Trabajo Práctico:

Movies Analysis

Nombre	Padrón	Mail
Boccazzi, Gianni Antonio	109304	gboccazzi@fi.uba.ar
Carosi Warburg, Juan Pablo	109386	jcarosi@fi.uba.ar
Vroonland, Mateo Daniel	109635	mvroonland@fi.uba.ar

Sistemas Distribuidos
1° Cuatrimestre 2025

Índice

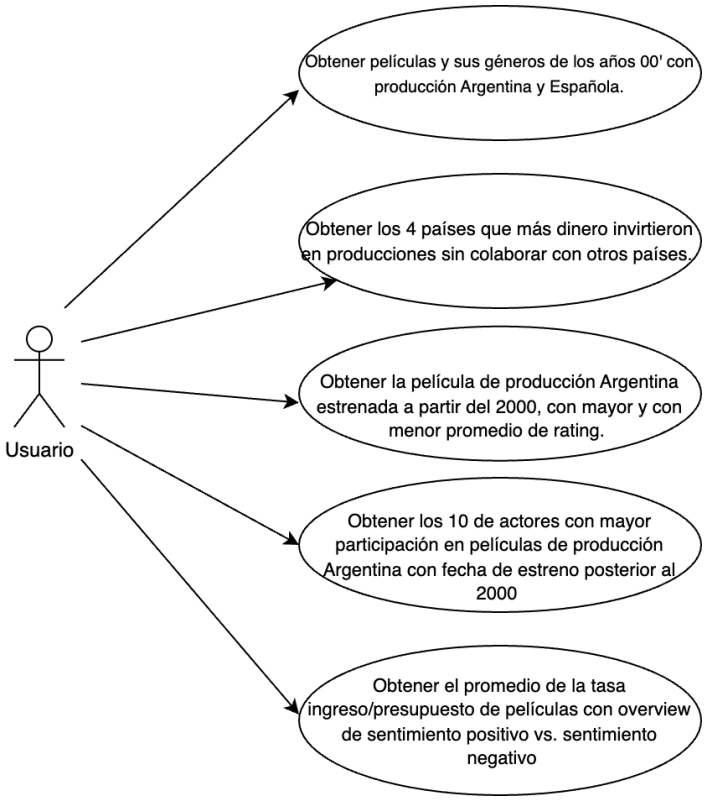
Alcance

El proyecto *Movie Analysis* es un sistema distribuido diseñado para analizar información sobre películas y los ratings de los espectadores en distintas plataformas. Cada película incluye datos como género, fecha de estreno, países involucrados en la producción, idioma, presupuesto e ingresos. Los ratings de los espectadores se expresan en una escala del 1 al 5.

Casos de uso

El sistema deberá permitir distintas queries:

1. Películas y sus géneros de los años 00' con producción Argentina y Española.
2. Top 5 de países que más dinero han invertido en producciones sin colaborar con otros países.
3. Película de producción Argentina estrenada a partir del 2000, con mayor y con menor promedio de rating.
4. Top 10 de actores con mayor participación en películas de producción Argentina con fecha de estreno posterior al 2000
5. Average de la tasa ingreso/presupuesto de películas con overview de sentimiento positivo vs. sentimiento negativo, para películas de habla inglesa con producción americana, estrenadas a partir del año 2000

