FACULTAD DE INGENIERIA

Universidad de Buenos Aires

75.74 Sistemas Distribuidos I

TP1 - Movies Analysis

1º Cuatrimestre 2025

Integrantes:

Nombre	**Padrón**	**Mail**
Agustin Nicolas Gonzalez	106086	agngonzalez@fi.uba.ar
Facundo Luzzi	105229	fluzzi@fi.uba.ar

Índice

Introducción

Objetivo del sistema:

El objetivo de este sistema es analizar información de películas y calificaciones utilizando datasets públicos de Kaggle. Se busca ofrecer distintos tipos de análisis y consultas sobre las películas, actores, directores y ratings por parte de los usuarios.
En particular, se busca obtener el resultado de las siguientes cinco consultas:

- 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

Alcance del trabajo:

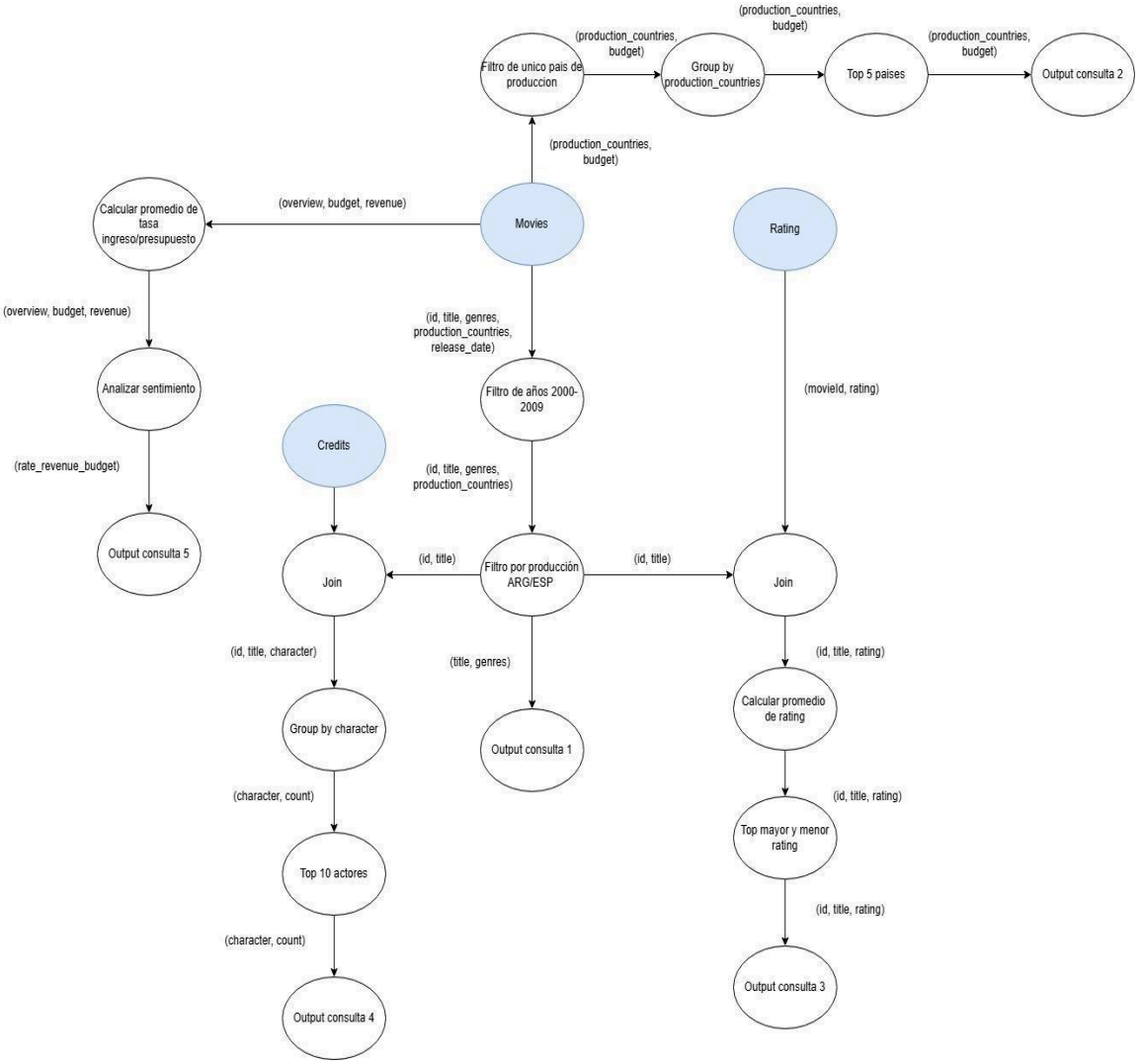
Este trabajo práctico se enfoca en el diseño arquitectónico del sistema distribuido para procesamiento y análisis de datos. No se presentará el código, sino la justificación del diseño a través de las vistas arquitectónicas estudiadas en clase y expresadas a través de distintos diagramas.

