



Documento de Arquitectura

Sistema Distribuido de Análisis de Películas

Grupo: 3

Integrantes:

- Camila Belén Sebellin - 100204
- Nathalia Lucia Encinoza Vilela - 106295
- Juan Pablo Fresia - 102396

Tabla de Contenidos

1. Alcance.....	4
1.1. Descripción general del sistema.....	4
1.2. Objetivos del sistema.....	4
1.3. Funcionalidades específicas.....	4
1.4. Requisitos no funcionales.....	5
2. Arquitectura.....	7
2.1. Estilo arquitectónico.....	7
2.2. Componentes principales.....	7
3. Objetivos y Restricciones.....	9
3.1. Objetivos.....	9
3.2. Restricciones.....	9
4. Escenarios.....	11
5. Vista de Procesos.....	13
5.1. Diagramas de secuencia.....	13
5.2. Diagramas de actividades.....	19
5.2.1. Funcionamiento del proxy.....	19
5.2.1.2. Conexiones y reconexiones de gateways.....	19
5.2.1.2. Conexiones de clientes.....	23
5.2.2. Inicio de nodo y recuperación.....	26
6. Vista de Desarrollo.....	27
6.1. Diagrama de Paquetes.....	27
7. Vista Física.....	30
7.1. Diagrama de Robustez.....	30
7.2. Diagrama de Despliegue.....	38
7.2.1. Descripción de Nodos.....	38
7.2.2. Importancia del MOM Broker en la Arquitectura.....	42
8. Direct Acyclic Graph (DAG).....	44
8.1. Descripción General.....	44
8.2. Componentes del DAG.....	45
9. Escalabilidad.....	48
9.1. Gestión de mensajes EOS.....	48
9.2. Escalado de nodos.....	50
9.2.1. Pruebas con Threads vs. Procesos.....	51
9.2.1.1. Actualización por tolerancia a fallos.....	52
10. Multi-Client.....	53
10.1. Gestión de proceso de mensajes para múltiples clientes.....	53
10.1.1. ClientManager.....	53
10.1.2. ClientState.....	53
10.1.2.1. Actualización por tolerancia a Fallos.....	54
10.1.3. Manejo de Side Tables para clientes.....	54
10.2. Adaptación del Gateway.....	55
10.2.1. Separación de responsabilidades.....	55
10.2.2. Manejo explícito de nodos del sistema.....	56
10.2.3. Mejor manejo del estado.....	57
10.2.3.1. Actualización por tolerancia a Fallos.....	57
11. Tolerancia a fallos.....	58

<i>11.0 Recepción de películas mediante sharding en los nodos Join.....</i>	58
11.0.1 Configuración de Sharding.....	58
11.0.2. Proceso de recepción de películas.....	58
11.0.2.1 Filtrado por Rango.....	58
11.0.2.2 Almacenamiento en Memoria.....	58
11.0.2.3 Persistencia a Disco.....	59
11.0.2.4 Manejo de Mensajes EOS.....	59
11.0.3 Coordinación desde el Nodo Líder.....	59
<i>11.1. Separación de contenedores: Sistema y Clientes.....</i>	59
<i>11.2. Introducción de nodo Coordinator.....</i>	60
<i>11.3. Introducción de nodo Proxy.....</i>	61
11.3.1. Arquitectura del proxy.....	62
11.3.1.1. Hilo de escucha de clientes.....	62
11.3.1.2. Hilo de escucha de gateways.....	63
11.3.2. Tolerancia a Fallos de Gateways.....	64
11.3.2.1. Identificación y Reconexión.....	64
11.3.2.2. Protocolo de Reconexión.....	64
11.3.2.3. Restauración de Clientes Suspensos.....	65
11.3.2.4. Robustez ante errores.....	65
11.3.2.5. Coordinación con el Proxy.....	66
<i>11.4. Tolerancia a fallos de los nodos Gateways.....</i>	66
11.4.1. Reconexión y recarga de estado de clientes por parte del Gateway.....	66
11.4.2. Persistencia de batches y respuestas.....	67
<i>11.5. Tolerancia a fallos en nodos de procesamiento.....</i>	67
11.5.1. Elección de Líder.....	67
11.5.1.1. Criterio de Liderazgo.....	67
11.5.1.2. Identificación y Comunicación entre Nodos.....	68
11.5.1.2.1. UDP.....	68
11.5.1.2.2. Fuente de la Configuración: Variables de Entorno vía Docker Compose..	68
11.5.1.3. Flujo de Inicialización.....	68
11.5.1.4. Mecanismo de Heartbeat.....	69
11.5.1.4.1. Funcionamiento General.....	69
11.5.1.4.2. Implementación con Threads.....	69
11.5.1.4.3. Robustez frente a falsos positivos.....	70
11.5.1.5. Recepción y Procesamiento de Mensajes.....	70
11.5.1.6. Proceso de Elección.....	70
11.5.1.6.1. Verificación Preliminar.....	70
11.5.1.6.2. Envío de Mensajes ELECTION.....	71
11.5.1.7. Anuncio del Coordinador.....	71
11.5.1.8. Detección y Manejo de Fallos.....	71
11.5.1.9. Escenarios.....	71
Escenario 1: Startup Normal (Inicio Simultáneo de Nodos).....	71
Escenario 2: Fallo del Líder.....	72
Escenario 3: Fallo de un Nodo que NO es el Líder.....	73
Escenario 4: Recuperación de Nodo Fallido.....	73
11.5.2. Lógica Master-Slave.....	74
11.5.2.1 Balanceo de carga.....	74
11.5.2.2. Manejo y recuperación de EOS (End Of Stream).....	76
11.5.2.3. Fallo del nodo líder durante la redistribución.....	76