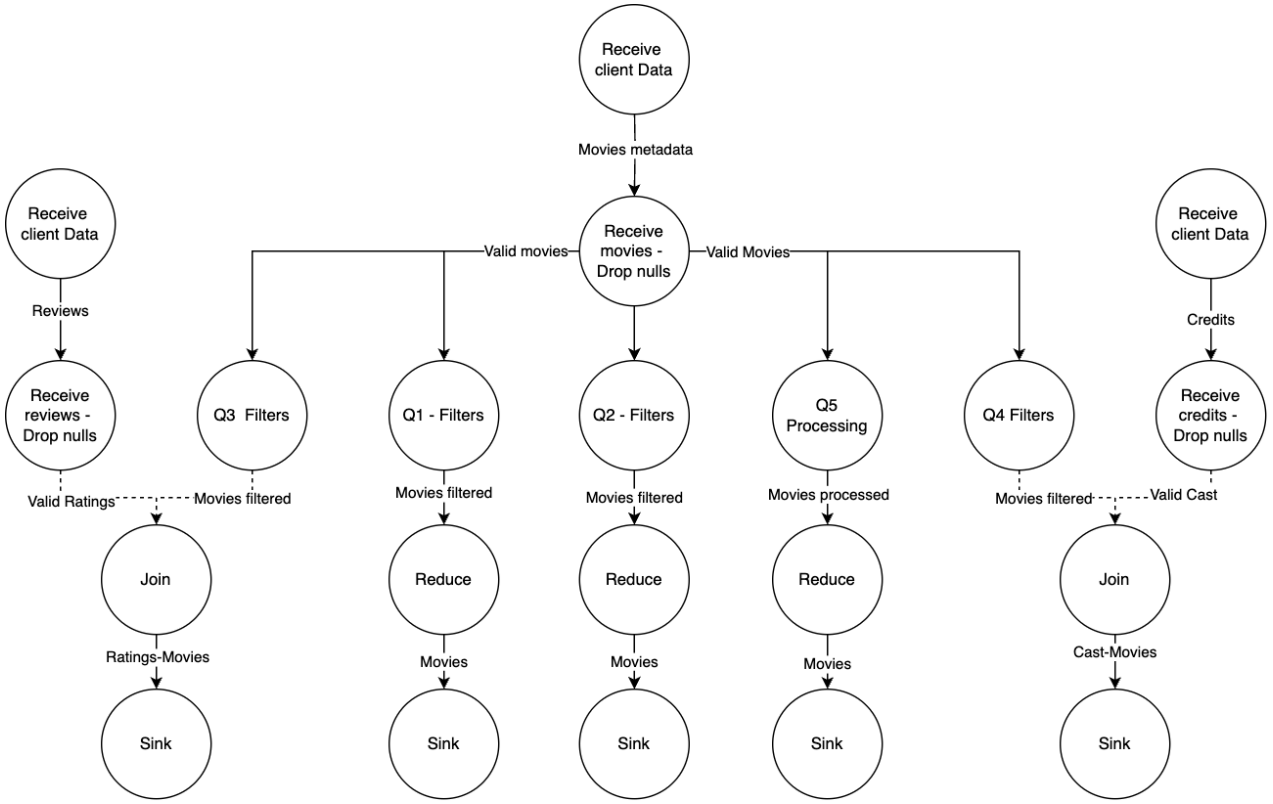


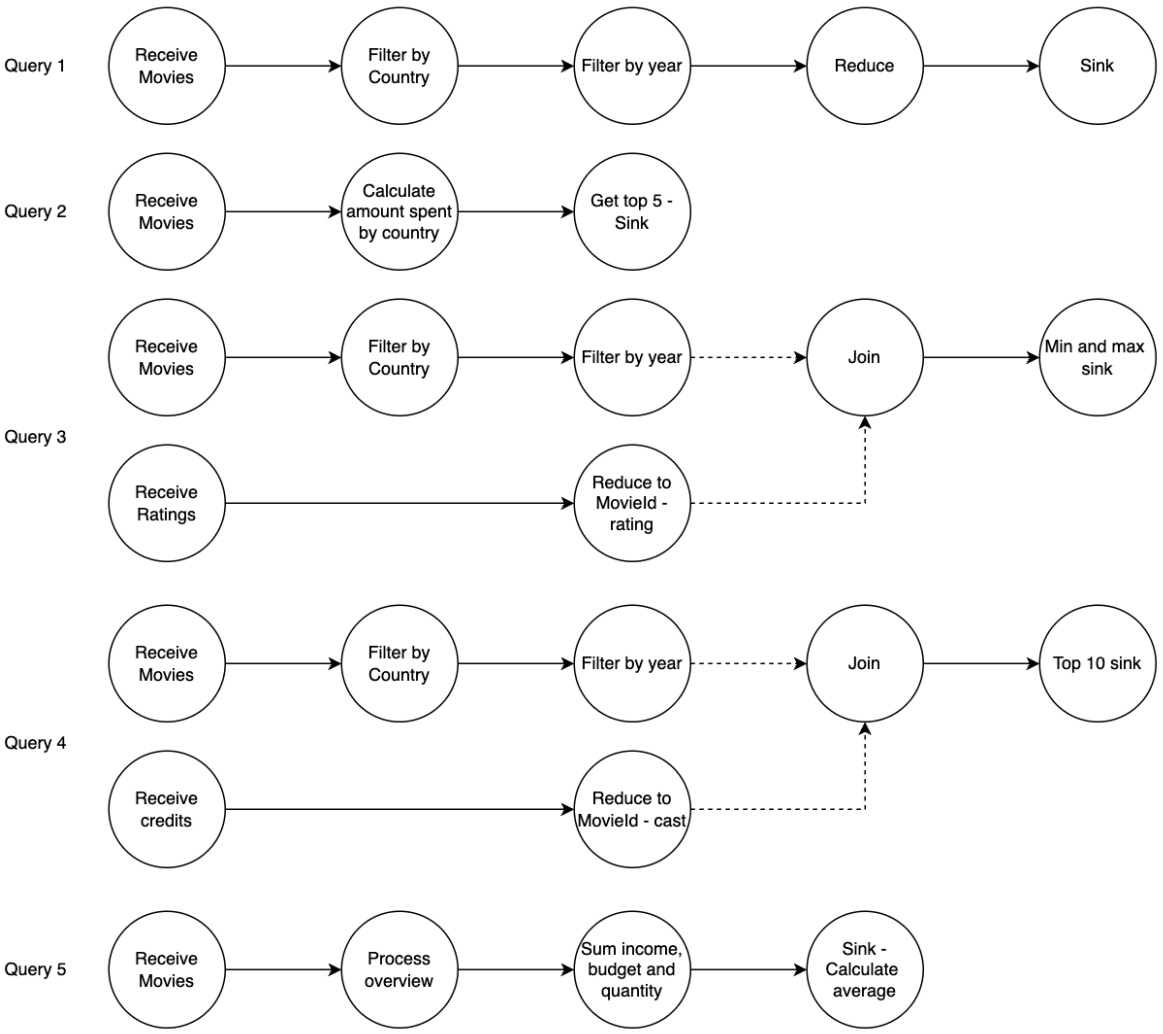
Arquitectura

Vista Lógica

DAG

Los siguientes DAGs (Directed Acyclic Graphs) representan el flujo de procesamiento de datos a partir de tres fuentes principales: **reviews**, **movies** y **credits**. A partir de estos datasets, se realizan transformaciones, filtrados y agregaciones para resolver cinco consultas (Q1 a Q5) y generar salidas finales ("sinks"), donde luego el cliente va a recibir estos resultados finales.




Query 1
Receive Movies
Filter by Country
Filter by year
Reduce
Sink
Query 2
Receive Movies
Calculate amount spent by country
Get top 5 - Sink
Query 3
Receive Movies
Filter by Country
Filter by year
Join
Min and max sink
Receive Ratings
Reduce to MovieId - rating
Query 4
Receive Movies
Filter by Country
Filter by year
Join
Top 10 sink
Receive credits
Reduce to MovieId - cast
Query 5
Receive Movies
Process overview
Sum income, budget and quantity
Sink - Calculate average

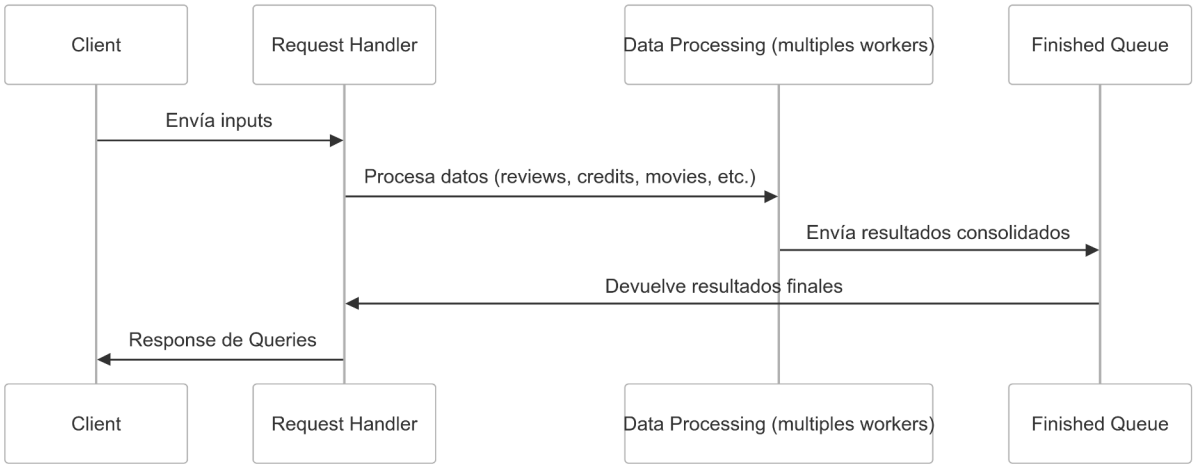
Vista de Procesos

La vista de procesos muestra cómo interactúan los diferentes componentes de nuestro sistema enviando mensajes y mostrando cómo se comporta cada componente en base a los mismos. Como resultado se puede ver el flujo total de cómo se va a comportar nuestro sistema a grandes rasgos. Además, debajo de este diagrama se puede encontrar un link a un segundo diagrama de secuencia el cual posee un grado de detalle mayor.