Diseño de Arquitectura

Vista Lógica

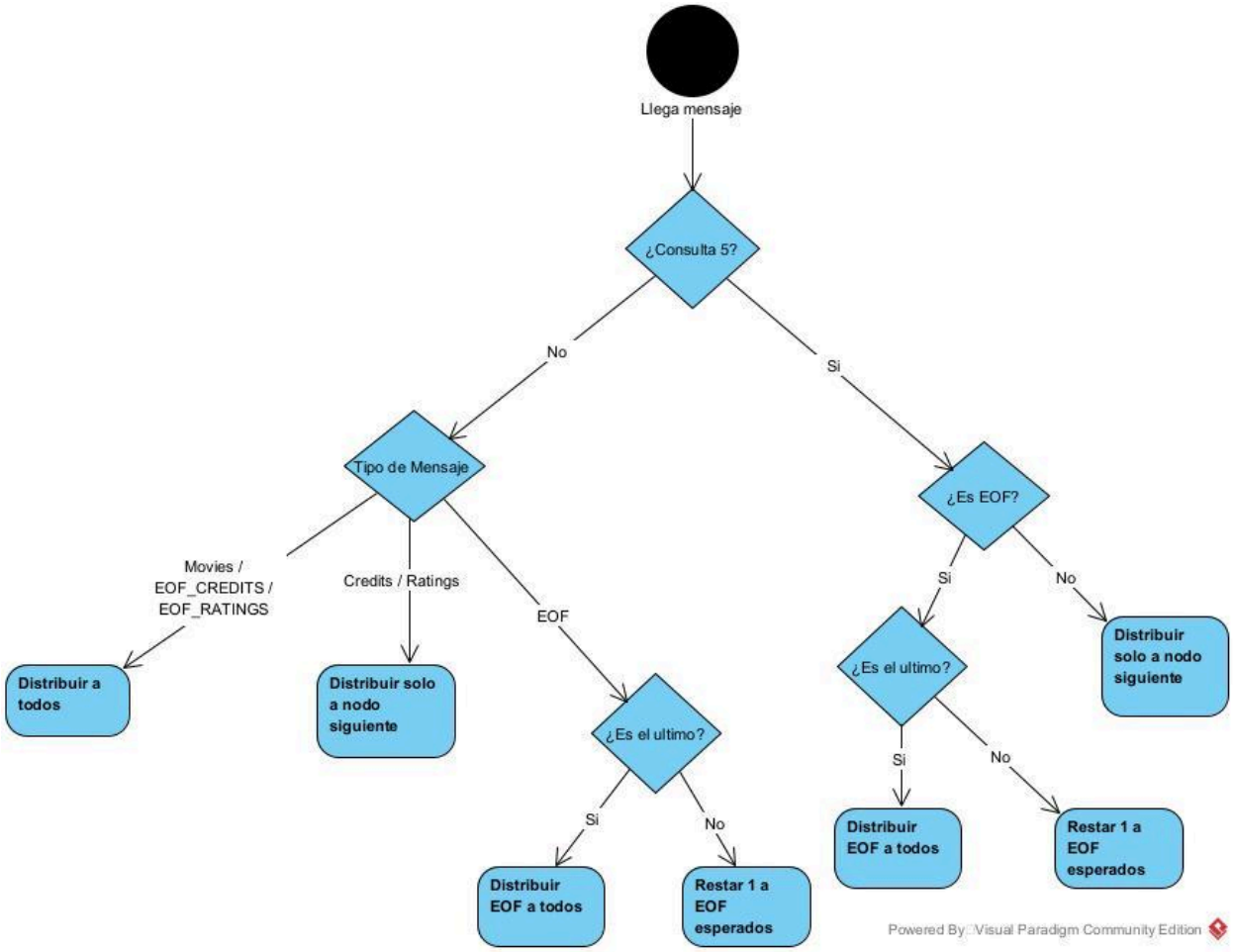
Procesamiento de datos

El sistema está diseñado para recibir información acerca de movies, credits y ratings. Se aplica una serie de operaciones sobre los datos, transformándolos a lo largo del flujo de forma escalonada, donde cada etapa cumple una función determinada para llegar al resultado final.
Cada etapa del procesamiento tiene como objetivo reducir el volumen de datos, refinando la información y estructurándola de acuerdo a los requerimientos de cada consulta.



Decisión de Nodos

Parece interesante mostrar la lógica en la decisión del nodo Broker a la hora de redirigir los mensajes provenientes desde el nodo Input para repartir la carga a los Joiner en los archivos de Ratings y Credits, y desde el nodo Filter para repartir la información a los nodos Pnl para la consulta 5, y broadcastear las movies para todos los nodos Joiner.
La incorporación de este nodo fue vital para el manejo de los EOF en los nodos mencionados, asegurándonos de que no haya pérdida de información ni mensajes duplicados.
De esta forma nos aseguramos que los nodos Joiner siempre tendrán todo el dataset de movies ya procesados por el Filter, y a su vez que la carga del resto de dataset sea distribuida exitosamente.



Agustin Gonzalez 106086 - Facundo Luzzi 105229

Vista Física

Componentes del sistema

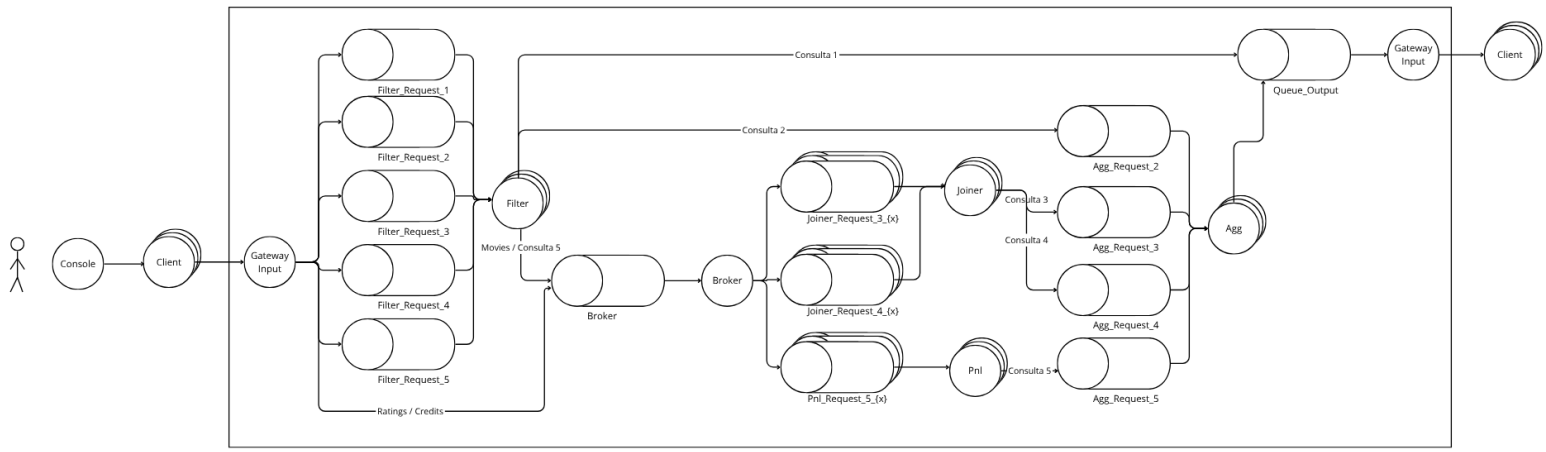
El diagrama de robustez muestra los componentes físicos y su distribución.

