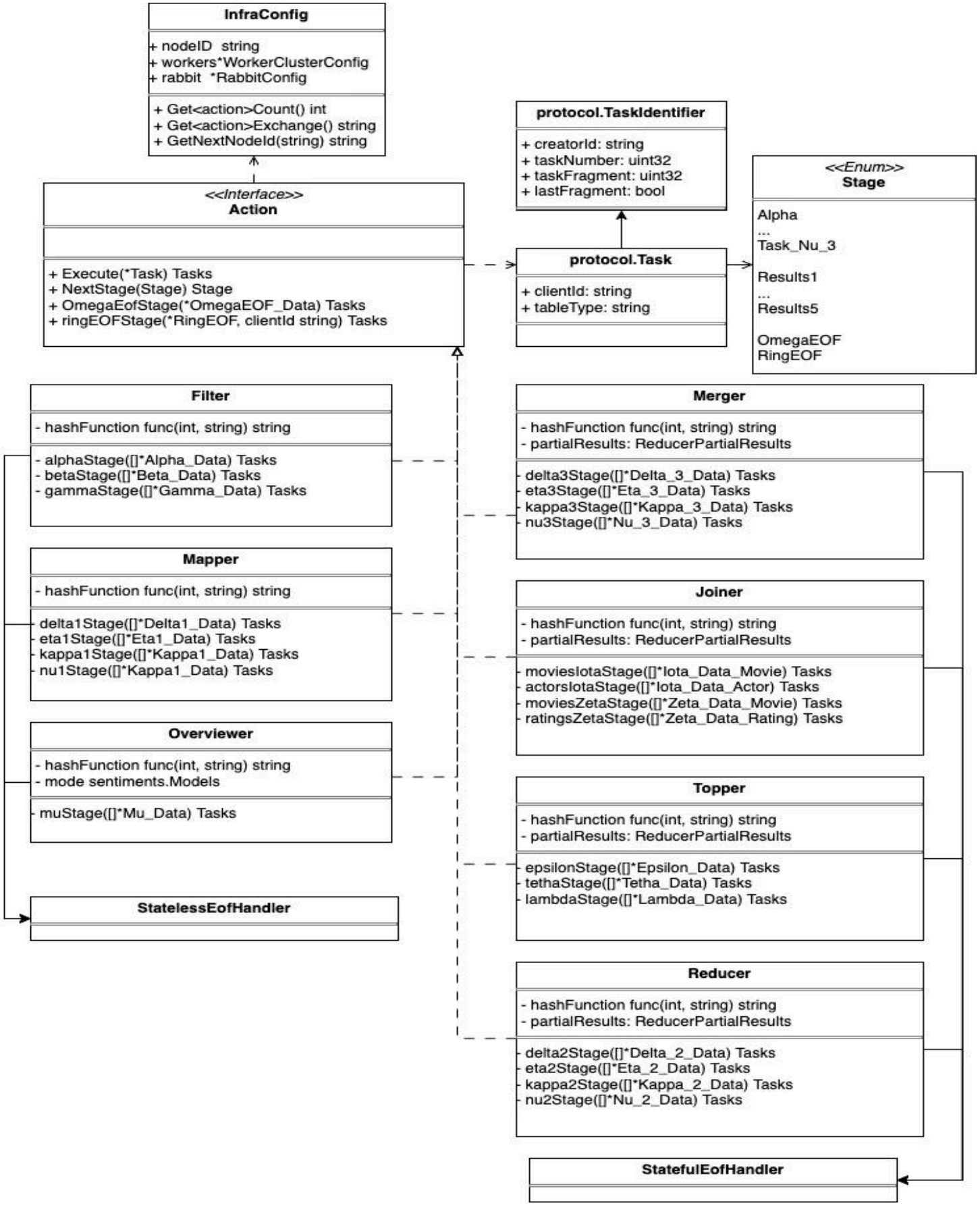
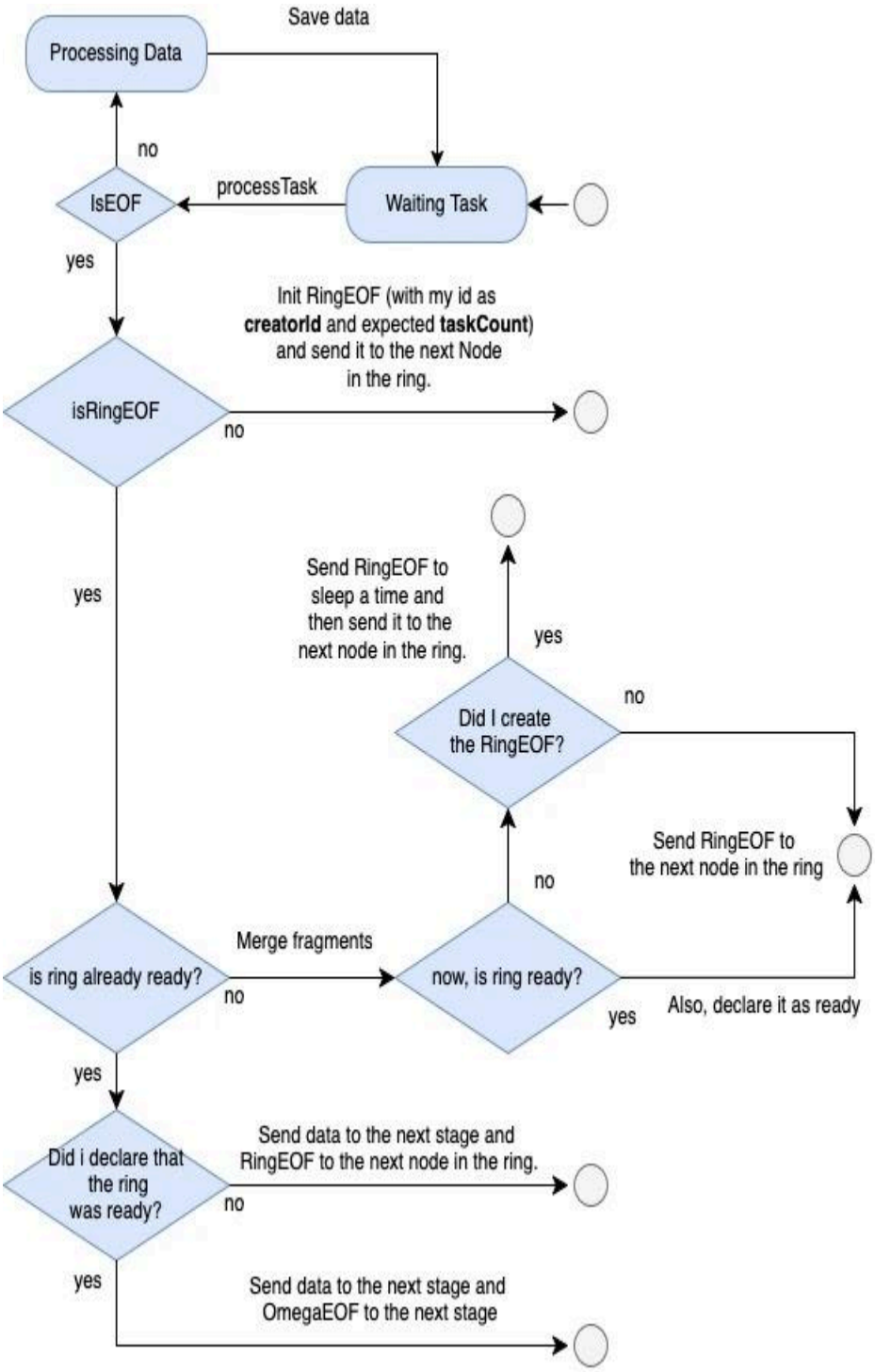