Diagrama de Secuencia

El diagrama de secuencia representa de forma vertical los procesos que tiene nuestro sistema distribuido y cómo se comunican entre ellos a través del tiempo.

Como nuestra aplicación es muy compleja hicimos 2 diagramas, el primero simplificado que se ve a continuación y uno detallado en el link debajo que lleva a un svg donde se puede hacer zoom.



Link a SVG de diagrama de secuencia detallado

Diagrama de Actividades

El diagrama de actividades permite ilustrar el flujo entre los distintos nodos del sistema, y la dependencia entre ellos:

Vista de desarrollo

Esta vista muestra la perspectiva del programador a la hora de desarrollar el sistema, donde ya podemos visualizar los diferentes módulos que se van a tener

que programar y cuáles se van a relacionar entre sí.

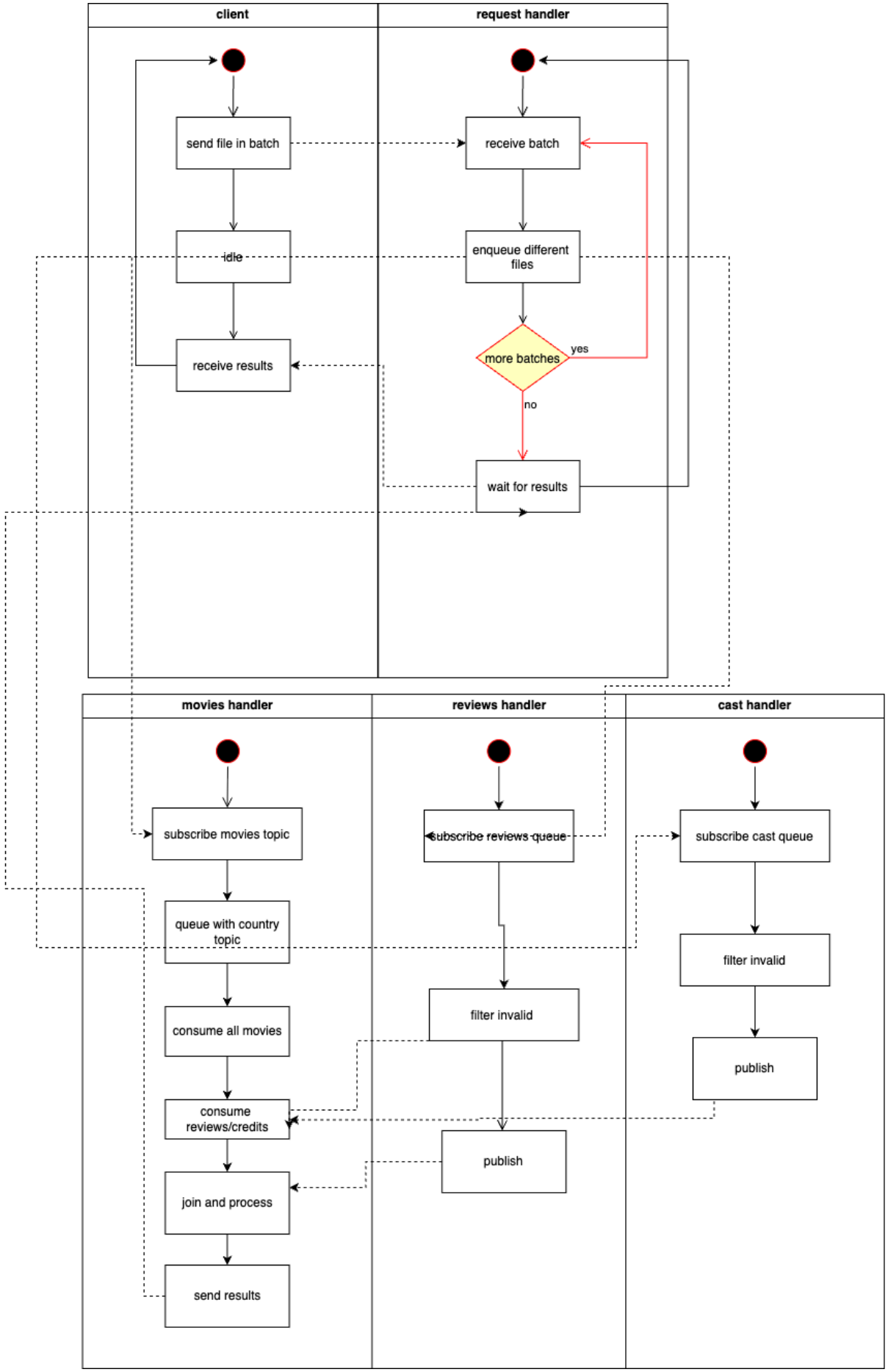
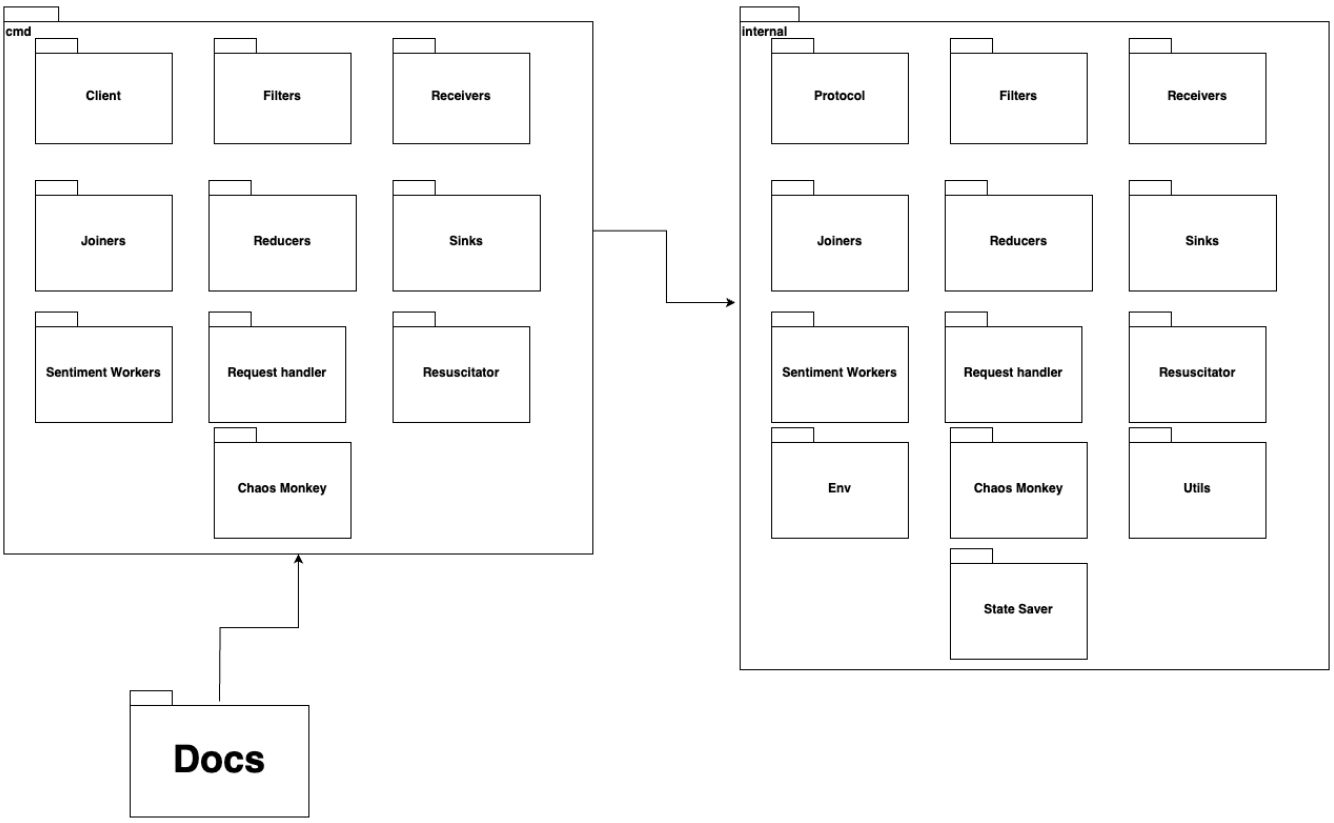