11.5.2.4. Control de duplicados.....	76
12. Iteraciones futuras.....	78
13. Anexo.....	79
13.1. Referencias.....	79
13.2. Diagramas completos.....	79
13.2.1. Diagrama de robustez completo.....	80
13.2.2. Diagrama DAG completo.....	81
13.2.3. Diagrama de despliegue completo.....	82
13.2.4. Diagrama de actividades: Proxy - Gateway listener completo.....	83
13.2.5. Diagrama de actividades: Proxy - Clients listener completo.....	84

1. Alcance

Este documento describe la arquitectura del sistema distribuido de análisis de películas basado en datasets públicos (IMDb). El sistema procesa consultas sobre películas, actores y ratings mediante una arquitectura distribuida diseñada para escalar horizontalmente en entornos multicomputadora.

1.1. Descripción general del sistema

El sistema tiene como objetivo procesar y analizar grandes volúmenes de datos relacionados con películas, actores y valoraciones de usuarios, provenientes de fuentes públicas como IMDb. Se propone una arquitectura distribuida capaz de ejecutar consultas complejas sobre los datos, maximizando el rendimiento y la escalabilidad en entornos multicomputadora.

La solución estará orientada al procesamiento distribuido, aprovechando la concurrencia y paralelismo para lograr eficiencia y permitir una expansión horizontal sin necesidad de rediseños estructurales.

1.2. Objetivos del sistema

- Procesar datos a gran escala de forma eficiente mediante distribución de carga en múltiples nodos.
- Permitir la ejecución de consultas específicas sobre datos de películas, actores y ratings.
- Diseñar una arquitectura extensible y mantenible, capaz de adaptarse al crecimiento de volumen de datos y usuarios.

1.3. Funcionalidades específicas

El sistema deberá resolver las siguientes consultas sobre los datasets:

- Películas y sus géneros de los años 2000 a 2009 con coproducción de Argentina y España.
- Top 5 de países que más presupuesto han invertido en producciones sin participación de otros países.
- Determinación de las películas de producción Argentina, con fecha de estreno posterior al año 2000, con el mayor y menor promedio de rating.
- Top 10 de actores con mayor participación en películas argentinas estrenadas a partir del año 2000.
- Comparación del promedio de la tasa ingreso/presupuesto entre películas con sentimiento positivo y negativo en la sinopsis.

1.4. Requisitos no funcionales

El sistema fue diseñado con una serie de requisitos no funcionales que aseguran su robustez, escalabilidad, adaptabilidad y resiliencia en entornos distribuidos. En primer lugar, debe estar optimizado para funcionar eficientemente en entornos multicomputadora, distribuyendo las tareas de procesamiento entre múltiples nodos de cómputo de forma paralela y coordinada. Esta capacidad permite aprovechar al máximo los recursos disponibles, reduciendo los tiempos de ejecución globales.