InfraConfig
+ nodeID string
+ workers*WorkerClusterConfig
+ rabbit *RabbitConfig
+ Get<action>Count() int
+ Get<action>Exchange() string
+ GetNextNodeId(string) string
<<Interface>>
Action
+ Execute(*Task) Tasks
+ NextStage(Stage) Stage
+ OmegaEofStage(*OmegaEOF_Data) Tasks
+ ringEOFStage(*RingEOF, clientId string) Tasks
protocol.TaskIdentifier
+ creatorId: string
+ taskNumber: uint32
+ taskFragment: uint32
+ lastFragment: bool
protocol.Task
+ clientId: string
+ tableType: string
<<Enum>>
Stage
Alpha
...
Task_Nu_3
Results1
...
Results5
OmegaEOF
RingEOF
Filter
- hashFunction func(int, string) string
- alphaStage([]*Alpha_Data) Tasks
- betaStage([]*Beta_Data) Tasks
- gammaStage([]*Gamma_Data) Tasks
Mapper
- hashFunction func(int, string) string
- delta1Stage([]*Delta1_Data) Tasks
- eta1Stage([]*Eta1_Data) Tasks
- kappa1Stage([]*Kappa1_Data) Tasks
- nu1Stage([]*Kappa1_Data) Tasks
Overviewer
- hashFunction func(int, string) string
- mode sentiments.Models
- muStage([]*Mu_Data) Tasks
StatelessEofHandler
Merger
- hashFunction func(int, string) string
- partialResults: ReducerPartialResults
- delta3Stage([]*Delta_3_Data) Tasks
- eta3Stage([]*Eta_3_Data) Tasks
- kappa3Stage([]*Kappa_3_Data) Tasks
- nu3Stage([]*Nu_3_Data) Tasks
Joiner
- hashFunction func(int, string) string
- partialResults: ReducerPartialResults
- moviesIotaStage([]*Iota_Data_Movie) Tasks
- actorsIotaStage([]*Iota_Data_Actor) Tasks
- moviesZetaStage([]*Zeta_Data_Movie) Tasks
- ratingsZetaStage([]*Zeta_Data_Rating) Tasks
Topper
- hashFunction func(int, string) string
- partialResults: ReducerPartialResults
- epsilonStage([]*Epsilon_Data) Tasks
- tethaStage([]*Tetha_Data) Tasks
- lambdaStage([]*Lambda_Data) Tasks
Reducer
- hashFunction func(int, string) string
- partialResults: ReducerPartialResults
- delta2Stage([]*Delta_2_Data) Tasks
- eta2Stage([]*Eta_2_Data) Tasks
- kappa2Stage([]*Kappa_2_Data) Tasks
- nu2Stage([]*Nu_2_Data) Tasks
StatefulEofHandler

Diagrama de estados

EOF

Representación de los estados en la recepción de Tareas y End Of Files, procesamiento de información, guardado de datos y pasaje a la siguiente etapa para workers Stateful y Stateless.

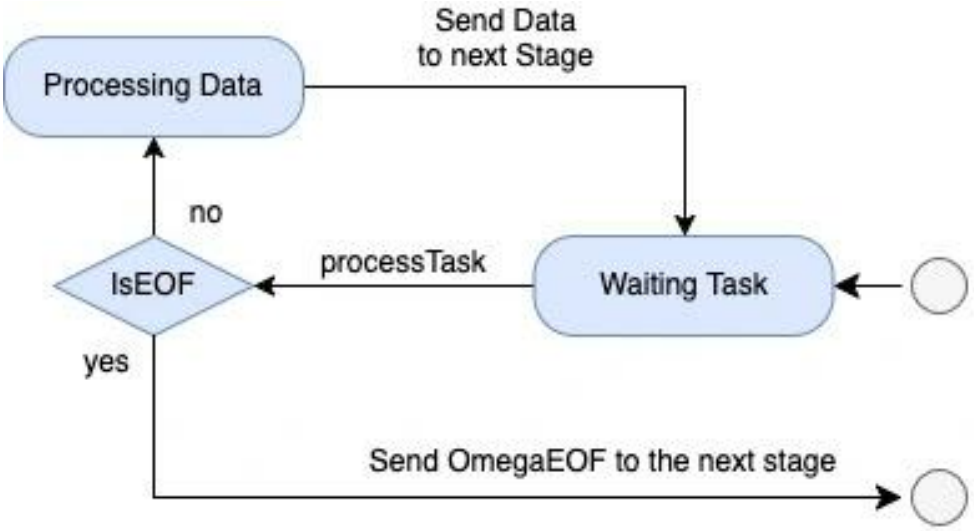
Stateful



Casos alternativos:

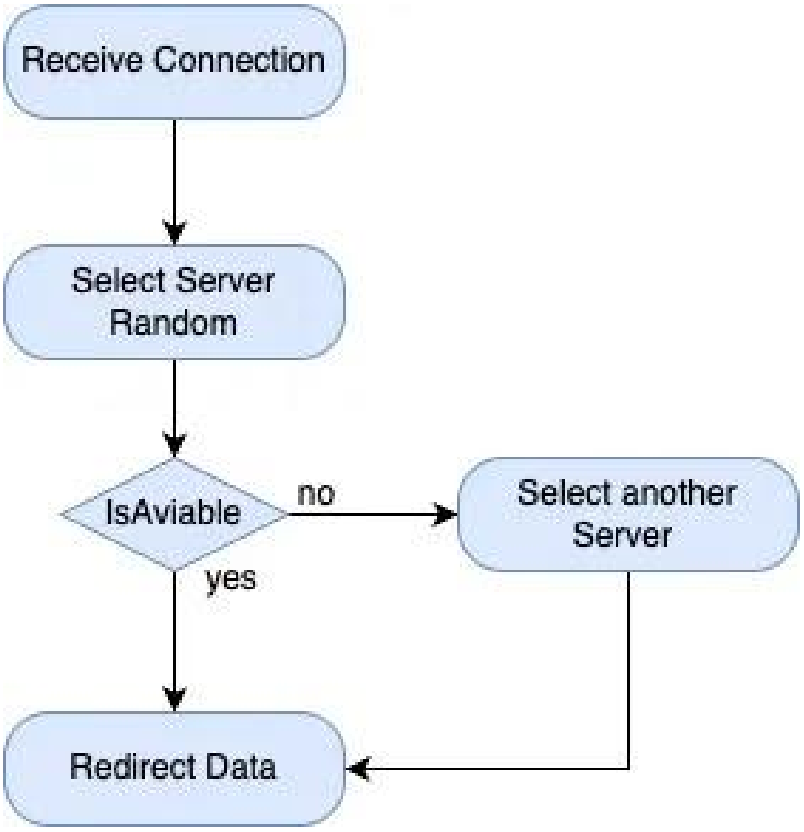
- **Topper:** Para el topper, cuando se refiere al próximo nodo, se refiere a él mismo.

Stateless



Proxy

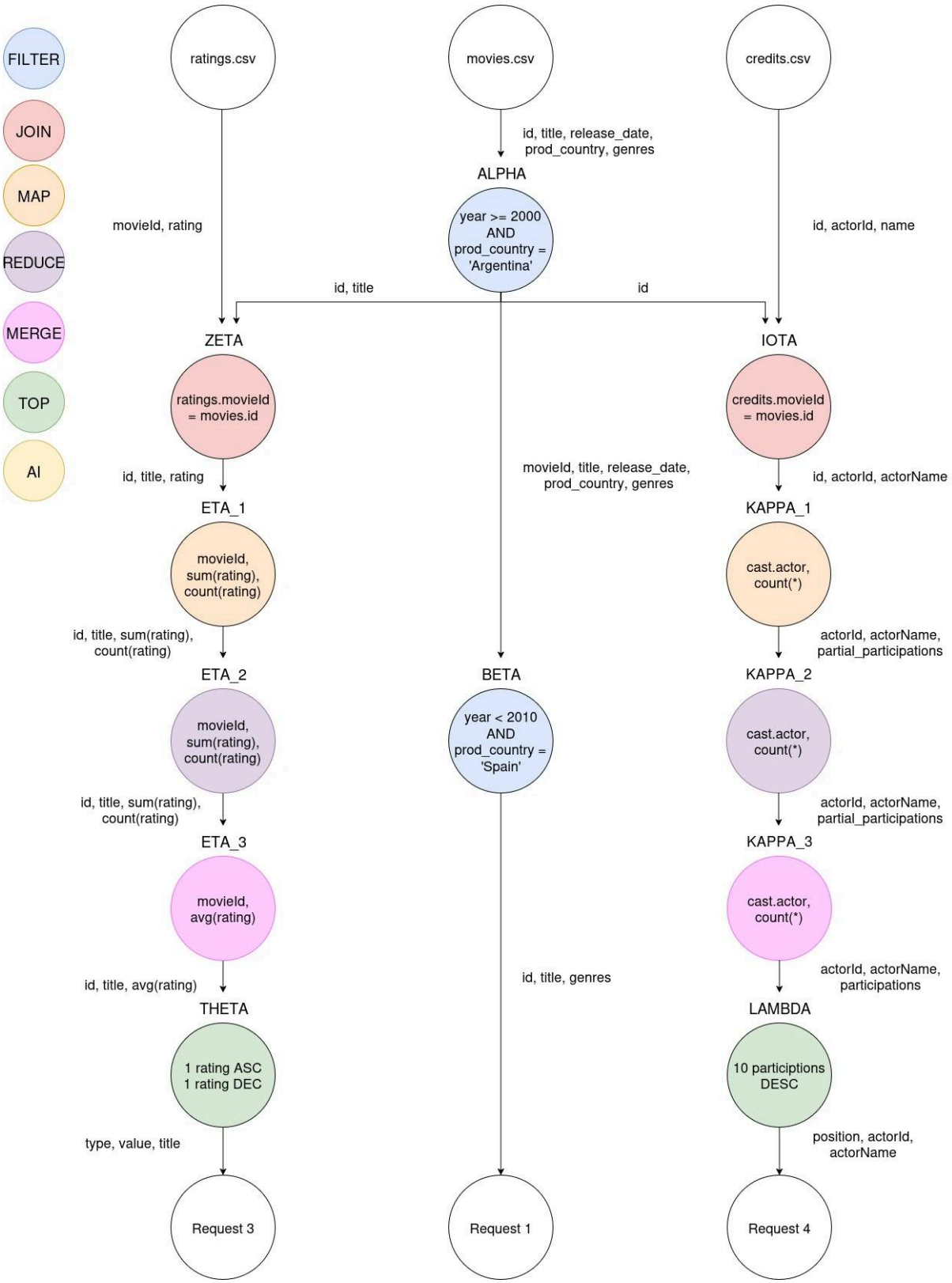
Lógica del proxy para determinar el próximo al que conectarse.