Diagrama de paquetes

Los diagramas de paquetes muestran cómo se agrupan los diferentes elementos (interfaces, módulos, etc) de un sistema y cuales se relacionan entre sí.



Vista física

Esta vista intenta representar cómo se va a desplegar el sistema, teniendo en cuenta la topología de la capa física y como los componentes de esta capa van a realizar su comunicación.

Diagrama de Robustez general

El siguiente diagrama representa la arquitectura robusta del sistema con un pantallazo general. A través del uso de colas, operaciones desacopladas y tareas especializadas, el sistema asegura un manejo eficiente de archivos de entrada, con módulos diseñados para procesar, combinar y derivar información de manera resiliente y escalable.

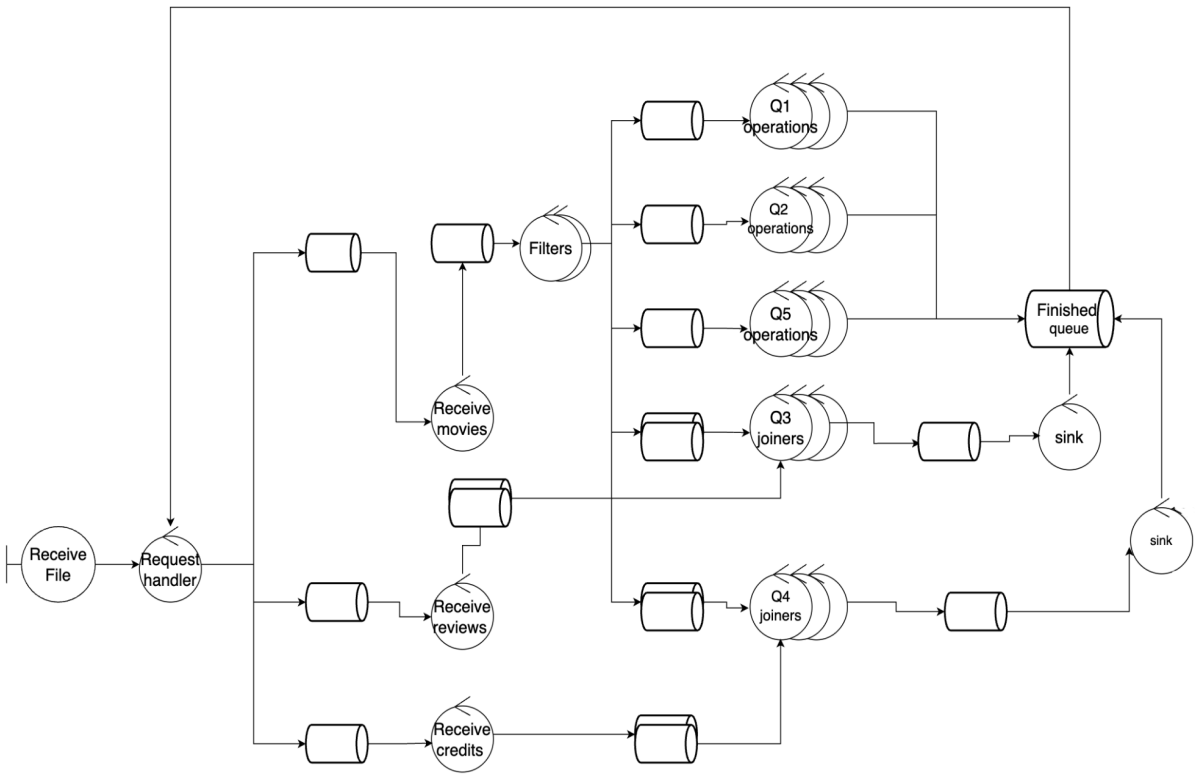


Diagrama de robustez por query

Como el diagrama de robustez general es muy complejo y posee demasiadas entidades, decidimos realizar este diagrama auxiliar donde se hace zoom a las estructuras elegidas para manejar las operaciones necesarias para cada query. Las tareas de cada nodo son simples:

- **Joiner**: se encarga de unir las muestras de las películas con las muestras del cast o reviews según sea necesario para la query.
- **Reducer**: agrupa y resume los datos recibidos aplicando funciones como suma, conteo o promedio.
- **Filter**: filtra los datos según la condición esperada, como por ejemplo filtrar por año de película que es necesario para casi todas las queries.
- **Sinker**: recibe los datos finales, los formatea y los envía a la cola de resultados.