Se exige también el soporte para escalabilidad horizontal, permitiendo que se puedan incrementar la cantidad de nodos en cualquier etapa del procesamiento sin comprometer la integridad ni la disponibilidad del sistema. Para ello, se diseñó una arquitectura modular, donde cada componente puede ser replicado dinámicamente según parámetros definidos en un archivo de configuración centralizado, manteniendo la consistencia en el procesamiento.

Otro requisito esencial es que el sistema debe ejecutar una única instancia del procesamiento por cada consulta recibida. Durante su ejecución, debe ser capaz de responder adecuadamente ante señales del sistema como SIGTERM, finalizando de forma ordenada mediante un mecanismo de *graceful shutdown* que garantice la correcta liberación de recursos y el cierre seguro de las conexiones.

En cuanto a la comunicación entre componentes, se desarrolló un middleware personalizado que abstrae la lógica de intercambio de mensajes entre procesos. Este middleware soporta comunicación basada en grupos lógicos, facilitando la construcción de flujos de procesamiento complejos sin acoplar firmemente a los workers entre sí.

Adicionalmente, se consideró la necesidad de soportar múltiples ejecuciones por parte de un mismo cliente sin necesidad de reiniciar el servidor. Esto implica que el sistema debe estar preparado para recibir múltiples solicitudes secuenciales, reutilizando los recursos disponibles sin degradar su rendimiento. En ese mismo sentido, también se prevé la posibilidad de ejecutar múltiples consultas de manera concurrente por parte de diferentes clientes, sin interferencias ni bloqueos entre ellas.

Un aspecto fundamental en la operación del sistema es la correcta limpieza de recursos luego de cada ejecución. Esto incluye el cierre de conexiones, el vaciado de colas, y la eliminación de estados internos que pudieran interferir con futuras ejecuciones.

Respecto a la tolerancia a fallos, el sistema debe ser capaz de detectar la caída de procesos y recuperarse automáticamente para mantener la continuidad del servicio y la integridad de los datos. Para la reactivación de procesos caídos, está permitido el uso de entornos de virtualización anidados, como docker-in-docker. Sin embargo, la detección y monitoreo del estado y disponibilidad de los nodos distribuidos debe ser implementada internamente en el sistema, sin delegar esta función a la plataforma de contenedores Docker. En caso de ser necesario el uso de algoritmos de consenso para mantener la consistencia entre nodos, dichos algoritmos deben ser diseñados, implementados y validados íntegramente por el equipo desarrollador, sin recurrir a bibliotecas externas o soluciones preconstruidas.

2. Arquitectura

2.1. Estilo arquitectónico

El sistema adopta un enfoque basado en **microservicios distribuidos**, donde cada componente cumple un rol bien definido, se comunica con otros a través de canales desacoplados y puede ser desplegado, escalado o reiniciado de forma independiente.

La arquitectura se encuentra fuertemente **orientada a eventos**, utilizando un **Message Broker (RabbitMQ)** para la intermediación y el desacoplamiento de los procesos. Las distintas etapas del procesamiento interactúan mediante colas lógicas que aseguran persistencia, orden y reintentos en caso de fallos.

El diseño promueve la **escalabilidad horizontal**, especialmente en el nivel de procesamiento, mediante un conjunto de **workers especializados**, organizados en grupos funcionales. Estos workers consumen mensajes de colas particulares y ejecutan operaciones específicas, lo que permite paralelismo y distribución del trabajo sin necesidad de coordinación centralizada.

Se incluye además un **middleware personalizado** que abstrae la lógica de intercambio de mensajes, permitiendo a los componentes enviar y recibir datos sin acoplamiento directo a las primitivas de bajo nivel del sistema de colas. Este middleware gestiona el enrutamiento, formateo y envío de los datos, además de soportar la creación de flujos de procesamiento dinámicos entre grupos de nodos.

2.2. Componentes principales

Componente	Descripción
Middleware de Comunicación	Abstracción sobre RabbitMQ que encapsula la lógica de envío, recepción y ruteo de mensajes. Soporta comunicación en grupos, balanceo entre workers y control de flujo para batches parciales.