Empezando con los Clientes, que como vemos pueden ser escalables, cuyos mensajes seran recibidos por el Gateway Input, el cual se encargará de enviarle a cada nodo lo necesario para ejecutar su consulta. Publicará en las colas Filter_Request de la 1 a la 5 dependiendo de que consulta se tratase, asi como tambien publicara en la cola del Broker los batches pertenecientes a los csv de credits y ratings necesarios para las consultas 3 y 4.

Exceptuando los Gateway y el nodo Broker, todos los otros componentes del sistema (Filter, Joiner, Pnl y Aggregator) son escalables en cantidad para mejorar la distribución y el balanceo de carga.

Una aclaración es que si bien los nodos Aggregator son escalables, el número máximo que puede existir de estos es de 4, ya que son 4 las consultas que atienden los mismos, y no puede lógicamente haber más de 1 nodo Aggregator ocupándose de la misma consulta debido a su propia naturaleza.

Las colas de las que consume cada nodo Filter, Broker y el Gateway son únicas, en cambio, las que consumen cada nodo Joiner y Pnl son escalables a la cantidad de nodos. Es decir, hay tantas colas como nodos, por lo que cada nodo consume solamente de su propia cola. Este escalamiento de colas es posible gracias al manejo de EOF por parte de su predecesor, el nodo Broker.



Distribución de Cargas

Esta mencionada distribución de cargas, se realiza al ejecutarse el script “mi-generador.sh”, el cual recibe por línea de comando la cantidad de nodos de cada tipo que deseamos levantar, este mismo, se encarga de generar en cada uno de los nodos sus variables de entorno para que las consultas se ejecuten de forma distribuida.

Este balanceo de cargas se hace según cantidad de nodos disponibles y necesarios para atender una consulta.

Por ejemplo si creamos un solo nodo Filter, ese único nodo estará encargado de atender las consultas de la 1 a la 5. Si en cambio levantamos dos nodos Filter, el Filter 1 estará encargado de atender 2 consultas y el Filter2 3 consultas. Si levantamos 10 nodos Filter, cada consulta tendrá asignado 2 nodos filter para que la atiendan.

Es decir, que la relación nodo-consulta, puede ser tanto de un nodo atendiendo muchas consultas, como de muchos nodos atendiendo una consulta. Esto exceptuando el caso ya mencionado del Aggregator, que debido a su naturaleza no permite a dos nodos atender la misma consulta.