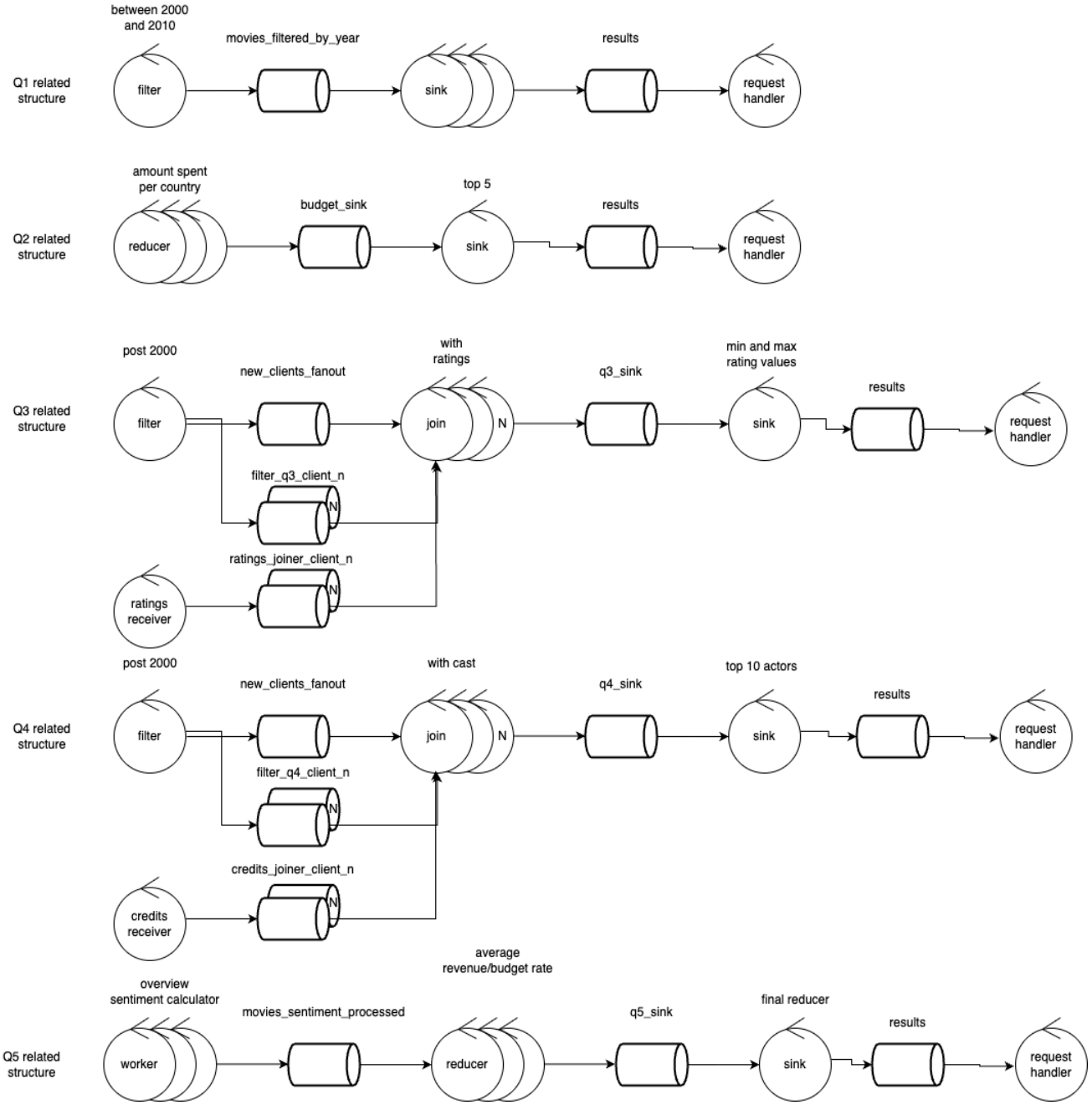
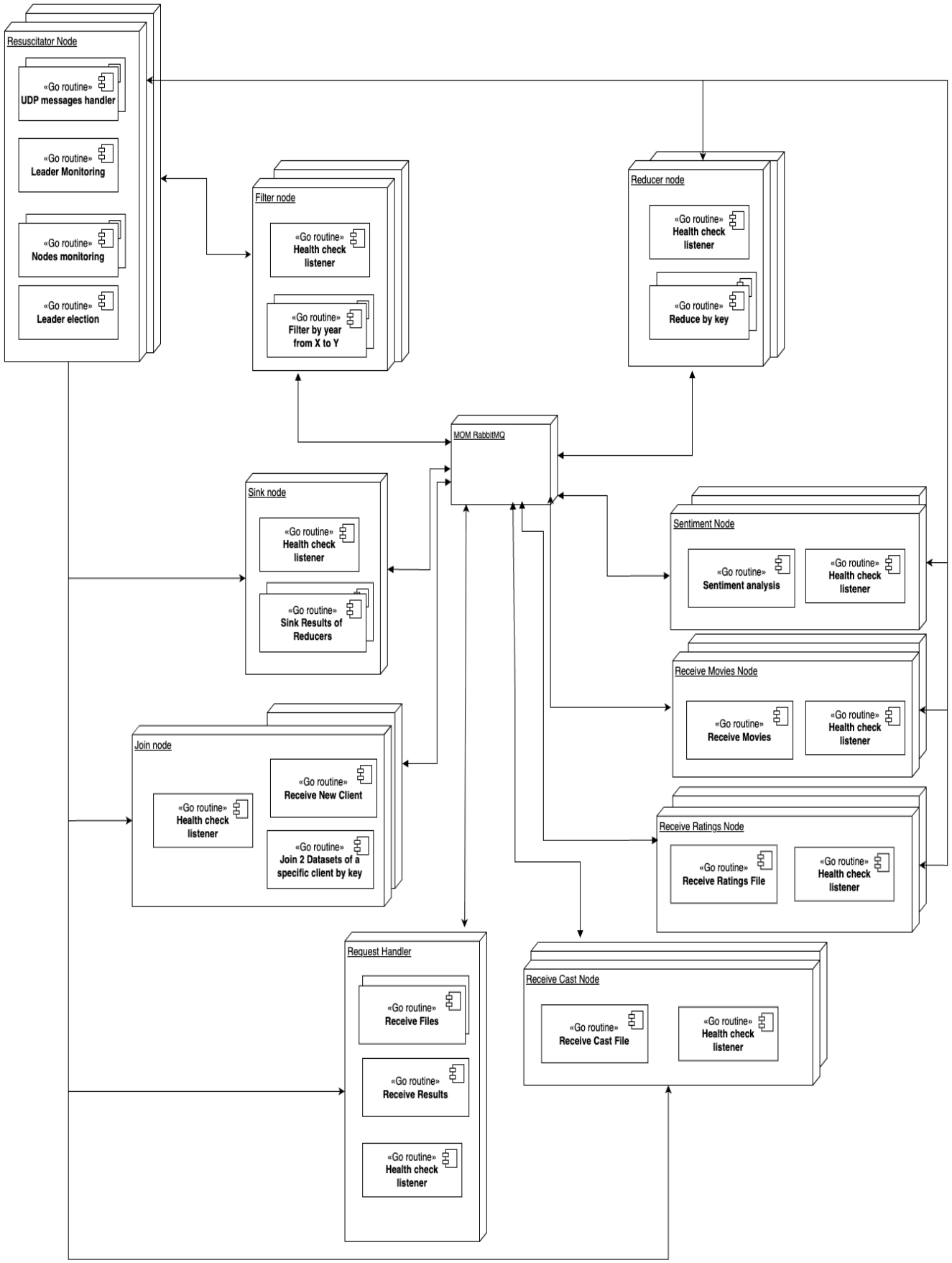