Workers de procesamiento	Nodos que realizan las transformaciones de datos según la funcionalidad requerida (por ejemplo: filtrado por año, análisis de sentimiento, agregación por producción). Están desacoplados, son stateless y se escalan horizontalmente.
Storage / Output Handler	Encargado de la persistencia de los resultados procesados. A su vez, reensambla los resultados parciales, detecta la finalización de cada consulta y los reenvía al gateway correspondiente.
Gateways	Nodos distribuidores encargados de recibir los batches de datos desde el proxy y enviarlas al middleware de procesamiento. Mantienen el estado parcial de cada ejecución y reenvían las respuestas al cliente.
Proxy	Punto de entrada y orquestador de las consultas de los clientes. Encapsula la lógica de multiplexado de conexiones, mantiene los estados de ejecución activos, distribuye las solicitudes a los gateways y reenvía las respuestas.
Coordinator	Nodo especial responsable de monitorear todos los contenedores del sistema. Detecta procesos caídos y los reinicia automáticamente, proporcionando tolerancia a fallos y disponibilidad continua.
Cliente	Proceso externo que envía los datasets al sistema y recibe los resultados finales. Su interfaz es simple y textual, comunicándose únicamente con el proxy.

3. Objetivos y Restricciones

3.1. Objetivos

- **Escalabilidad horizontal:**
Poder aumentar la cantidad de workers de manera sencilla para soportar mayor volumen de datos.
- **Tolerancia a fallos:**
RabbitMQ y el diseño desacoplado permiten que si un worker falla, otro pueda continuar el procesamiento.
- **Procesamiento distribuido eficiente:**
Distribuir la carga de trabajo de forma balanceada mediante RabbitMQ.
- **Middleware modular:**
Permite cambiar o extender el middleware sin impactar los componentes de negocio.
- **Graceful shutdown:**
Implementación de manejo de señales SIGTERM y SIGINT para que el sistema pueda finalizar las tareas en curso sin cerrar algún recurso (sockets, archivo, colas, etc.).
- **Fácil incorporación de nuevos requerimientos:**
Al agregar nuevos workers, es posible extender las funcionalidades del sistema sin modificaciones significativas de la arquitectura existente.

3.2. Restricciones

- **Dependencia de RabbitMQ:**
La arquitectura depende de la disponibilidad de RabbitMQ como sistema de mensajería central.
- **Dependencia de Proxy:**
La arquitectura depende de la disponibilidad del nodo Proxy como nexo entre los

distintos gateways y los clientes.

- **Procesamiento batch inicial:**

El sistema está diseñado para un flujo de procesamiento por lotes ("batch") inicial y no procesamiento en tiempo real de flujos continuos.

- **Consistencia eventual:**

Dado que se trata de un sistema distribuido asincrónico, los resultados se consolidarán de manera eventual, no inmediata.

- **Gestión del crecimiento del dataset:**

A medida que crezca el volumen de datos, será necesario escalar no solo los workers, sino también la infraestructura de soporte (almacenamiento, networking).

- **Suposiciones sobre tolerancia a fallas:**

- En cuanto a los nodos diseñados con replicas para tolerancia a fallas (coordinator, gateways, filtros y joins), se asume que al menos una réplica permanecerá activa ante una caída parcial, permitiendo conservar el estado necesario para enviárselo al potencial nuevo líder antes de que este sea designado.

4. Escenarios

A continuación se describen los escenarios de uso que el sistema debe cubrir. Cada uno se corresponde con las consultas mencionadas en el Alcance.

Escenario 1: Películas de los 2000s con producción Argentina y Española

Un analista desea obtener una lista de películas estrenadas entre los años 2000 y 2009, producidas por Argentina y España. Se requiere además identificar los géneros correspondientes a cada película. El sistema debe procesar los datos de películas y sus países de producción, filtrando por año y países involucrados.

Escenario 2: Países con mayor inversión sin co-producción

Se solicita el ranking de los 5 países que más dinero han invertido en producciones cinematográficas sin colaboración de otros países. El sistema debe analizar el presupuesto de cada película que haya sido producida por un único país y agrupar los valores para calcular el top 5.

Escenario 3: Películas argentinas post-2000 con mejor y peor rating promedio

Un periodista especializado en cine desea conocer cuál es la película argentina mejor valorada y cuál es la peor, considerando aquellas estrenadas a partir del año 2000. El sistema debe realizar un join entre los datos de películas argentinas y los ratings de usuarios para calcular los promedios y determinar los extremos.