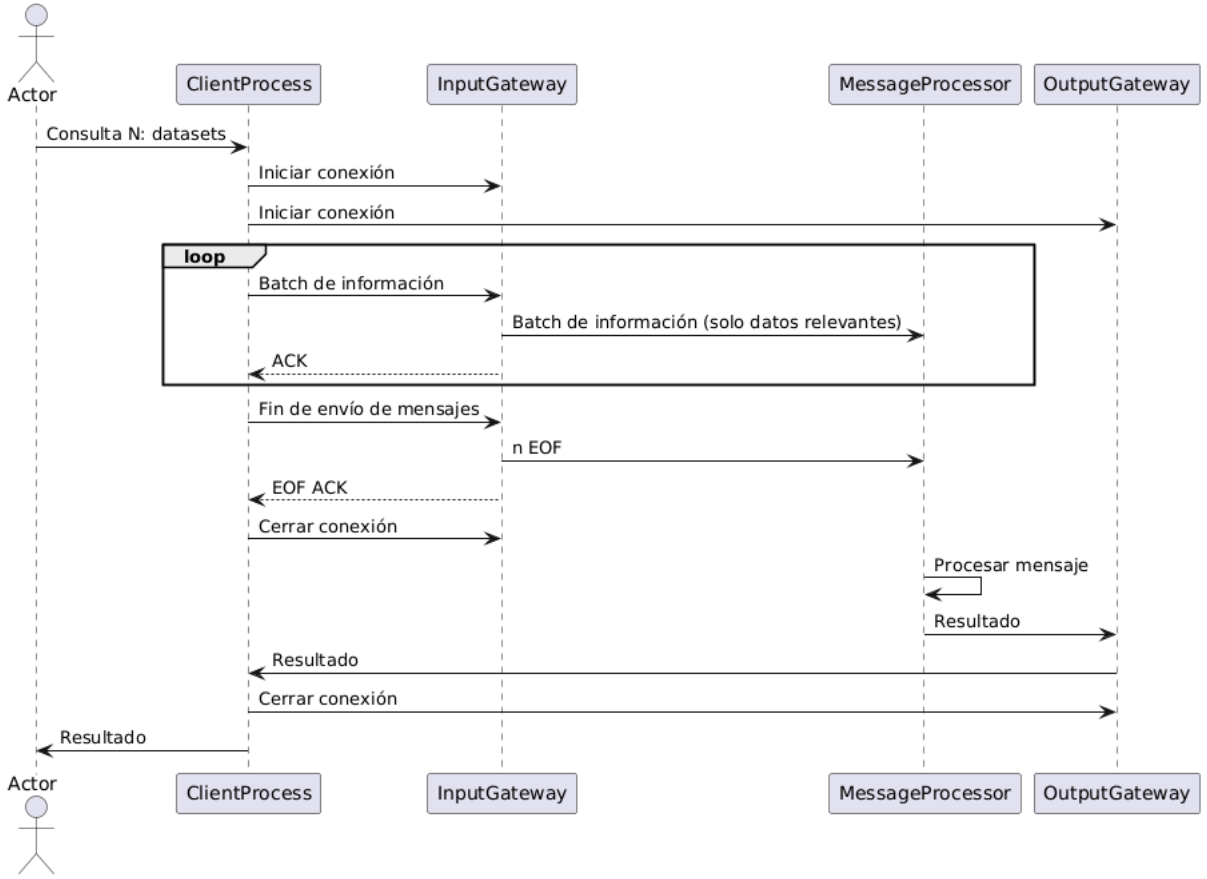
Vista de Procesos

Secuencia de Mensajes

En el diagrama podemos observar a alto nivel la comunicación entre el cliente y el servidor con sus distintos nodos/componentes. El proceso arranca cuando el cliente inicia una conexión TCP con el Gateway, la cual se mantiene activa durante todo el envío de información, y la entrega del resultado.

Podemos separarlo en cuatro puntos importantes:

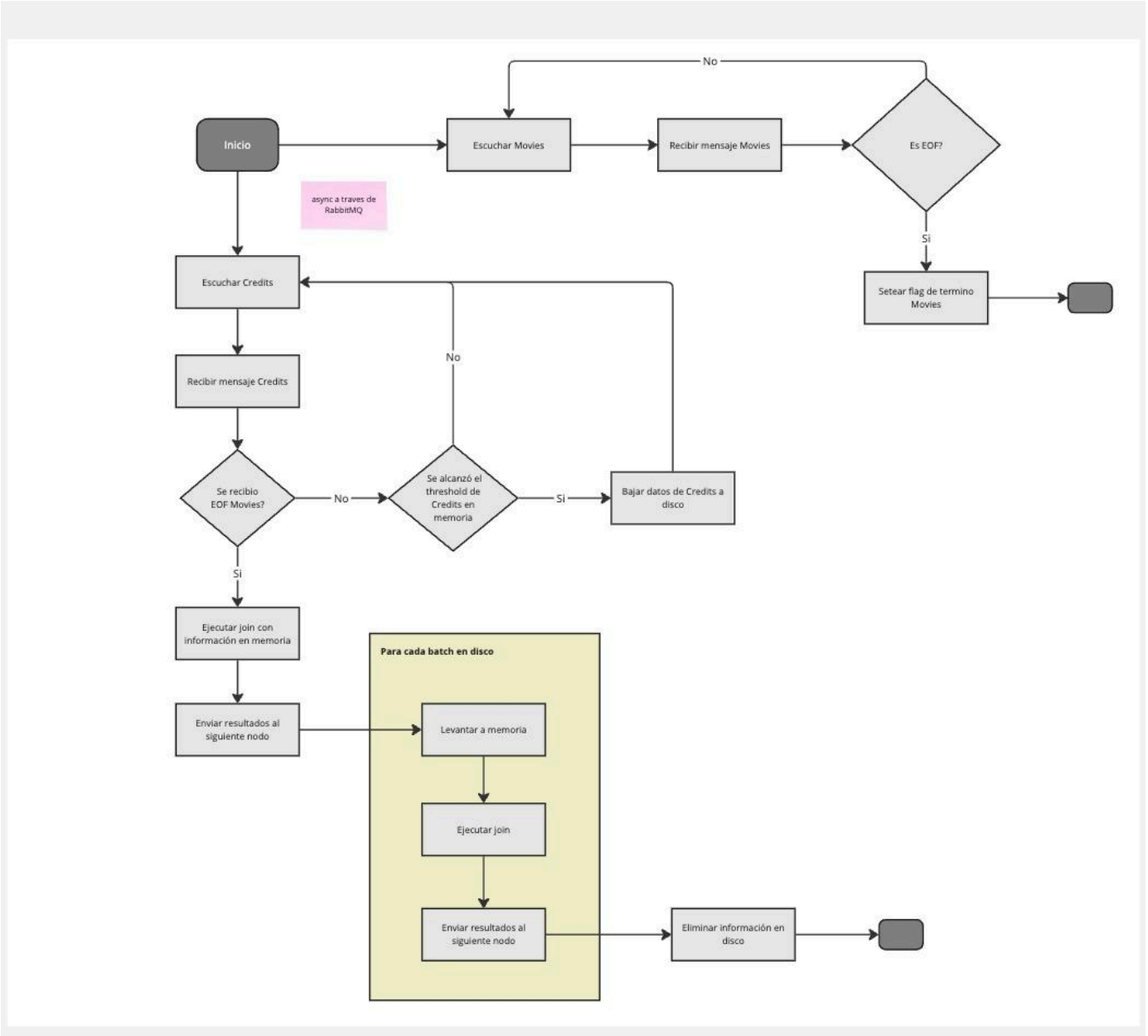
1. Conexión TCP.
2. Envío de datos en batch.
3. Procesamiento de la consulta.
4. Envío de resultados al cliente.



Optimización de la memoria

Para optimizar el uso de la memoria y evitar cargar tanta información en memoria implementamos la lógica que se da el diagrama. Cuando empezamos a procesar una consulta todo lo que llega de Movies lo persistimos en memoria. Por otro lado todo lo que llega de Credits o Ratings memoria, hasta que llegue el EOF de Movies y podamos ejecutar la consulta, o hasta que se supere un threshold configurable. Cuándo esto último ocurre,

bajamos a disco toda la información que tengamos en ese momento de Credits o Ratings. Esto se repite hasta que llega el EOF de movies, y en ese momento procesamos todo lo que tenemos en memoria, y vamos levantando de a poco todo lo que tenemos cargado un disco, para evitar una sobrecarga en memoria. De esta forma reducimos al mínimo posible el consumo de memoria de cada consulta.