Diagrama de despliegue

En el diagrama se muestra la distribución de las entidades en sus respectivas unidades de cómputo. Además, se puede observar que toda comunicación se realiza a través del middleware rabbitMQ, exceptuando la conexión inicial entre el cliente y el gateway receiver mencionado anteriormente.


Resuscitator Node
«Go routine»
UDP messages handler
«Go routine»
Leader Monitoring
«Go routine»
Nodes monitoring
«Go routine»
Leader election
Filter node
«Go routine»
Health check listener
«Go routine»
Filter by year from X to Y
Reducer node
«Go routine»
Health check listener
«Go routine»
Reduce by key
MOM RabbitMQ
Sink node
«Go routine»
Health check listener
«Go routine»
Sink Results of Reducers
Sentiment Node
«Go routine»
Sentiment analysis
«Go routine»
Health check listener
Receive Movies Node
«Go routine»
Receive Movies
«Go routine»
Health check listener
Join node
«Go routine»
Health check listener
«Go routine»
Receive New Client
«Go routine»
Join 2 Datasets of a specific client by key
Receive Ratings Node
«Go routine»
Receive Ratings File
«Go routine»
Health check listener
Request Handler
«Go routine»
Receive Files
«Go routine»
Receive Results
«Go routine»
Health check listener
Receive Cast Node
«Go routine»
Receive Cast File
«Go routine»
Health check listener

Detalles de implementación

1. **Sanitización de Datos:** Realizamos sanitización de datos en dos lugares diferentes, en primer lugar en el cliente, al levantar de dataset para enviarlo aprovechamos que el mismo fue escrito en python para parsear las columnas que son *dictionary-like* a un formato JSON válido para que el servidor en go pueda parsear el mismo de forma correcta, evitando reinventar la rueda programando un parseador de diccionarios de python para poder leer el dataset. En segundo lugar, apenas ingresan los datos al servidor, descartamos aquellas filas que son inválidas porque contienen datos vacíos en los campos que vamos a utilizar para calcular las queries. Por último algunas queries tienen su filtro específico ya que un valor no vacío no es suficiente, sino que por ejemplo para la query 5 hay que descartar valores inválidos como presupuesto igual a 0, ya que arruinaría el cálculo del promedio, o directamente no se podría realizar el cálculo (por divisiones entre 0).
2. **Comunicación de datos:** Se desarrolló un middleware propio el cual posee una interfaz para crear colas consumidoras y productoras, poder publicar y consumir de las mismas y que también manejen el EOF de las comunicaciones, separando de esa manera la arquitectura y topología de nuestro despliegue de la lógica de negocio. Además aquí se encapsula toda la lógica de rabbit, por lo que si el día de mañana queremos utilizar otro MOM, solo bastaría con cambiar este pequeño paquete manteniendo las firmas y nuestra lógica de negocio no cambiaría en absoluto.
3. **Uso de Rabbit:** Por como se puede ver en los diagramas, dependemos de rabbit para casi toda la comunicación de nuestro sistema, a la hora de correr el proyecto rabbit consume muchos recursos (especialmente RAM) por la cantidad de mensajes "*in flight*", por lo que definimos un prefetch de 5000 mensajes para las comunicaciones, de esta manera reduciendo la cantidad de ram que utiliza este módulo.
4. **Shardeo de Mensajes y EOF:** El sistema que se decidió utilizar para los EOF es una implementación de un Poison Pill, donde cada productor al terminar envía un finish por la misma cola de datos y el consumidor al recibirla ya sabe que tiene que dejar de consumir. Esto es simple en las colas 1 a 1 (1 productor a 1 consumidor) pero se dificulta a la hora de tener muchos productores y muchos consumidores. Para afrontar este problema se decidió shardear todos los mensajes que se envían (cada mensaje tiene su forma de ser identificado para saber a qué shard va a ir a parar, por ejemplo las rows de películas se identifican por su id de película pero las que tienen ya actores se identifican por el id del actor). De esta manera cada mensaje tiene un nodo al cual va a ir a parar deterministicamente, por lo que cada productor al terminar puede enviarle un mensaje a cada consumidor y sabe que le va a llegar exactamente una poison pill a cada uno de sus consumidores. Por el lado de los consumidores saben cuántos productores hay produciendo entonces simplemente cuentan la cantidad de poison pills que recibieron y cuando ya llegaron todas saben que

terminaron. Para que tanto productores como consumidores sepan cuantos nodos hay del otro lado, se tomó la decisión que la cantidad de nodos se pase via env y sea fija durante la ejecución del programa, donde todos los nodos saben la cantidad de nodos de otras instancias que hay.

Tolerancia a fallas

Para que el sistema pueda ser tolerante a fallas, se agregaron las siguientes features al mismo.