Escenario 4: Actores con mayor participación en cine argentino post-2000

Una plataforma de streaming desea conocer los actores más recurrentes en producciones argentinas desde el año 2000, con el fin de optimizar sus recomendaciones. El sistema debe contar la frecuencia de aparición de cada actor en las películas argentinas post-2000, utilizando los datos de créditos.

Escenario 5: Análisis económico y de sentimiento

Se busca analizar la eficiencia económica de las películas, contrastando aquellas con resumen argumental de sentimiento positivo versus negativo. El sistema debe realizar un

análisis de sentimiento sobre el overview de cada película, calcular el ratio ingreso/presupuesto, y obtener el promedio de este valor para cada categoría de sentimiento.

5. Vista de Procesos

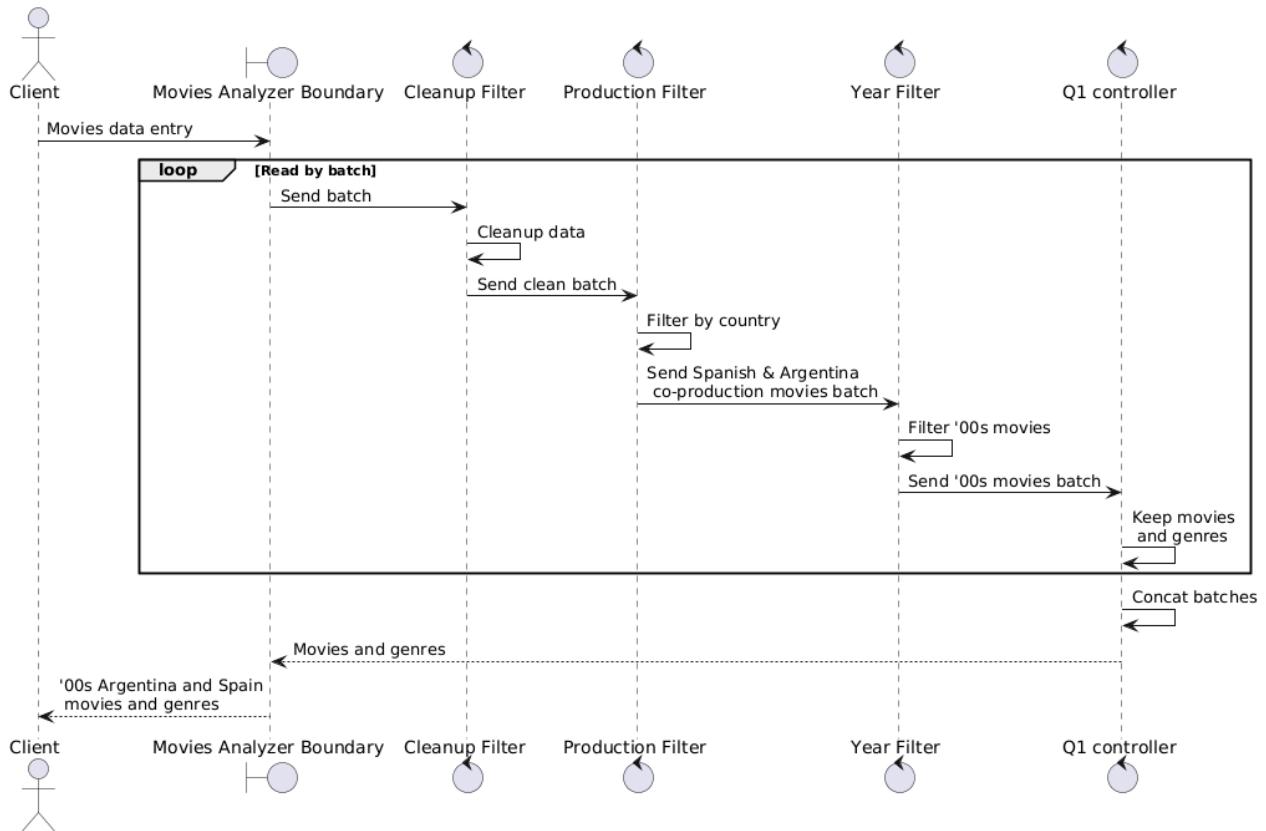
La vista de procesos describe la dinámica de ejecución del sistema y su comportamiento en tiempo de ejecución. En el contexto de este sistema, la vista se centra en cómo se distribuye el procesamiento de datos entre distintos nodos computacionales, considerando aspectos como la concurrencia, la comunicación entre componentes y la coordinación de tareas.