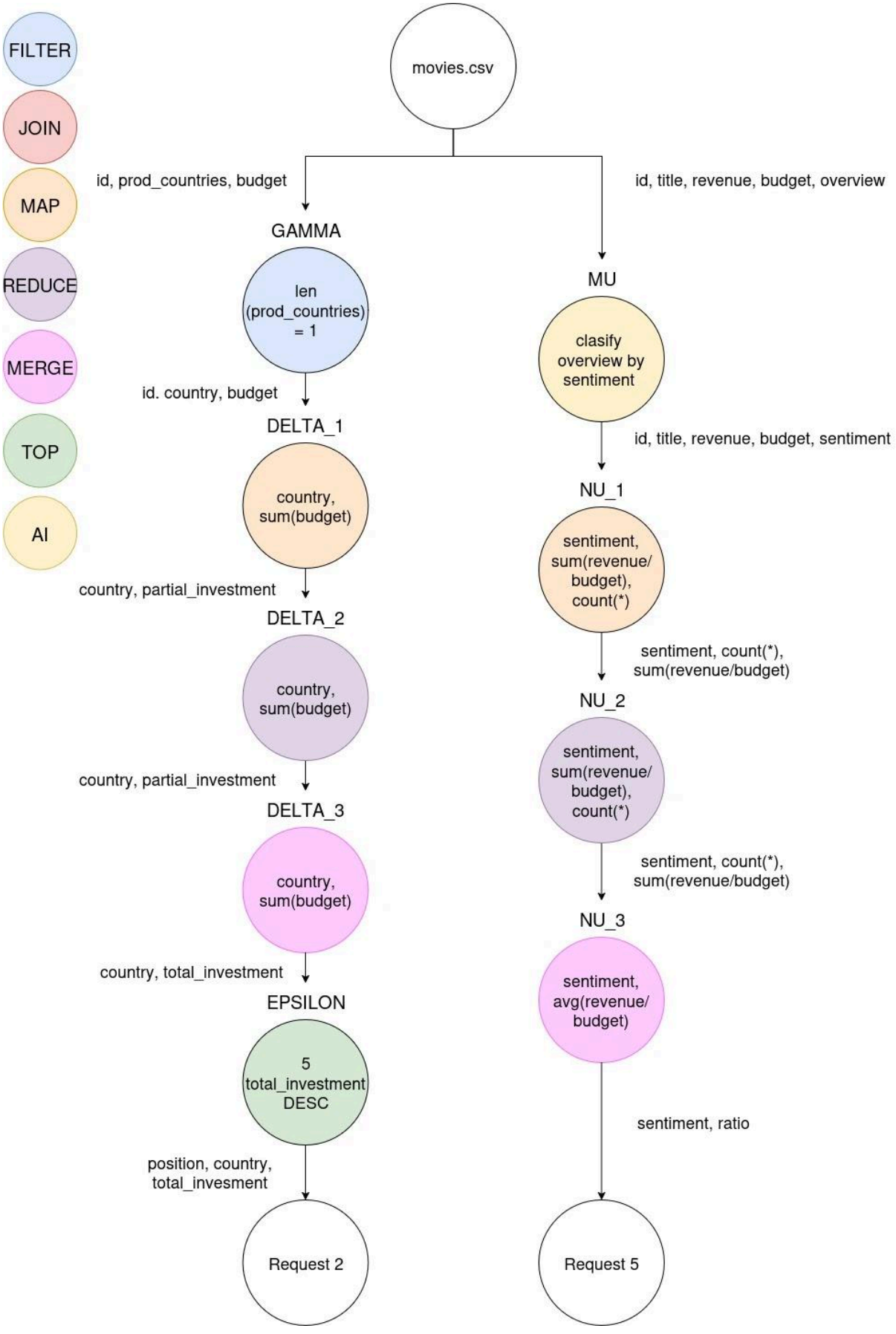


Flujo de mensajes

Directed Acyclic Graph (DAG)

En este diagrama podemos ver el flujo que toman las 5 queries en cada una de sus etapas, mostrando que acción se realiza y qué acciones se comparten entre queries.


FILTER
JOIN
MAP
REDUCE
MERGE
TOP
AI
ratings.csv
movies.csv
credits.csv
id, title, release_date, prod_country, genres
ALPHA
year >= 2000 AND prod_country = 'Argentina'
movieId, rating
id, actorId, name
id, title
id
ZETA
IOTA
ratings.movieId = movies.id
credits.movieId = movies.id
id, title, rating
movieId, title, release_date, prod_country, genres
id, actorId, actorName
ETA_1
KAPPA_1
movieId, sum(rating), count(rating)
cast.actor, count(*)
id, title, sum(rating), count(rating)
actorId, actorName, partial_participations
ETA_2
BETA
KAPPA_2
movieId, sum(rating), count(rating)
year < 2010 AND prod_country = 'Spain'
cast.actor, count(*)
id, title, sum(rating), count(rating)
actorId, actorName, partial_participations
ETA_3
KAPPA_3
movieId, avg(rating)
cast.actor, count(*)
id, title, avg(rating)
id, title, genres
actorId, actorName, participations
THETA
LAMBDA
1 rating ASC 1 rating DEC
10 participtions DESC
type, value, title
position, actorId, actorName
Request 3
Request 1
Request 4