Tolerancia a Fallos

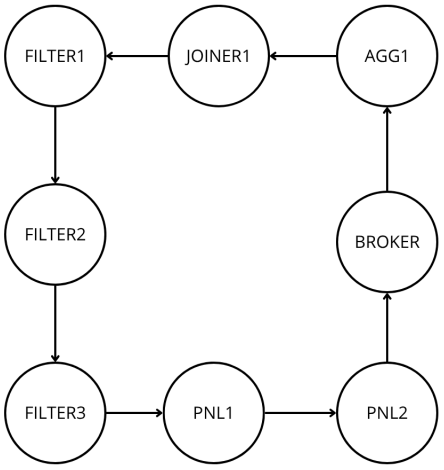
Cada nodo tendrá dos procesos paralelos ejecutándose, uno es el sistema ya conocido de manejo de consultas, el otro será el del procesamiento de un objeto llamado HealthMonitor,

este objeto, como su propio nombre indica, será el encargado de verificar la salud de los nodos.
Los nodos están conectados en una estructura de anillo, en el que se enviaran mensajes utilizando conexiones UDP para agilizar el envío y recepción de mensajes sin conexión. Cada nodo está conectado sólo con su nodo siguiente, al que cada X tiempo le enviará un “heartbeat”. Si luego de un tiempo un nodo no recibe un heartbeat de su predecesor, entenderá que éste sufrió una caída, por lo que procede a hacerle un restart.
Este diseño presenta una serie de ventajas y desventajas en comparación con una alternativa basada en nodos externos dedicados exclusivamente al monitoreo y reinicio (por ejemplo, nodos health).

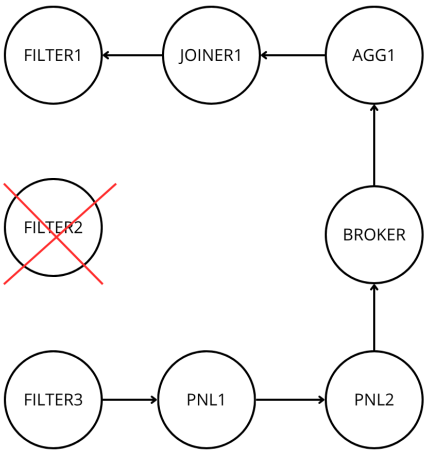
- Ventajas:
    - Menor complejidad estructural: No se incorporan nodos adicionales al sistema, lo que reduce la carga de mantenimiento, configuración y consumo de recursos.
    - Responsabilidad distribuida: Cada nodo es autosuficiente y responsable de verificar el estado de su predecesor, promoviendo una forma de monitoreo descentralizada y escalable.
    - Reducción de puntos únicos de falla: Al no centralizar el control de disponibilidad en nodos especializados, se evita que la caída de estos afecte la capacidad de recuperación del sistema.
- Desventajas:
    - Mayor acoplamiento entre nodos: Cada nodo debe poseer el conocimiento y las credenciales necesarias para reiniciar a su predecesor mediante Docker-in-Docker, lo cual introduce una dependencia adicional y potencialmente sensible en términos de seguridad.

A continuación se muestran unos diagramas para explicar la secuencia en la topología de anillo.

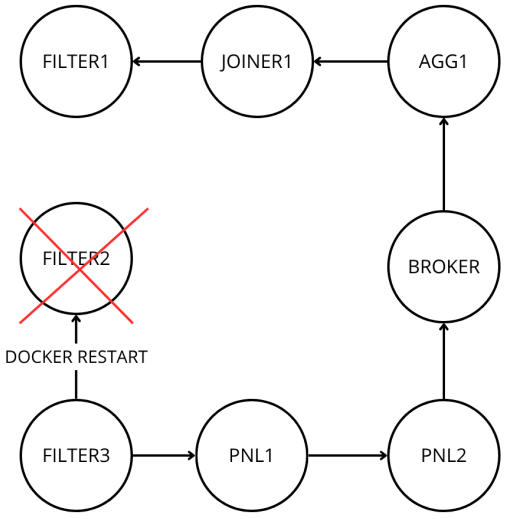
Aclaración: tanto el input gateway como el output gateway forman parte del anillo, el diagrama es ilustrativo para explicar el algoritmo.



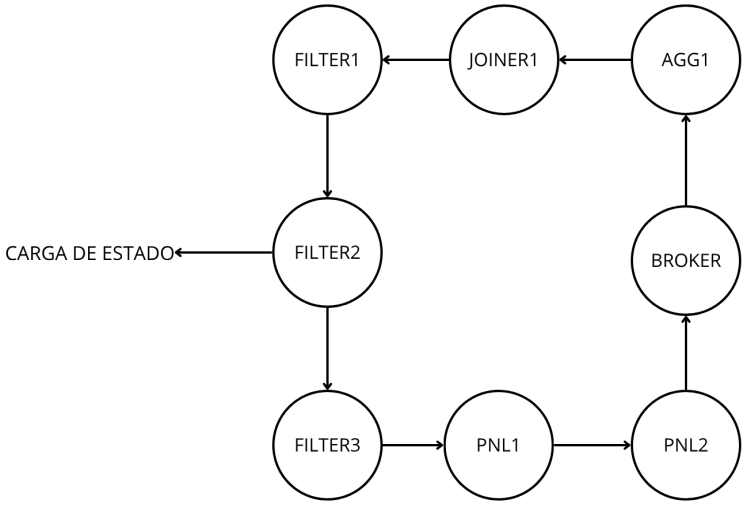
Este sería un ejemplo de estado inicial de la topología, con 2 nodos Pnl, 3 Filter, 1 Aggregator, 1 Joiner y 1 Broker.



En determinado momento, el nodo Filter2 sufre una caída, por lo que Filter3 dejará de recibir mensajes de heartbeats.