Dado que el sistema está orientado a ambientes multicomputadora, las consultas se resuelven a través de una arquitectura basada en colas de mensajes, donde distintos procesos especializados actúan de forma cooperativa y desacoplada. Esto permite tanto la escalabilidad como la tolerancia a fallos ante interrupciones controladas mediante señales como SIGTERM.

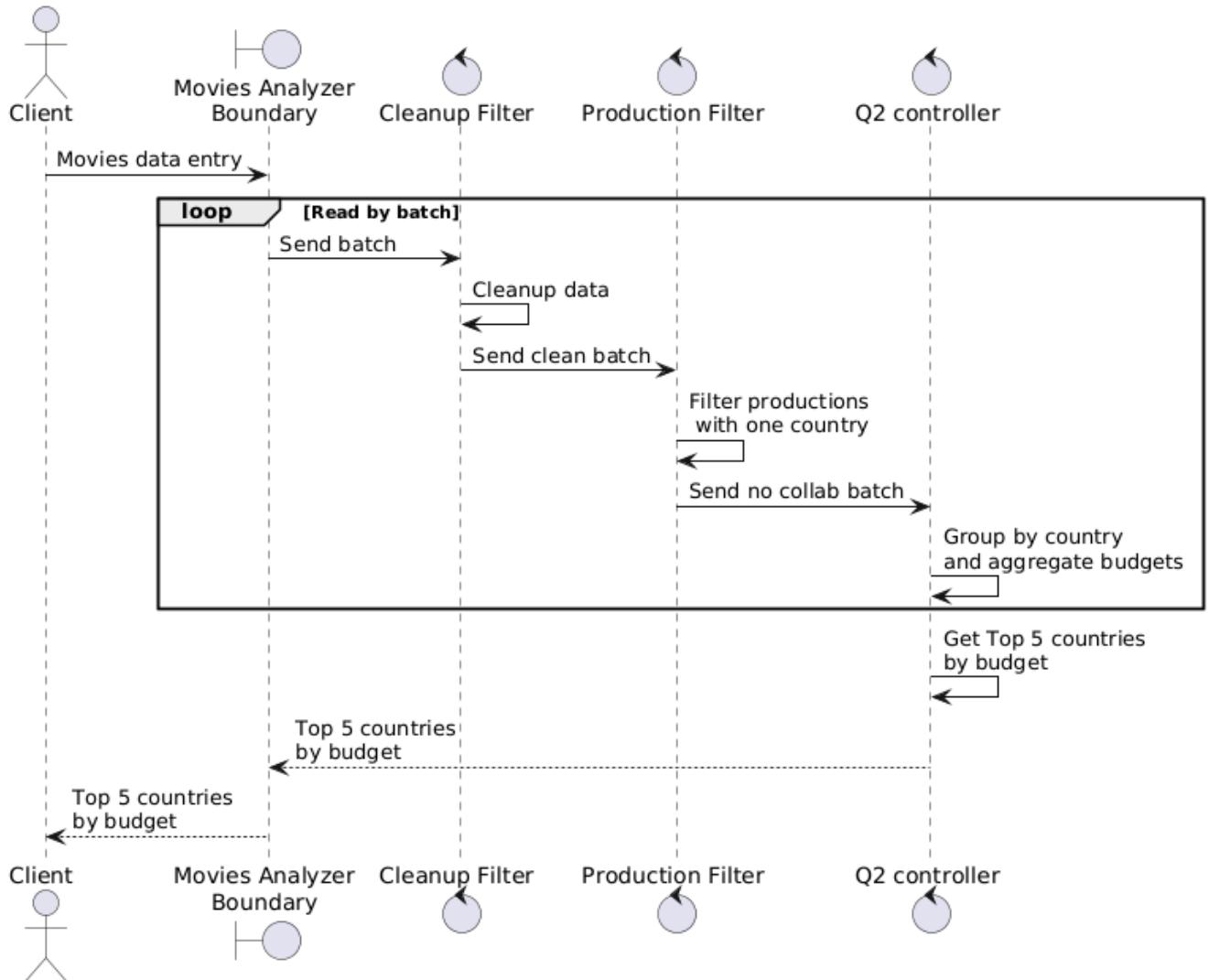
5.1. Diagramas de secuencia

A continuación, se presentan los diagramas de secuencia correspondientes a cada uno de los escenarios funcionales definidos, detallando la interacción entre procesos, colas y componentes del sistema para la resolución de cada consulta:

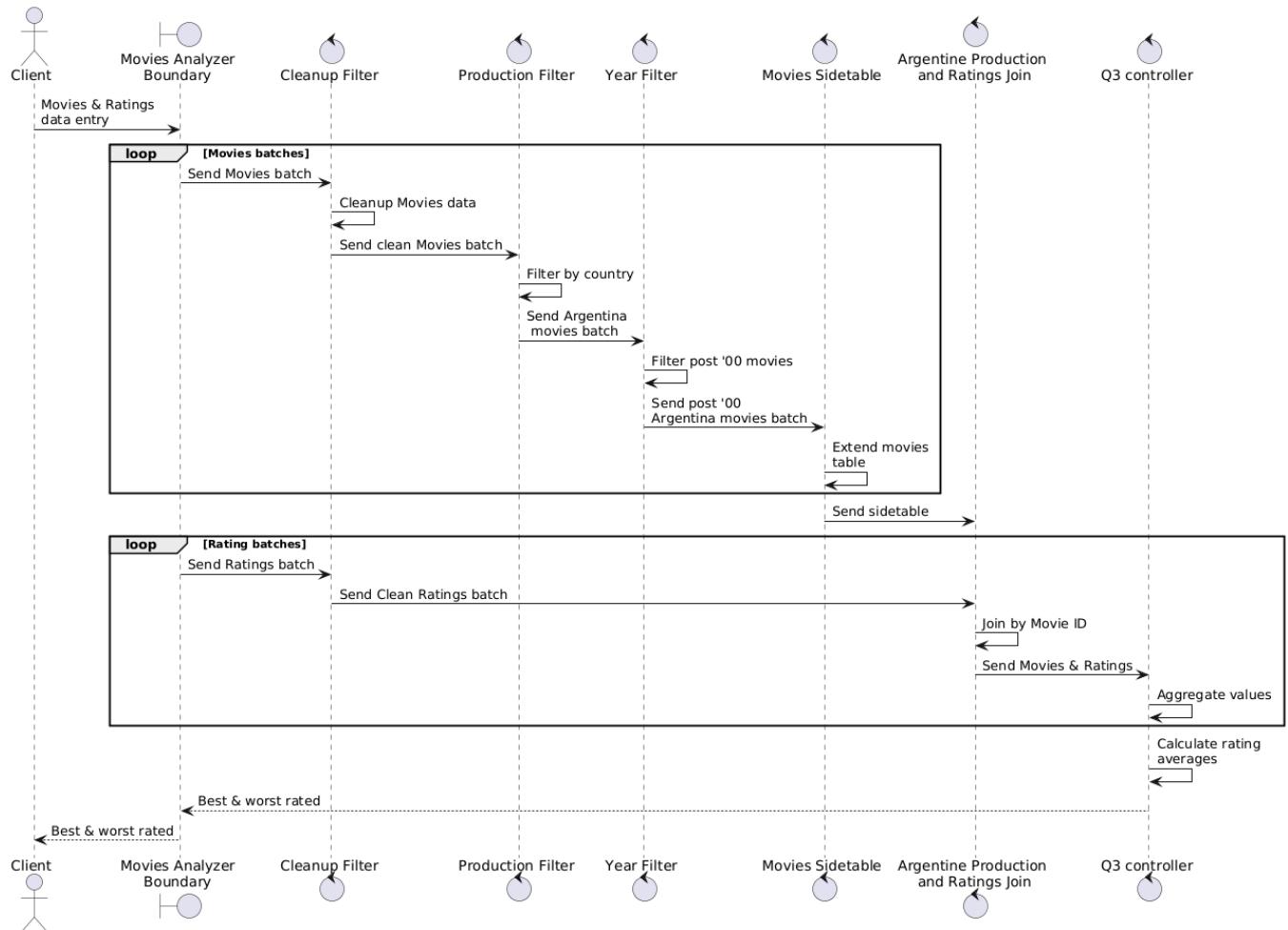
- **Escenario 1:** Películas y sus géneros de los años 00' con producción Argentina y Española