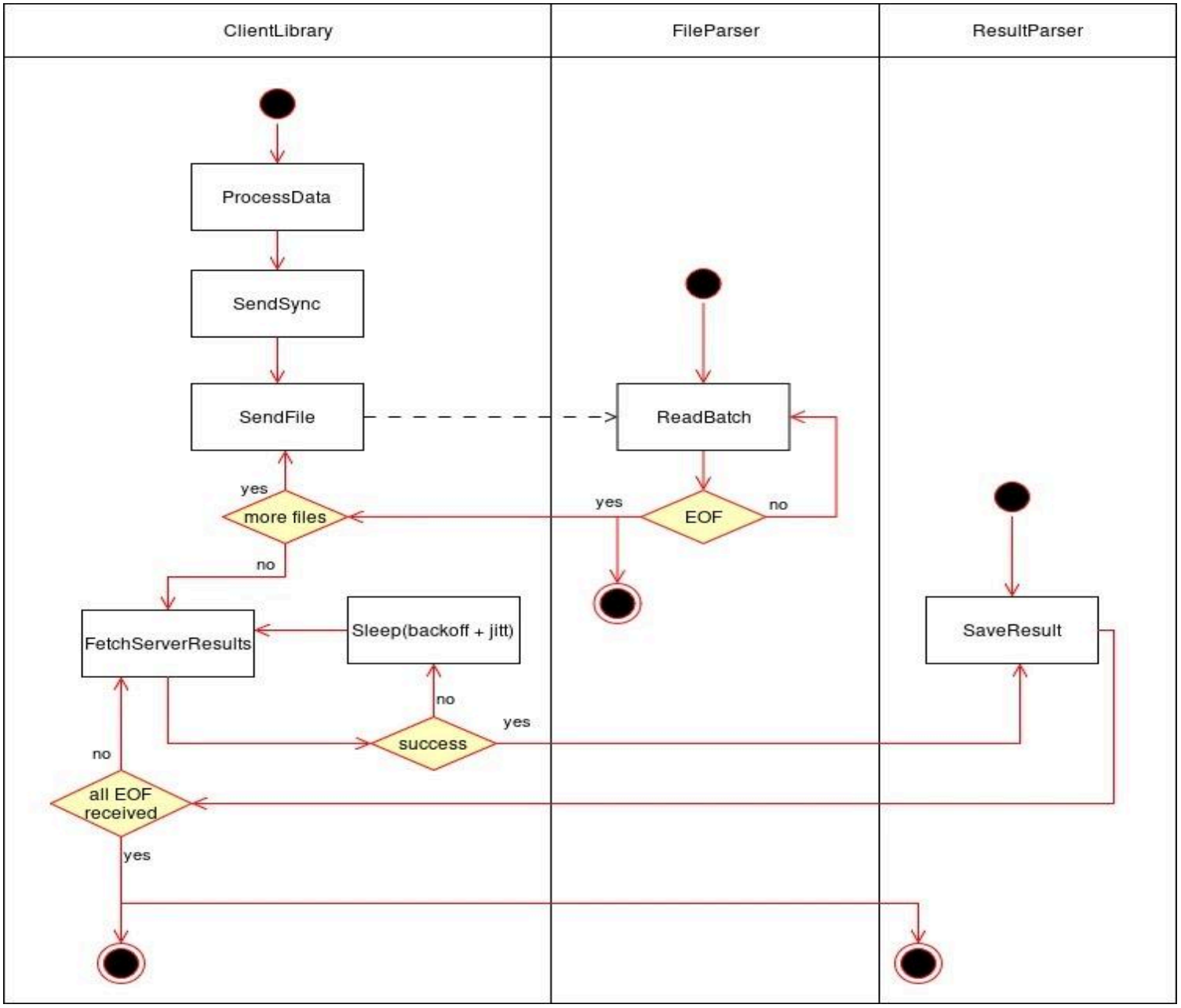

FILTER
JOIN
MAP
REDUCE
MERGE
TOP
AI
movies.csv
id, prod_countries, budget
id, title, revenue, budget, overview
GAMMA
len (prod_countries) = 1
id. country, budget
DELTA_1
country, sum(budget)
country, partial_investment
DELTA_2
country, sum(budget)
country, partial_investment
DELTA_3
country, sum(budget)
country, total_investment
EPSILON
5 total_investment DESC
position, country, total_invesment
Request 2
MU
clasify overview by sentiment
id, title, revenue, budget, sentiment
NU_1
sentiment, sum(revenue/ budget), count(*)
sentiment, count(*), sum(revenue/budget)
NU_2
sentiment, sum(revenue/ budget), count(*)
sentiment, count(*), sum(revenue/budget)
NU_3
sentiment, avg(revenue/ budget)
sentiment, ratio
Request 5

Vista de proceso

Actividades

Flujo de un cliente

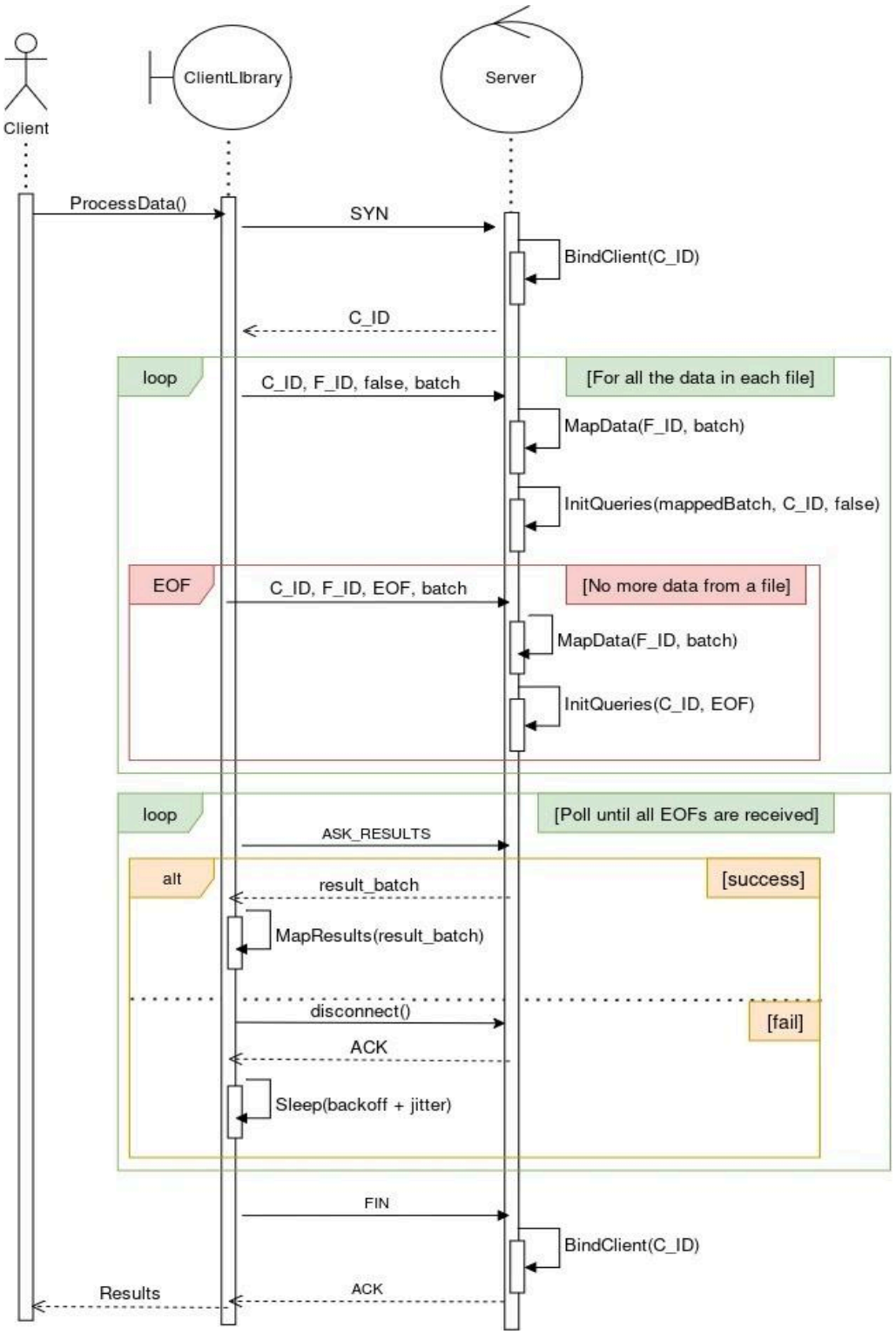
En este diagrama podemos ver como un cliente se conecta al servidor, lee la información y la envía, y hace polling al servidor esperando los resultados de las queries.