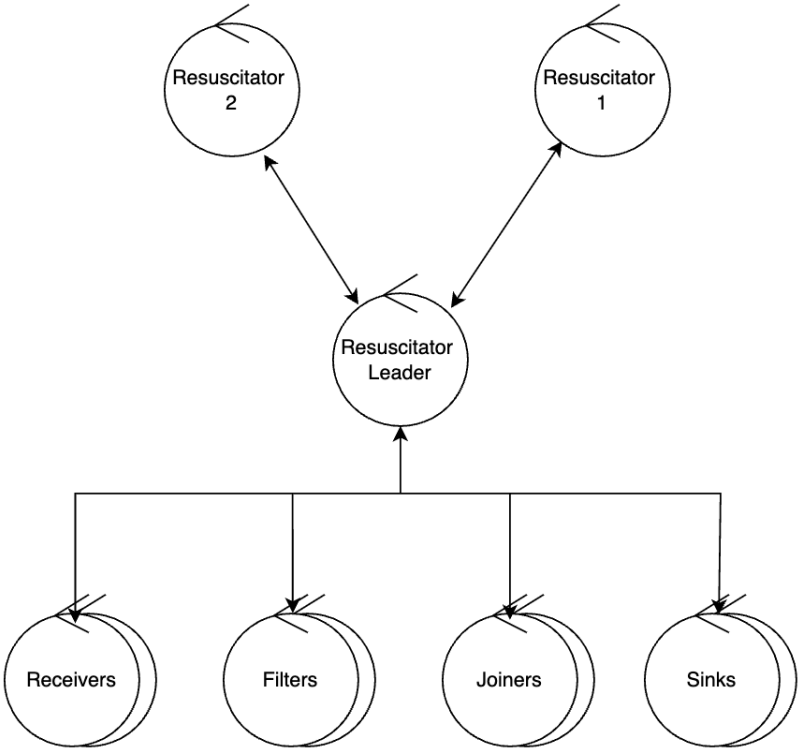
Resucitador

Se agrega una nueva entidad al sistema, denominada **resucitador**, cuya función principal es monitorear el estado de cada uno de los nodos del sistema. Para ello, cada nodo, al inicializarse, abre un socket UDP y permanece en escuchando en paralelo. A través del socket, el resucitador envía, cada cierto tiempo, health checks y espera una respuesta por parte de cada nodo. Si tras **tres intentos consecutivos** no obtiene respuesta, considera que el nodo está caído y procede a reiniciarlo. Luego de esto, retoma el monitoreo habitual.

Esta nueva entidad también cuenta con **réplicas**. Al iniciar el sistema, cada réplica se comunica con las demás a través de sockets UDP y sabiendo de antemano cuántas están activas. La tarea de monitorear los nodos recaerá exclusivamente en una de las réplicas, la cual será designada como **líder**. Para determinar cuál de ellas asume ese rol, se ejecuta el algoritmo **Bully** de elección de líder durante el arranque del sistema.

Una vez elegido, el resucitador líder se encargará de monitorear tanto a los nodos como al resto de las réplicas. Si alguna réplica no responde tras tres intentos, el líder asumirá que está caída e intentará reactivarla.

Por su parte, las réplicas deben responder al líder cuando éste consulte su estado, y además registrarán la última vez que recibieron un mensaje del mismo. Si transcurre un tiempo determinado sin recibir comunicación del líder, las réplicas iniciarán nuevamente el algoritmo de elección, con el fin de designar un nuevo líder que retome las tareas de monitoreo.



Orden de Mensajes

Para manejar el posible desordenamiento de mensajes debido a los requeues causados por caídas o mensajes duplicados se implementó un sistema de números de secuencia por cada mensaje, por lo que al recibir un mensaje duplicado, el consumidor puede saber instantáneamente si ese mensaje ya lo había consumido previamente, y así mismo para mensajes desordenados, el consumidor si llega a recibir un mensaje mayor a su número de secuencia simplemente realiza un Nack para reencolar el mensaje hasta que llegue en su orden debido. Esta implementación facilitó muchísimo la implementación de EOF , ya que al asegurarnos el orden de los mensajes independientemente del MOM que utilicemos, podemos saber con seguridad que siempre la poison pill consumida no va a estar antes de un mensaje de datos.

Persistencia de estados

Se implementó un mecanismo de persistencia de estados que permite a cada componente del pipeline de procesamiento de datos mantener su estado interno y recuperarse automáticamente ante fallos. En su primera implementación habíamos implementado la persistencia guardando el estado por cada mensaje que recibía el nodo, lo cual fue bastante simple de resolver.

Sin embargo, debido al overhead que genera el guardado a disco por cada mensaje, se decidió cambiar la implementación a un sistema que graba de a batches de mensajes que va agrupando. Este se guarda el estado a bajar a disco en conjunto con una función para ackear/nackear, donde una vez que se llega al límite del batch, se flushea todo a disco y se manda el ack/nack de cada mensaje según sea necesario. Este enfoque trabaja en conjunto con la característica del

handleo de mensajes en orden ya que, si por ejemplo el nodo se cae inmediatamente después de grabar en disco pero antes de realizar los acks pertinentes, los mensajes no van a ser ackeados/nackeados y por lo tanto volverán a la cola. Sin embargo, como los números de secuencia de cada cola fueron grabados apropiadamente, cuando esos mensajes reencolados entren nuevamente al nodo que revivió, estos van a ser ignorados debidamente ya que su número de secuencia será menor al esperado.

El núcleo de esta arquitectura está basado en la estructura StateSaver que utiliza generics de Go para ser reutilizable con cualquier tipo de estado. Este gestiona los ACKs y NACKs de mensajes de forma inteligente, manteniendo listas de operaciones pendientes que solo se ejecutan tras un guardado exitoso del estado como fue mencionado anteriormente.

Cada tipo de componente del sistema define su propia estructura de estado que contiene toda la información necesaria para reanudar el procesamiento desde el último punto conocido. Estos estados específicos incluyen una función de serialización que convierte el estado a formato binario GOB, y una instancia de StateSaver configurada específicamente para ese tipo de componente.