Luego de un tiempo configurable (necesario para no confundir caídas con posibles demoras en la red) el Filter3 hará un Restart de su nodo predecesor.



Al volver a la normalidad, Filter2 realizará una carga de su estado para volver al mismo punto en el que sufrio la caida, (en este caso no haría nada ya que Filter es un nodo Stateless), a continuación se ahonda sobre esta carga de estado y el almacenamiento del mismo.

Guardado y Carga de Estado

Cada nodo implementa un mecanismo de persistencia basado en commits para garantizar la recuperación de su estado en caso de una caída. Cada vez que el nodo realiza una operación que implique un cambio en su estado interno (por ejemplo, almacenar datos en memoria o modificar estructuras ya existentes), se genera un commit persistente que registra dicha modificación.

Al reiniciarse, el nodo verifica si existe un archivo de estado previamente creado. Si este archivo está presente y contiene información, el nodo interpreta que debe recuperar su estado anterior, para lo cual recorre secuencialmente las instrucciones registradas, ejecutándose nuevamente en el orden en que fueron persistidas. De esta manera, el nodo es capaz de restaurar su estado tal como estaba antes del fallo.
Este mecanismo se apoya en tres archivos diferenciados:

- Archivo de estado completo: contiene todas las operaciones significativas que alteran el estado del nodo. Cada línea representa una operación y está identificada por un sufijo que indica el tipo de modificación. Por ejemplo, una actualización sobre los EOF esperados tendrá un commit como:
  id_cliente|nro_consulta|-1|/E/n
  donde el sufijo /E indica una modificación sobre los EOF esperados.

Ejemplos

- **Linea 1:** **8c5b797f-57a8-412f-a555-880850da1c49|ratings|{'batch_id': '0', 'datos': [{'id': '110', 'rating': '1.000000'}]}|368/D**
- **Linea 2:** **8c5b797f-57a8-412f-a555-880850da1c49|ratings|False/K**

- Archivo de última acción: es un archivo más liviano (una sola línea) en el que se almacena información sobre la última acción realizada:
  El tipo de acción
  El mensaje asociado
  Un flag indicando si el mensaje debe ser enviado o no

  Ejemplo
  1538||no_enviar/C

- Archivo de último batch id: también es un archivo liviano, de una sola línea, en el que se almacena por cliente y por consulta el id del último batch cargado a disco:
  El tipo de acción
  El mensaje asociado
  Un flag indicando si el mensaje debe ser enviado o no

  Ejemplo (formateado para que sea más visible, pero es una única línea)

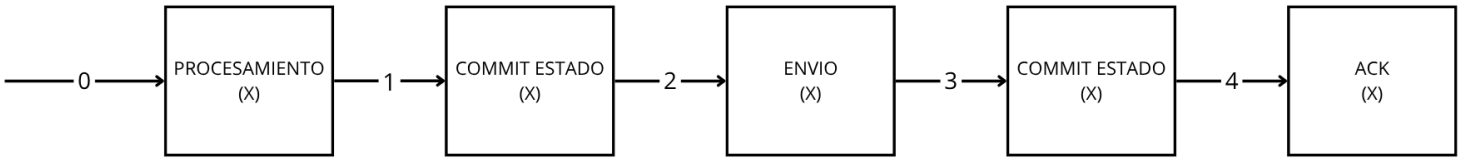
```
{
  "8c5b797f-57a8-412f-a555-880850da1c49": {
    "ratings": 4552,
    "credits": 4988
  }
}/B
```

El segundo archivo permite evitar duplicaciones en caso de reinicio. Si una operación ya fue procesada pero no se había enviado su resultado, el nodo la puede reenviar directamente sin repetir el procesamiento. En cambio, si la operación ya fue procesada y enviada, el nodo se limitará a realizar el acknowledgement correspondiente, evitando el reenvío y reduciendo la posibilidad de inconsistencias. **Este proceso se explicará en detalle más adelante**.

El tercer archivo nos permite no cargar a memoria todo el estado completo, sino únicamente lo que había en memoria al momento de la caída.

Mitigación de Fallos

Para explicar estas secuencias se mostrará un ejemplo de los momentos en los que un nodo puede sufrir una caída entre sus operaciones, y la forma de mitigar las mismas.
Dado un mensaje X recibido por un nodo, y considerando que pueden producirse caídas en distintos momentos del flujo de procesamiento, se identifican cinco posibles puntos de fallo (del 0 al 4). A continuación se detalla el impacto de cada uno:



0. Caída antes de procesar el mensaje X:
Sin consecuencias. El nodo, al reiniciarse, no encuentra ninguna acción asociada al mensaje en sus archivos, por lo que vuelve a procesar X desde el inicio.

1. Caída durante el procesamiento de X, antes de realizar el commit:
Tampoco hay pérdida de información. Al reiniciarse, el nodo no tiene persistido ningún cambio relacionado con X, por lo que volverá a procesar el mensaje completo. La única consecuencia es una pequeña pérdida de tiempo por duplicar el trabajo.

2. Caída después del commit, pero antes del envío del mensaje:
Sin consecuencias. El nodo encuentra en su archivo de última acción que ya se realizó el commit del mensaje X y que su estado indica **ENVIAR**. Por lo tanto, al reiniciarse, procede a enviar el mensaje y luego realizar el acuse de recibo (ack).

3. Caída justo después del envío del mensaje, pero antes de actualizar la flag a **NO_ENVIAR**:
Este es el punto más delicado del flujo. El mensaje X fue efectivamente enviado, pero el nodo aún tiene registrada la acción como **ENVIAR**. Tras reiniciarse, volverá a enviar el mismo mensaje, lo que puede provocar duplicación. Este escenario representa la ventana de fallo más crítica del sistema, aunque su probabilidad de ocurrencia es baja. Se considera aceptable dentro de los límites de tolerancia establecidos.