Secuencia

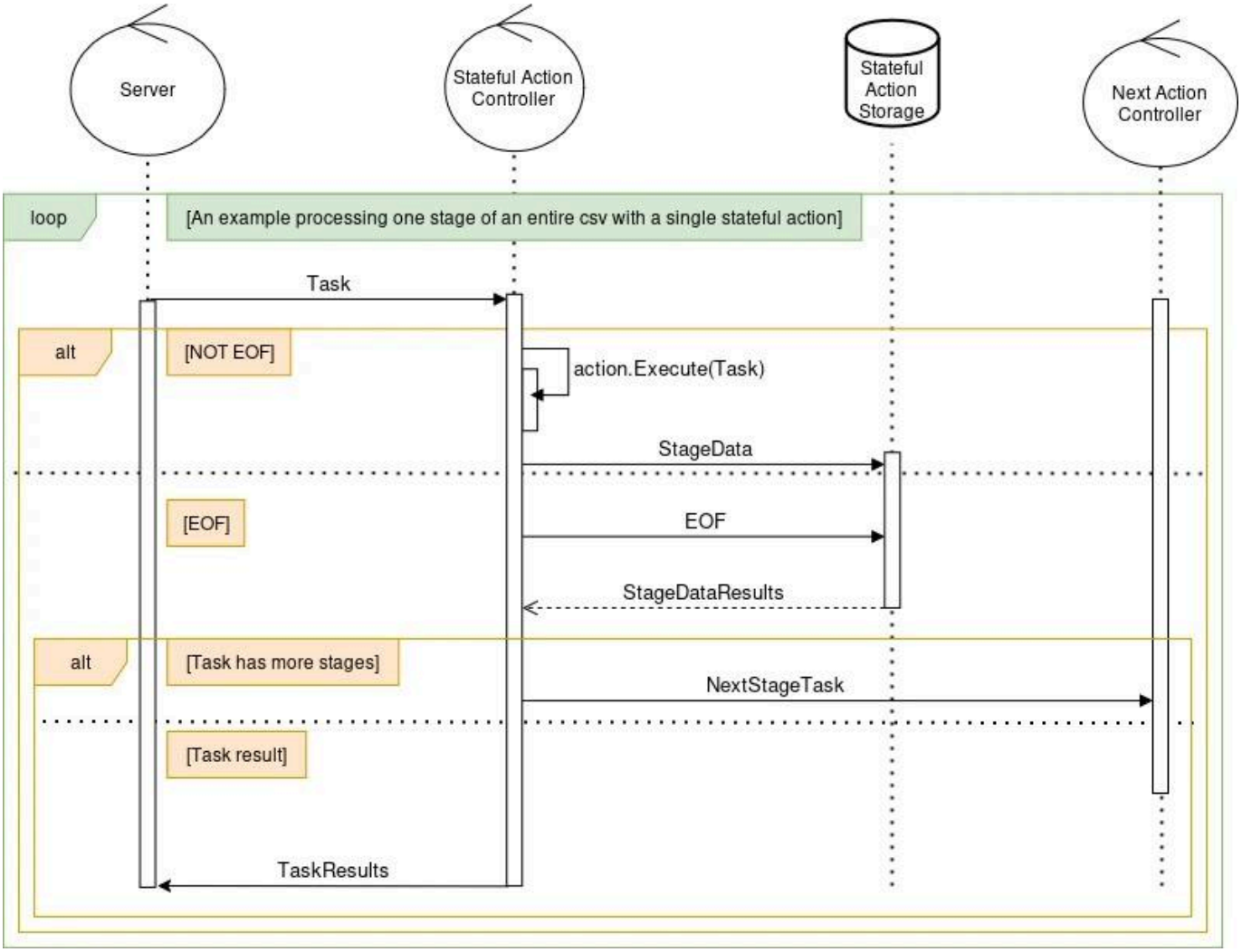
Flujo general

En este primer diagrama se muestra el flujo típico de una request completa (el cliente envía los 3 datasets esperados). Se simplifica el funcionamiento de los workers en la función InitQueries() del servidor y del proxy ya que funciona como un pasamanos.



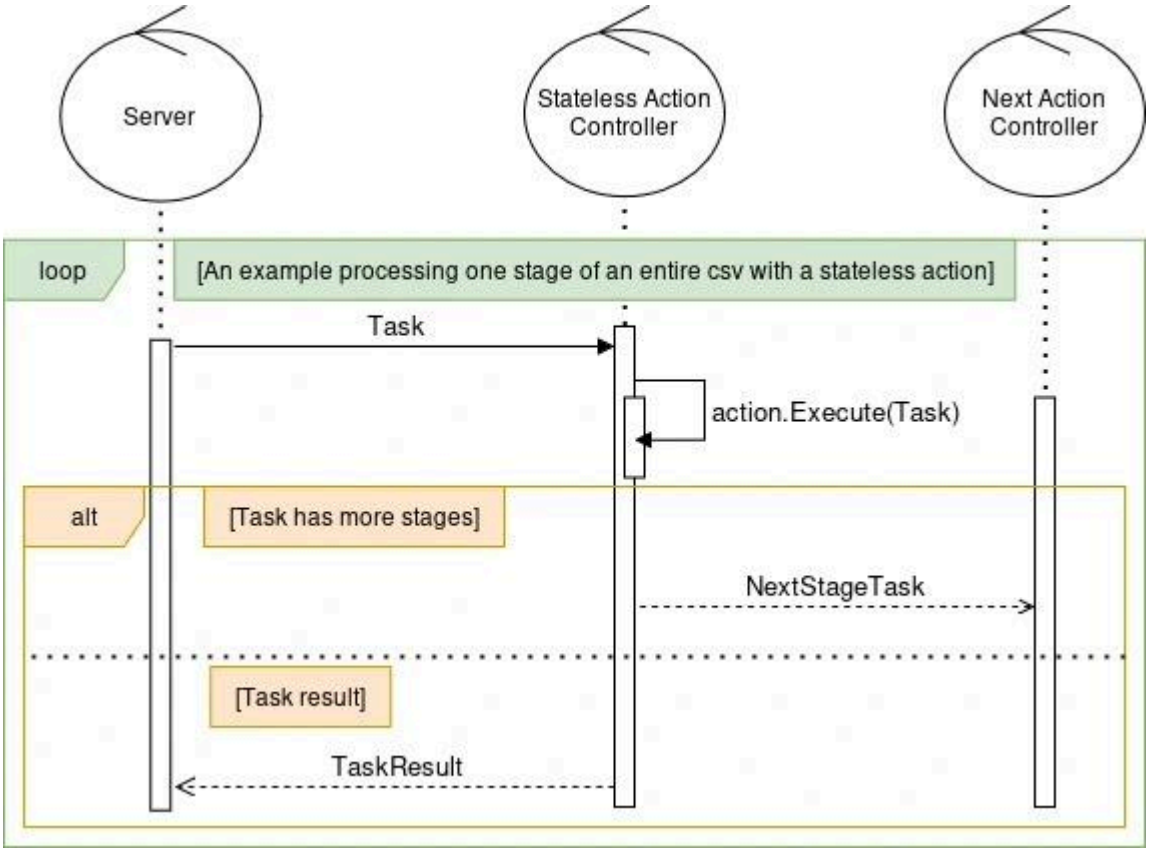
Flujo con un worker stateful

En este diagrama mostramos como sería el caso a modo de ejemplo en el cual se procesa un csv entero con el mismo worker y acción la cual es stateful.



Flujo con un worker stateless

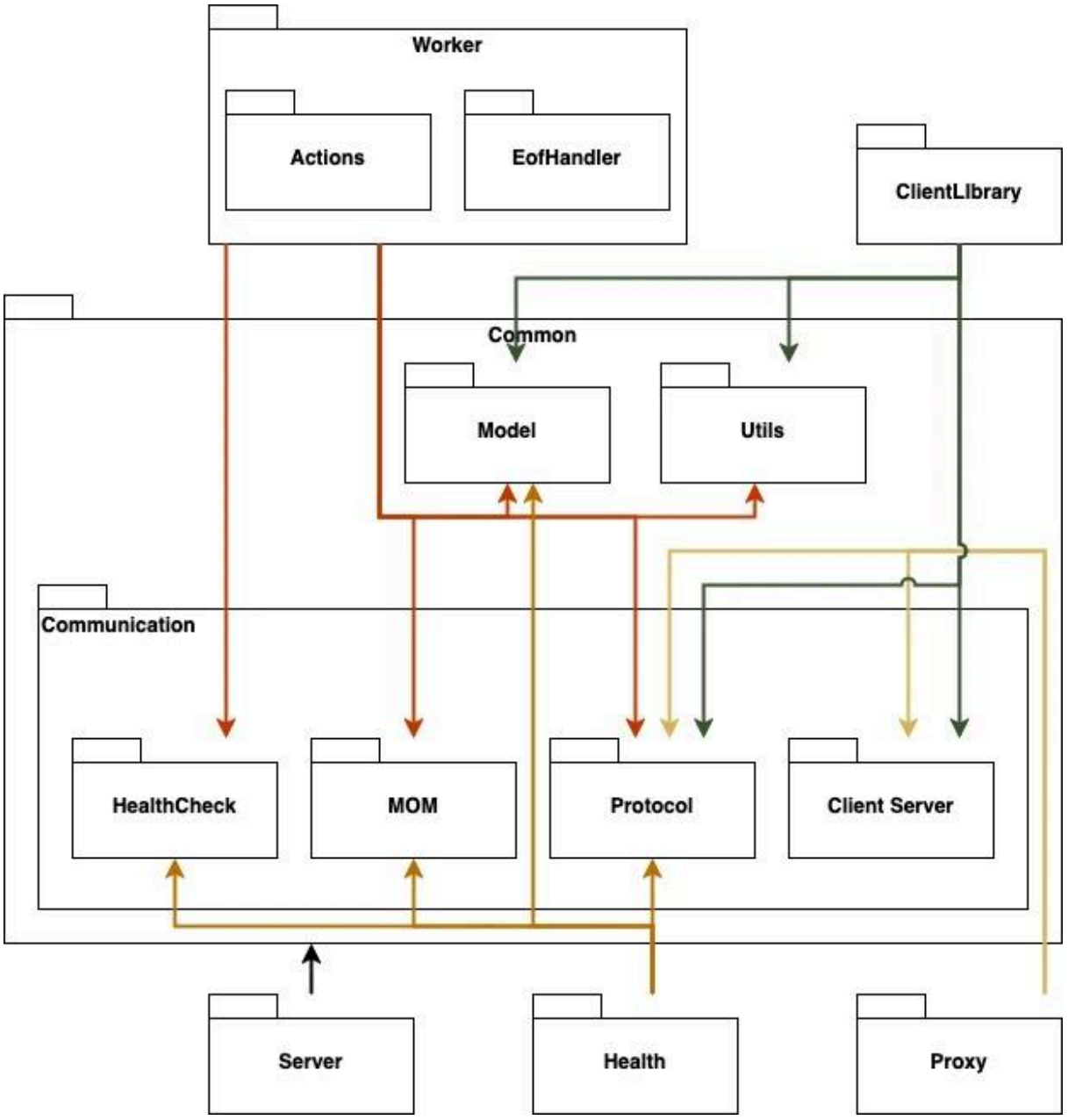
Por último, en este diagrama mostramos como sería el caso a modo de ejemplo en el cual se procesa un csv entero con el mismo worker y acción la cual es stateless.