Un ejemplo de que tipo de estructura guardamos es la del joiner, la cual como mencionamos anteriormente se guarda el struct entero en formato gob, la cual tiene el siguiente esquema:

```go
type CreditsJoinerClientState struct {
  MoviesConsumer         utils.ConsumerQueueState
  CreditsConsumer        utils.ConsumerQueueState
  SinkProducer           utils.ProducerQueueState
  MoviesIds              map[int]bool
  ClientId               string
  FinishedFetchingMovies bool
}
```

El estado es serializado con gob y guardado en formato binario, donde se pueden ver algunos estados propios del joiner como *clientId* que indica que cliente está joineando esa estructura particular (se crea una go routine por cada cliente), *moviesId*, que indica que peliculas ya recibió para arrancar a joinear con los ratings y también un booleano *finishedFetchingMovies* que indica si ya se recibieron todos as las movies así al recuperarse de una caída pasan directo a los ratings. Los primeros 3 estados son lo estados que tenían las colas de nuestro middleware en el instante que se guardó, las cuales tienen el siguiente esquema:

```go
type ProducerQueueState struct {
  SequenceNumbers map[string]map[string]int // clientId -> routingKey -> sequenceNumber
}
```

```
type ConsumerQueueState struct {
  FinishedReceived map[string]map[string]bool
  SequenceNumbers  map[string]map[string]int // clientId -> producerId -> sequenceNumber
}
```

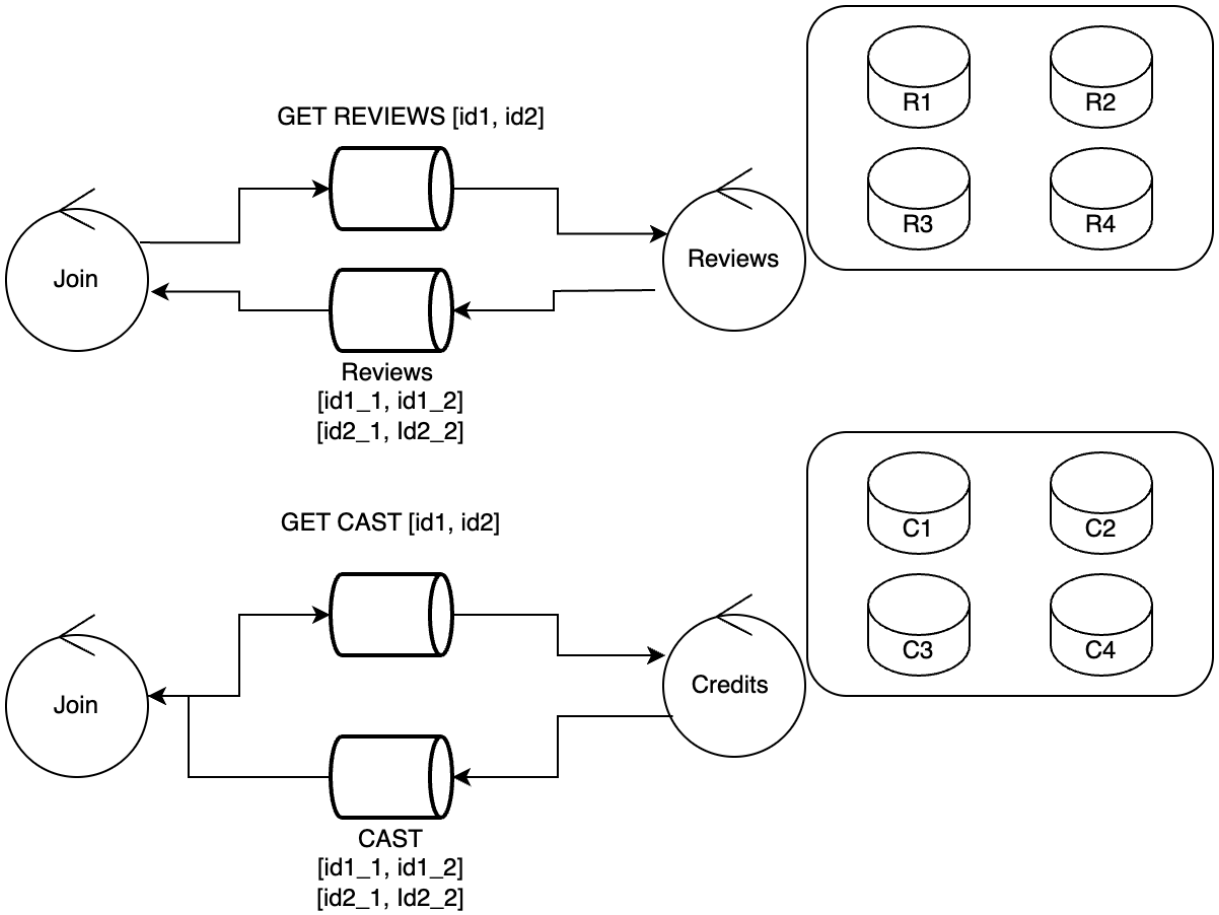
Estos estados contienen qué número de secuencia venía enviando o cual venía recibiendo, acompañado de la cantidad de mensajes de finish (poison pill) que se recibieron de diferentes productores.

Persistencia - Deprecado

Durante la implementación del diseño, se identificó que era necesario procesar primero la información de películas para luego obtener los Ids de los *reviews y credits* requeridos en ciertos joins. Esta situación generaba la necesidad de retener temporalmente algunos datos y solicitarlos en etapas posteriores del flujo.

Para resolver este problema, se propuso la incorporación de nodos tipo *DB*, los cuales almacenan en disco los datos recibidos (como *casts* y *reviews*), permitiendo que otros nodos accedan a dicha información bajo demanda. Esta solución reduce el uso intensivo de memoria y disminuye el riesgo de pérdida de datos ante posibles fallos en comparación con alternativas como dejar todas las reviews en memoria o tener una cola para cada worker que las quiera consumir. Además, se decidió particionar los datos en disco, permitiendo acceder a los registros en disco en paralelo y leer menos líneas en general por búsqueda (si se tienen N registros en M shards, la complejidad de buscar un registro en especifico seria de leer N / M registros, contra los N si no se realiza este particionado), optimizando el acceso y mejorando la eficiencia en las consultas.

Esto no se terminó llevando a cabo ya que el cast en si no termino siendo overhead suficientemente grande como para que valga la pena dicha implementación y para el caso de las reviews no pudo ser implementado ya que las mismas debían ser interpretadas como un stream infinito de datos, lo cual no sería compatible con bajar todos los registros a disco.



División de tareas

Cliente: envío de datos al sistema	Gianni
RequestHandler: Lectura de datos, envío a tópicos	Juan Pablo
ReceiveMoviesWorker: Filtrado de nulos y envío a tópicos por país	Mateo
Filters: filtro por año	Gianni
Reducers: movie/genre, dinero gastado por país, tasa ingreso/presupuesto	Juan Pablo
Sinks	Mateo
ReceiveRatingsWorker: guardado a disco - sharding. Envío de data solicitada	Gianni
ReceiveCastWorker: guardado a disco - sharding. . Envío de data solicitada	Juan Pablo
Joiners: request de datos a nodos BD y join	Gianni
Sentiments worker	Mateo
Resucitadores	Gianni
Persistencia en Disco	Juan Pablo - Mateo

Anexo

Boceto de manejo de EOF