4. Caída después del envío del mensaje y de actualizar la flag a **NO_ENVIAR**:
Sin consecuencias. Al reiniciarse, el nodo reconoce que ya procesó y envió el mensaje X, y que su flag está correctamente actualizada como **NO_ENVIAR**. En consecuencia, no se produce un reenvío, y solo se realiza el acknowledgement en caso de ser necesario.

Mitigación de Fallos en el Input Gateway

El input gateway también implementa lógica específica para ser tolerante a fallos, tanto por caídas del nodo, como desconexiones de los clientes. Al igual que los otros nodos, persiste un archivo de estado, donde registra para cada cliente las consultas activas. Ante una caída, el nodo recupera su estado, y envía los end of files correspondientes a cada consulta, para poderlas dar por finalizadas, y liberar los recursos asociados. Además, ante una desconexión del lado del cliente, esta misma lógica es aplicada.

Ejemplo del archivo de estado

```
{
  "4b668253-e3a6-47f4-bd98-149a184d2e48": {
    "RATINGS|TOP-ARGENTINIAN-MOVIES-BY-RATING": {}
  }
}
```

Prevención de duplicados

Para garantizar la integridad y la idempotencia cada mensaje es enviado desde el cliente con un id de batch asignado (**oportunidad de mejora**). Este id se propaga en el header de cada mensaje a lo largo de todo el flujo, desde el input gateway hacia los nodos.

Cada nodo (joiner, aggregator y pnl) lleva un registro del último mensaje que procesó. Es por esto que antes de procesar un nuevo mensaje se valida que el id sea mayor al último recibido. En caso de qué no lo sea, el mensaje es descartado, únicamente enviando el ack correspondiente.

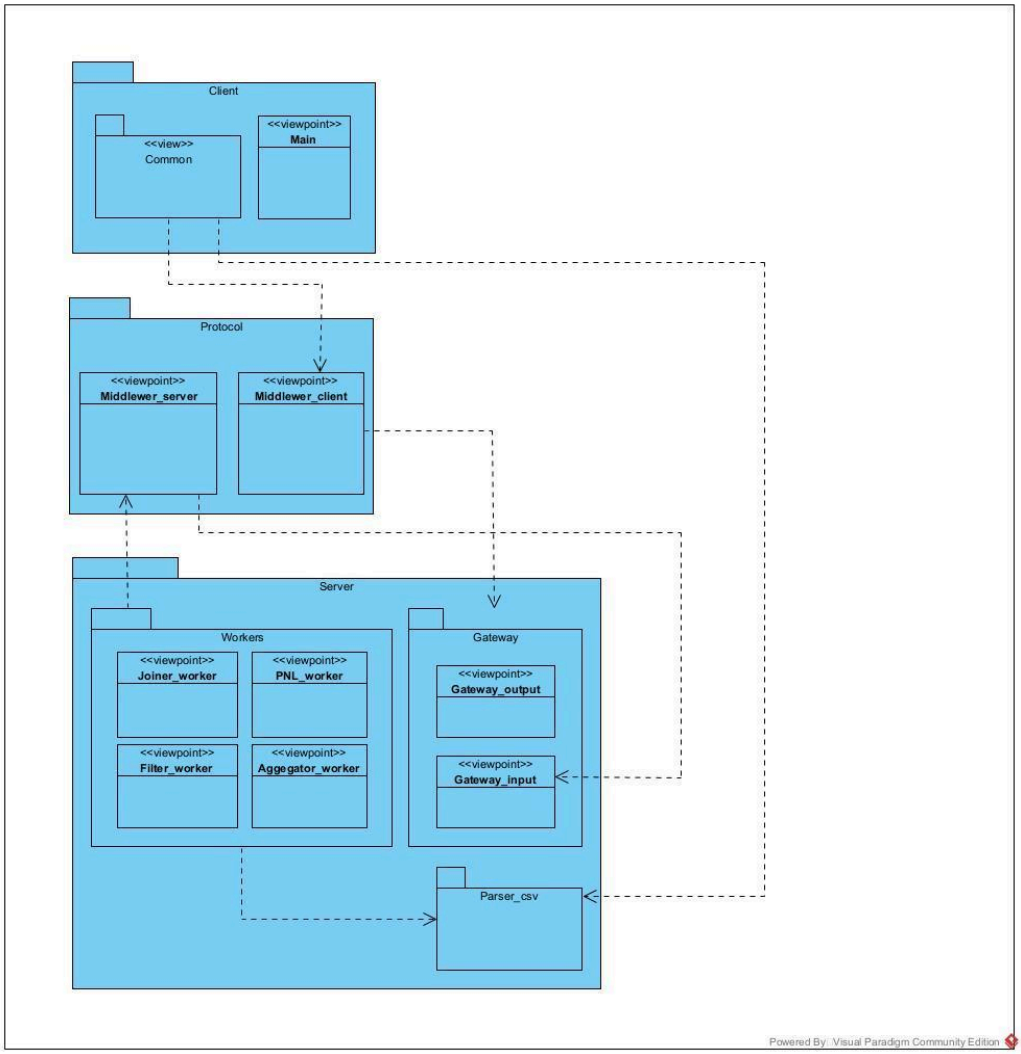
Este header se propaga con el nombre `message_id`. Pero además, también se propaga otro header `batch_id`, el cual se persiste junto con la información de cada batch en el nodo Join, para poder generar un nuevo `message_id` con el concatenado de todos los batch que se procesen al momento de ejecutar la consulta.

Vista de Desarrollo

En nuestro sistema, se agrupan los componentes en paquetes funcionales bien definidos:

- **Client**: contiene el código de interfaz de consola y lógica del cliente.
- **Protocol**: abstrae la lógica de comunicación entre cliente y servidor, con módulos para cada extremo.
- **Server**: agrupa todos los módulos del lado servidor. Está subdividido en:

    - **Workers**: cada uno especializado en una tarea de procesamiento.
    - **Gateway**: maneja la entrada y salida de mensajes.
    - **Common**: Posee objetos y funciones utilizadas por todos los workers.
    - **Parser_csv**: encargado de leer y procesar los archivos de dataset.