Vista de desarrollo

Paquetes

En este diagrama mostramos todos los paquetes que componen nuestro sistema, y la relación que hay entre ellos. Es importante notar que las principales entidades tienen acceso a el paquete common en donde se van a encontrar todas las funcionalidades compartidas, dentro de ellas la de comunicación.



Protocolo

El protocolo fue implementado utilizando Protocol Buffers

```
message Task {
    oneof Stage {
        Alpha alpha = 1;
        Beta beta = 2;
        Gamma gamma = 3;
        Delta_1 delta_1 = 4;
        Delta_2 delta_2 = 5;
        Delta_3 delta_3 = 6;
        Epsilon epsilon = 7;
        Zeta zeta = 8;
        Eta_1 eta_1 = 9;
        Eta_2 eta_2 = 10;
        Eta_3 eta_3 = 11;
        Theta theta = 12;
        Iota iota = 13;
        Kappa_1 kappa_1 = 14;
        Kappa_2 kappa_2 = 15;
        Kappa_3 kappa_3 = 16;
        Lambda lambda = 17;
        Mu mu = 18;
        Nu_1 nu_1 = 19;
        Nu_2 nu_2 = 20;
        Nu_3 nu_3 = 21;
        Result1 result1 = 22;
        Result2 result2 = 23;
        Result3 result3 = 24;
        Result4 result4 = 25;
        Result5 result5 = 26;
        OmegaEOF omegaEOF = 27;
        RingEOF ringEOF = 28;
    }

    string clientId = 29;
    TaskIdentifier taskIdentifier = 30;
    string tableType = 31;
}
```

En cuanto al servidor - workers, los mensajes que se pueden enviar son los siguientes:

```
message TaskIdentifier {
    string creatorId = 1;
    uint32 taskNumber = 2;
    uint32 taskFragmentNumber = 3;
    bool lastFragment = 4;
}
```

Siendo cada una de las etapas las listadas en el DAG, a excepción de OmegaEOF y RingEOF que contienen los siguientes elementos:

```
message OmegaEOF {
    message Data {
        string stage = 1;
        string worker_creator_id = 2;
        string eofType = 3;
    }

    Data data = 1;
}
```

En cuanto al cliente - servidor, los mensajes que se pueden enviar son los siguientes

```
message Message {
    oneof message {
        ClientServerMessage client_server_message = 1;
        ServerClientMessage server_client_message = 2;
    }
}
```

```
enum MessageStatus {
    SUCCESS = 0;
    FAIL = 1;
    PENDING = 2;
}

message BatchAck {
    string batch_id = 1;
    MessageStatus status = 2;
}

message FileEOFAck {
    MessageStatus status = 1;
}

message SyncAck {
    string client_id = 1;
}

message FinishAck {}

message DisconnectAck {}

message ResultsResponse {

    message Result{
        oneof Message {
            Result1 result1 = 1;
            Result2 result2 = 2;
            Result3 result3 = 3;
            Result4 result4 = 4;
            Result5 result5 = 5;
            OmegaEOF omegaEOF = 6;
        };

        TaskIdentifier task_identifier = 7;
    }

    MessageStatus status = 1;
    repeated Result results = 2;
}
```

Siendo ServerClientMessage los mensajes que envía el servidor hacia el cliente.

```
message ServerClientMessage{
    oneof message {
        BatchAck batch_ack = 1;
        FileEOFAck file_eof_ack = 2;
        SyncAck sync_ack = 3;
        ResultsResponse results = 4;
        FinishAck finish_ack = 5;
        DisconnectAck disconnect_ack = 6;
    }
}
```

Para los resultados se utilizaron las etapas de resultado del DAG

Y ClientServerMessage los mensajes que envía el cliente hacia el servidor.

```
enum FileType {
    MOVIES = 0;
    CREDITS = 1;
    RATINGS = 2;
}

message Sync {
    string client_id = 1;
}

message Finish {
    string client_id = 1;
}

message Disconnect {
    string client_id = 1;
}

message Result {
    string client_id = 1;
}

message Batch {
    message Row {
        string data = 1;
    }

    FileType type = 1;
    repeated Row data = 2;
    string client_id = 3;
    uint32 batch_number = 4;
}

message FileEOF {
    FileType type = 1;
    uint32 batch_count = 2;
    string client_id = 3;
}
```

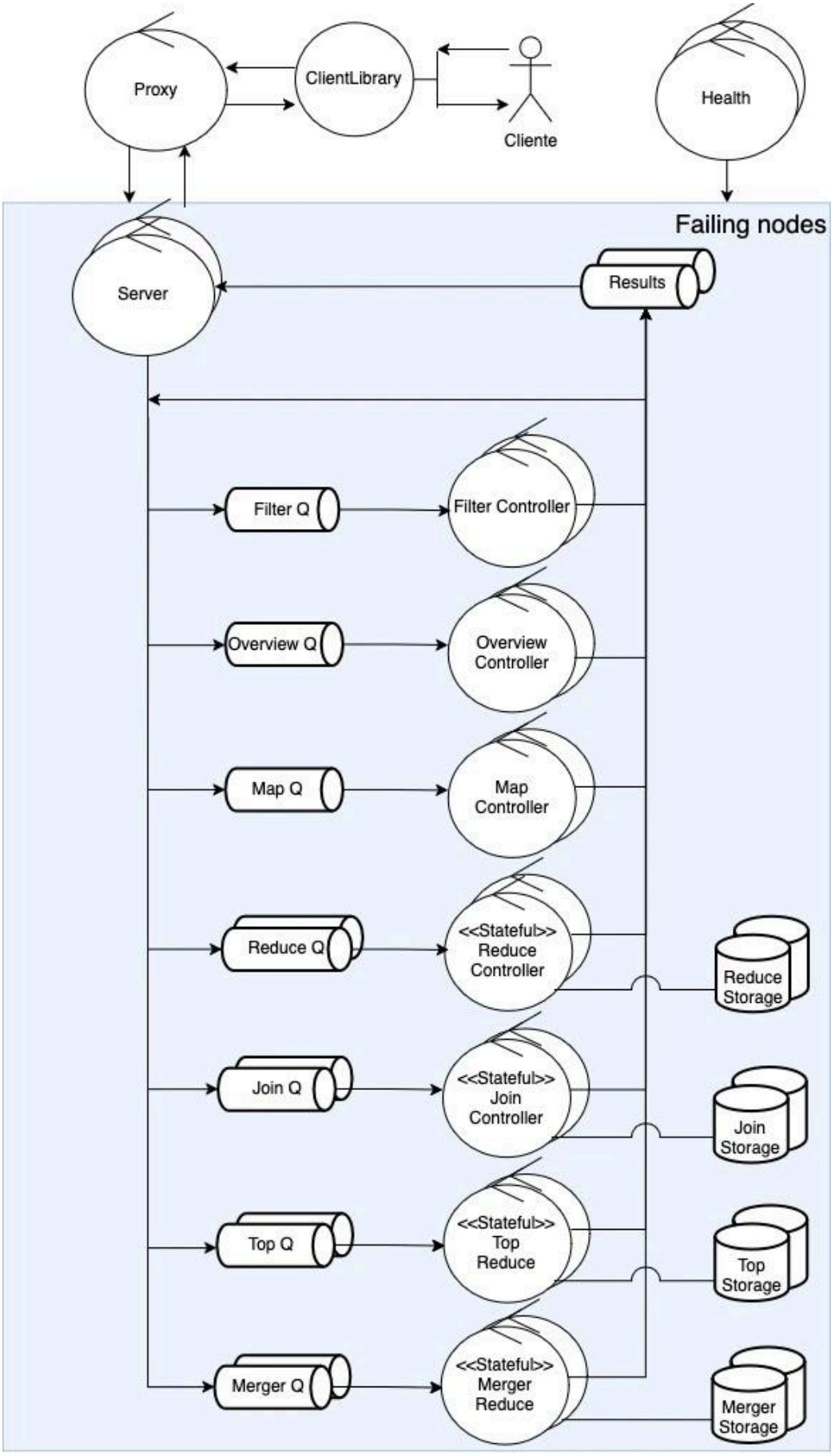
```
message ClientServerMessage{
    oneof message {
        Batch batch = 1;
        FileEOF file_eof = 2;
        Sync sync = 3;
        Finish finish = 4;
        Result result = 5;
        Disconnect disconnect = 6;
    }
}
```

Vista Física

Robustez

En este diagrama mostramos todas las entidades de nuestro sistema:

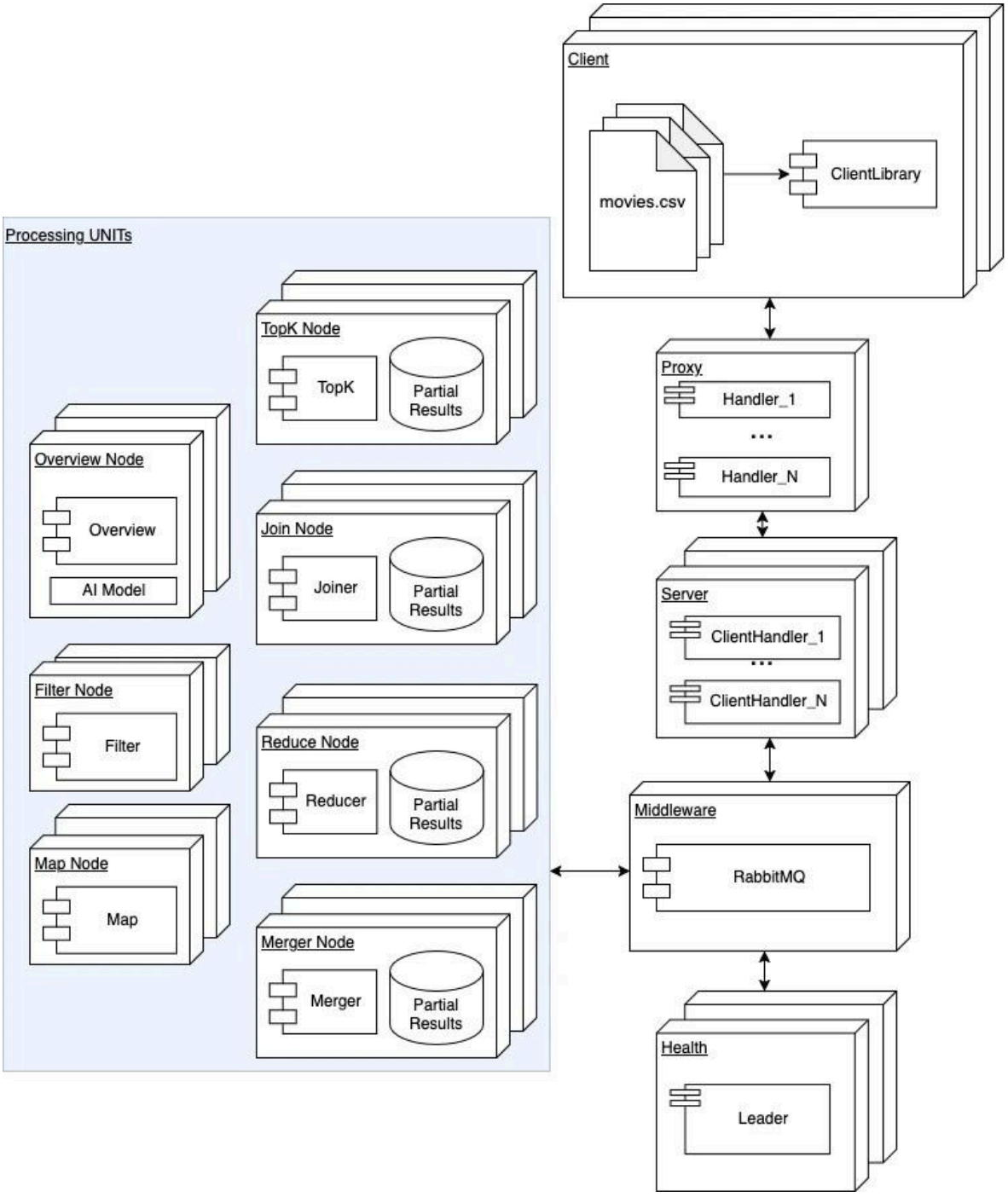
- Cliente: Representa al usuario que interactúa con el sistema.
- Client Library: Interfaz por la cuál el usuario envía los lotes de datos de datos al sistema.
- Proxy: Encargado de redireccionar los datos enviados por un cliente a un servidor. Además de hacer transparente la comunicación al mismo ante fallas.
- Server: Encargado de recibir los lotes de datos (batches) de datos del cliente y filtrar la información irrelevante. Y enviar las respuestas ya procesadas al cliente.
- Stateless Controllers: Consumen tareas desde varias colas, sin distinción de origen. Procesan cada lote de datos de forma independiente y según la lógica definida para cada tipo de solicitud.
- Stateful Controllers: Consumen tareas desde una única cola asignada y mantienen información interna entre ejecuciones, lo que les permite gestionar procesos que requieren persistencia de contexto o seguimiento.
- Health: Mantiene el sistema en funcionamiento ante eventuales fallas en los nodos server o worker.


Proxy
ClientLibrary
Cliente
Health
Failing nodes
Server
Results
Filter Q
Filter Controller
Overview Q
Overview Controller
Map Q
Map Controller
Reduce Q
<<Stateful>> Reduce Controller
Reduce Storage
Join Q
<<Stateful>> Join Controller
Join Storage
Top Q
<<Stateful>> Top Reduce
Top Storage
Merger Q
<<Stateful>> Merger Reduce
Merger Storage

Despliegue

En definitiva el sistema se compondrá de cinco nodos;

- Cliente
  - Enviará mediante la ClientLibrary los datasets al servidor.
- Servidor
  - Mapeara los datasets recibidos por parte del cliente y los enviará mediante RabbitMQ a los workers correspondientes
- Middleware
  - La instancia de rabbitmq
- Worker
  - Contiene una acción a realizar (FILTER, OVERVIEW (AI), JOIN, GROUP o TOP), procesa el request y la envía a la siguiente etapa
- Health
  - Se encarga de que estén todos los nodos que pueden fallar (servidores y workers) estén funcionales.


Client
movies.csv
ClientLibrary
Processing UNITs
TopK Node
TopK
Partial Results
Overview Node
Overview
AI Model
Join Node
Joiner
Partial Results
Filter Node
Filter
Reduce Node
Reducer
Partial Results
Map Node
Map
Merger Node
Merger
Partial Results
Proxy
Handler_1
...
Handler_N
Server
ClientHandler_1
...
ClientHandler_N
Middleware
RabbitMQ
Health
Leader

Tolerancia a fallos

Dentro de los potenciales fallos del sistema, encontramos los siguientes

Supuestos

Para la determinación de los fallos a tolerar se tomaron los siguientes supuestos

- El nodo proxy al realizar poco esfuerzo (redirige el flujo de datos entre N clientes a M servidores) se considera que tiene alta disponibilidad y no se caerá en ningún momento.
- Consideramos que la instancia de RabbitMQ tampoco se caerá en ningún momento.

Caída de un nodo

Un nodo puede caerse por diversas razones, ya sea por sobrecarga de trabajo, anomalías externas, etc. Por lo cual, para solucionar este problema optamos por crear un nuevo tipo de nodo llamado Health, el cual puede estar o no replicado (a la hora de definir la configuración considerar que este nodo puede caerse) y se define mediante un consenso que nodo será el líder entre las réplicas (o el mismo). El líder realiza pings constantes a todos los nodos que consideramos que se pueden caer (servidores o workers), si un nodo no llega a responder al segundo ping, se considera caído y se lo levanta nuevamente. En cada ping, el líder comunica a sus réplicas el estado del cluster de nodos. Este estado es la cantidad de veces que un nodo no respondió al ping.
De esta manera detectamos de forma rápida cuando un nodo dejo de funcionar y lo levantamos nuevamente para que pueda continuar su trabajo sin necesidad de que otro nodo funcional requiera continuarlo.