Escenarios

Caso de uso: Consulta 1
Se ejecuta una consulta tipo 1.
El Gateway Input publica el mensaje en la cola Filter_Request_1.
Un nodo Filter que atiende la consulta 1 procesa los datos.
Publica el resultado en la cola de Gateway Output.
El Gateway Output recibe el resultado y lo reenvía al cliente.

Caso de uso: Consulta 2
Se ejecuta una consulta tipo 2.
El Gateway Input publica el mensaje en la cola Filter_Request_2.
Un nodo Filter que atiende la consulta 2 realiza el procesamiento.
Envía el resultado al Aggregator 2.
El Aggregator reúne los datos y publica el resultado en la cola del Gateway Output.
El Gateway Output devuelve la respuesta al cliente.

Caso de uso: Consulta 3
Se ejecuta una consulta tipo 3.
El Gateway Input publica el mensaje en la cola Filter_Request_3.
Un nodo Filter procesa y extrae las condiciones iniciales.
El Gateway también envía los batches de credits y ratings al Broker.
El Broker reparte los datos a los nodos Joiner por sus colas individuales.
Los Joiner procesan y envían los resultados al Aggregator 3.
El Aggregator junta los datos y publica el resultado en Gateway Output.
El Gateway Output devuelve la respuesta al cliente.

Caso de uso: Consulta 4
Se ejecuta una consulta tipo 4.
El Gateway Input publica el mensaje en la cola Filter_Request_4.
Un nodo Filter procesa el mensaje.
El Gateway publica los batches de credits y ratings en el Broker.
El Broker reparte a los Joiner (uno por cola).
Los Joiner unen y procesan la información.
Envían resultados al Aggregator 4.
El Aggregator junta y publica en Gateway Output.
El Gateway Output devuelve el resultado al cliente.

Caso de uso: Consulta 5
Se ejecuta una consulta tipo 5.
El Gateway Input publica el mensaje en la cola Filter_Request_5.
Un nodo Filter procesa la parte inicial.
El Gateway envía batches al Broker.
El Broker distribuye a los nodos Pnl para análisis de sentimiento.
Cada Pnl procesa y publica su resultado en la cola del Aggregator 5.
El Aggregator reúne todo y lo envía a Gateway Output.
El Gateway Output responde al cliente.

Caída de un nodo Joiner luego de procesar pero antes del envío
El nodo Joiner se reinicia, ve que el último mensaje fue commiteado y su flag es ENVIAR.
Lo envía nuevamente y continúa el flujo.
El Aggregator podrá recibir el mensaje sin errores (acepta duplicados si ya no recibió ese batch).

Caso de uso: Caída después del envío con flag NO_ENVIAR
Un nodo Joiner procesa un mensaje de una consulta.
Realiza el commit de la operación y envía el resultado al Aggregator correspondiente.
Inmediatamente después, actualiza su archivo de última acción marcando la flag como NO_ENVIAR.
Antes de enviar el ack, el nodo sufre una caída.
Al reiniciarse, el nodo analiza su archivo de última acción.
Detecta que el mensaje ya fue procesado, ya fue enviado, y que su flag es NO_ENVIAR.
Por lo tanto, no vuelve a enviar el mensaje para evitar duplicados.
Solo realiza el acknowledgement (ack) pendiente hacia el nodo que espera confirmación.

Caída justo después de enviar pero antes de cambiar la flag (ENVIAR a NO_ENVIAR)
El nodo reiniciado volverá a enviar el mensaje.
Se puede producir duplicado.
El Aggregator o receptor debe manejar duplicados como tolerables.
Este es el único caso que puede generar redundancia, y se acepta como una ventana mínima de error.

Caso Graceful shutdown ante SIGTERM

Dado un nodo que está procesando mensajes
Cuando recibe una señal SIGTERM del sistema operativo
Entonces:

- El nodo deja de consumir nuevos mensajes de su cola.
- Termina de procesar los mensajes que ya recibió.
- Envía los ACK correspondientes si el procesamiento fue exitoso.
- Cierra la conexión con RabbitMQ de forma controlada.
- Libera recursos y finaliza el proceso sin pérdida de datos.

Oportunidades de mejora

Una de las oportunidades de mejora identificadas es la responsabilidad sobre la generación del id de batch (`batch_id`) que se propaga en el input gateway hacia los nodos. En nuestra implementación es el cliente quien asigna el valor. Sin embargo, sería más adecuado que esta responsabilidad sea del input gateway. Esto nos permitiría desacoplar la lógica de identificación de los batches de la lógica de los clientes, y tener una estructura más robusta que no dependa de que el cliente envíe correctamente estos ids.

Actualmente nuestro sistema nos permite escalar la cantidad de nodos Aggregator, pero estamos limitados a 4 nodos, como máximo, debido a la cantidad de consultas. Se podría configurar a 4 nodos por default, y eliminar del código toda la lógica para diferenciar que consulta está atendiendo cada uno, lo cual volvería el código mucho más claro y simple. Cada nodo tendría una responsabilidad mucho más acotada y definida.

En el Joiner persistimos un archivo donde almacenamos toda la información procesada. Sin embargo, no tenemos un mecanismo de limpieza ni eliminación de información histórica para una vez que las consultas ya han sido completadas exitosamente. Esto provoca que el archivo crezca indefinidamente y funcione como un histórico completo. Lo cual implica:

- Ineficiencia al reiniciarse: cada vez que el nodo deba cargar su estado se ve obligado a procesar todo el archivo completo, incluso cuando hay información que no es relevante para las consultas activas.
- Incremento del up-time: cada vez tarda más en levantarse el nodo debido a la cantidad de información.
- Desperdicio de recursos: mantener y parsear información antigua que ya no se utiliza implica un consumo innecesario de I/O, CPU y memoria.