Caída de un servidor mientras el cliente envía datos

En este caso, el proxy al detectar la caída de un servidor al intentar redirigir los datos del cliente, se intentará conectar con otro servidor disponible, haciéndolo transparente al cliente. En caso de que el proxy no pueda conectar con ningún otro servidor, se le avisará al cliente que falló la conexión y que debe intentar nuevamente más tarde.

Caída de un cliente sin retorno

Cuando se cae un cliente y este no vuelve a conectarse, luego de N minutos se limpiarán los datos asociados a tal cliente ya que esos datos no fueron actualizados dentro de esos N minutos.

Paquetes duplicados y desordenados

Para este problema, implementamos un conteo de paquetes dentro del EOF.
El proceso inicia con la ClientLibrary enumerando los batches que le envía al servidor, al enviar un EOF lo que se hace es incluir la cantidad de batches de ese tipo de archivo.

El servidor al parsearlos en tareas puede crear fragmentos de los mismos generando un algoritmo de fragmentación similar al de IPv4.
Un worker stateless al recibir una tarea normal la ejecuta y envía el resultado a la próxima etapa, en caso de ser un EOF el comportamiento es el mismo.
Ahora, cuando el worker es stateful, y recibe una tarea, la ejecuta y persiste el resultado en conjunto con los resultados previamente procesados. En caso de recibir un EOF, si el mismo es del tipo OmegaEOF, el nodo genera un RingEOF que circulará entre los nodos del mismo tipo hasta que se llegue a los N paquetes prometidos.
Para no incluir todos los identificadores de todas las tareas dentro del RingEOF, se optimizó mediante el uso de bloques inspirados en la opción de TCP SACK, por lo cual, en cada nodo por el que pasa el RingEOF debe mergear estos bloques con las tareas que el mismo proceso, **descartando duplicados en el proceso**.

Cabe aclarar que cada vez que pasemos por un nodo stateful, debemos tener en cuenta que el conteo de paquetes se reinicia ya que volvemos a tener un único punto de sincronización. Este nueva cuenta de paquetes se verá reflejada en el **TaskCount** del OmegaEOF que enviemos a la siguiente etapa.

Todo esto es posible gracias a que los paquetes se identifican con un Task Identifier (TID) y dada la estructura del sistema y el método de redirección de mensajes los mismos siempre serán EXACTAMENTE iguales e irán todos al mismo destino. Esto se traduce en que para cada nodo de una etapa en particular, el contenido del mismo será SIEMPRE igual e irá dirigido al mismo destino. La forma en la que la redireccionamos en forma única es gracias a un método de hashing sobre los datos principales y alternativos, el ID del cliente y/o el ID de la Stage según el caso. Esto garantiza que los mensajes irán siempre al mismo destino y la carga se repartirá de forma uniforme entre ellos.

Para poder cumplir con este approach, tenemos que vivir con una gran desventaja la cual es el **envío de paquetes vacíos**, de esta manera se genera una saturación innecesaria de la red. Otra forma de poder usar este approach sin necesidad de enviar paquetes vacíos era **agrandar la cantidad de tipos de worker que son stateful (según cuántos de los mismos destruyen paquetes, caso idóneo: el filter)** la cual consideramos que era aún peor dado que habría que manejar la persistencia de cuantos paquetes fuimos descartando y modificar el contador del total del OmegaEOF en base a eso, y dada la cantidad de tareas que estos nodos procesan decaerá la performance del sistema.

Por último dado que podrían llegar paquetes repetidos al cliente este último debería ser un último filtro de paquetes duplicados el mismo podrá identificar los TID únicos que tiene cada paquete y filtrar aquellos que ya haya recibido.

Para más detalles del RingEOF repasar el diagrama de estados

Persistencia de datos

La persistencia de datos es un proceso esencial para garantizar que los workers puedan recuperar los resultados parciales independientemente del momento en el que se caigan.

Los datos se almacenan en disco en una ubicación cercana al worker de manera que cada worker tiene su propio almacenamiento y ningún nodo necesitará nunca acceder a los datos de otro nodo.

El sistema de persistencia propone una estructura de carpetas del siguiente estilo (excepto en el Joiner donde tendremos un nivel extra para diferenciar la tabla grande y la tabla chica) y para cada carpeta 3 archivos. Uno para guardar los datos que se están procesando en la etapa, otro para guardar metadata útil para evitar algunas duplicaciones de paquetes y sobretodo para el conteo de paquetes y la eliminación de datos y por último un log que lleve el registro de todos los paquetes o tareas que el nodo haya visto en esa etapa:

- Etapa1
    - client_id_1
        - data.json
        - metadata.json
        - logs.log
    - …
    - client_id_2
        - data.json
        - metadata.json
        - logs.log
- …
- Etapa2
    - client_id_1
        - data.json
        - metadata.json
        - logs.log
    - …
    - client_id_2
        - data.json
        - metadata.json
        - logs.log

El sistema de guardado de archivos está pensado de modo tal que no existan puntos en los que la data guardada sea inconsistente y el sistema no se pueda recuperar ya que al momento de procesar guardamos la información en una versión temporal del archivo `temp_{archivo}.json` y más adelante, en un momento en particular ejecutemos una acción atómica que denominamos `commit` pero que en el fondo es un rename atómico de UNIX al nombre del archivo que consideramos definitivo y consistente.

Los nodos tras una caída, se reiniciarán por el nodo Heather y procederán a leer los datos guardados en disco antes de procesar cualquier paquete. Cabe aclarar que solo se tomarán en cuenta los archivos cuyo nombre sea el definitivo y se descartarán archivos temporales que nunca llegaron a commitear.

Para ser más precisos, los datos se guardan a medida que se procesan en el archivo temporal, luego se envían las tareas a la siguiente etapa, luego se ejecuta el commit, damos el ACK a rabbit y por último borramos toda la información que ya no es necesaria. Este flujo garantiza que si el sistema se cae en algún punto intermedio, el sistema en su totalidad siempre se podrá recuperar.

Veamos caso por caso:

- Si se cae durante el procesamiento de la data, volveremos a procesarla al levantarnos ya que no habremos hecho el commit de los datos que recibimos previamente ni habremos dado el correspondiente ACK a rabbit, por lo que volvemos a un estado anterior al procesamiento del paquete y el paquete eventualmente volverá a llegar.
- Si se cae en la escritura del archivo temporal, los resultados se verán invalidados por el proceso de iniciación del sistema, ya que estos son descartados y la tarea volverá a llegar.
- Si se cae enviando las tareas a la siguiente etapa (o luego de concluir todos los envíos) rabbit ignorará paquetes incompletos y aquellos que efectivamente hayan sido enviados se procesarán con normalidad aunque luego el nodo recuperado re envíe estos paquetes. En este caso puntual, cada paquete reenviado saldrá exactamente con el mismo TID y contenido por nuestro sistema de paquetes duplicados y la siguiente etapa será encargada de ignorarlo.
- El sistema NO puede caerse durante el commit, ya que la operación es atómica.
- Si se cae luego del commit y antes del ACK a Rabbit, el nodo cuando se recupere podrá detectar que ese paquete fue efectivamente leído y procesado en el pasado, por lo que lo descartará tan pronto como llegue.
- Si se cae luego del ACK pero antes de borrar los datos, tendremos temporalmente una serie de datos innecesarios en el sistema por lo que tras iniciar el nodo siempre dejaremos corriendo un subproceso que cada N segundos se encarga de borrar los archivos con más de M segundos de antigüedad. Esto no solo asegurará que se limpiará para casos normales, sino que también se limpiará para casos en los que el cliente no envíe datos tras M segundos, por lo que consideraremos una desconexión.

Dado que los logs crecen en forma lineal (un log por paquete) y los archivos de datos y metadatos de forma esporádica, el modo de guardado en el caso de los archivos de datos y de metadatos será por reemplazo completo siempre para garantizar un estado consistente y en el caso de los logs, agregamos una optimización en la que se escribe de forma append.

Esto podría traer problemas ya que existirían 2 puntos de sincronización, y por ende el commit dejaría de ser atómico al 100%. Es por ello que le agregamos un timestamp a cada

log el cual previamente fue agregado en el archivo temporal de datos (no de metadatos) (en el caso del joiner es un caso particular ya que los archivos de datos y metadatos están unificados) luego al hacer el commit este timestamp será el último punto hasta el cual consideremos logs, por lo que si nos levantamos posteriormente debemos leer log a log y guardarlo en un archivo temporal hasta que nos encontremos con un log invalido (no cumple el formato (se le indica un largo de línea al principio)), lleguemos al fin del archivo o el timestamp sea mayor al timestamp del archivo de datos. De esta manera realizamos un commit y limpiamos los logs correspondientes a paquetes que no se haya llegado a guardar su estado correctamente, es decir no hayan commiteado.