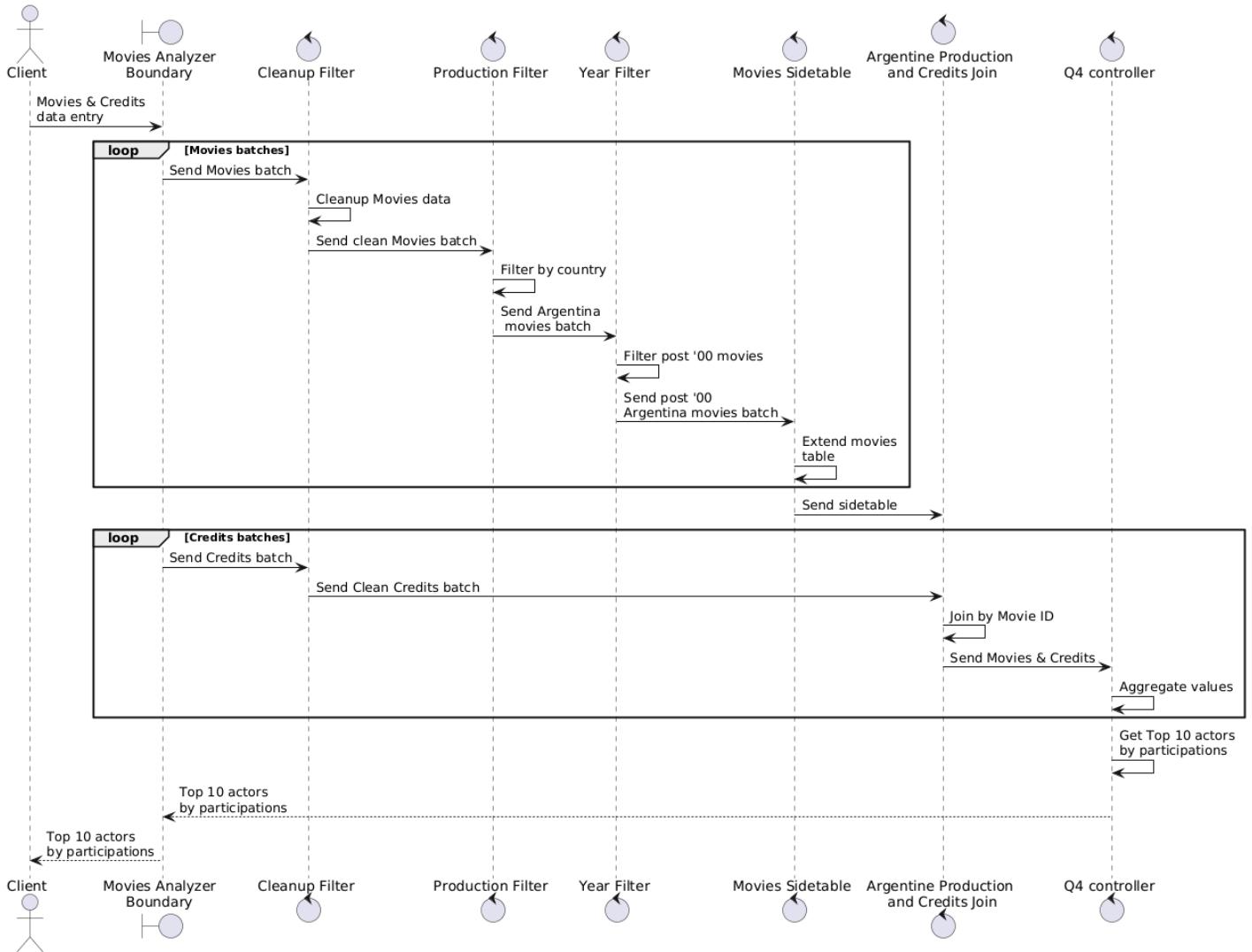
- **Escenario 2:** Top 5 de países que más dinero han invertido en producciones sin colaboración con otros países



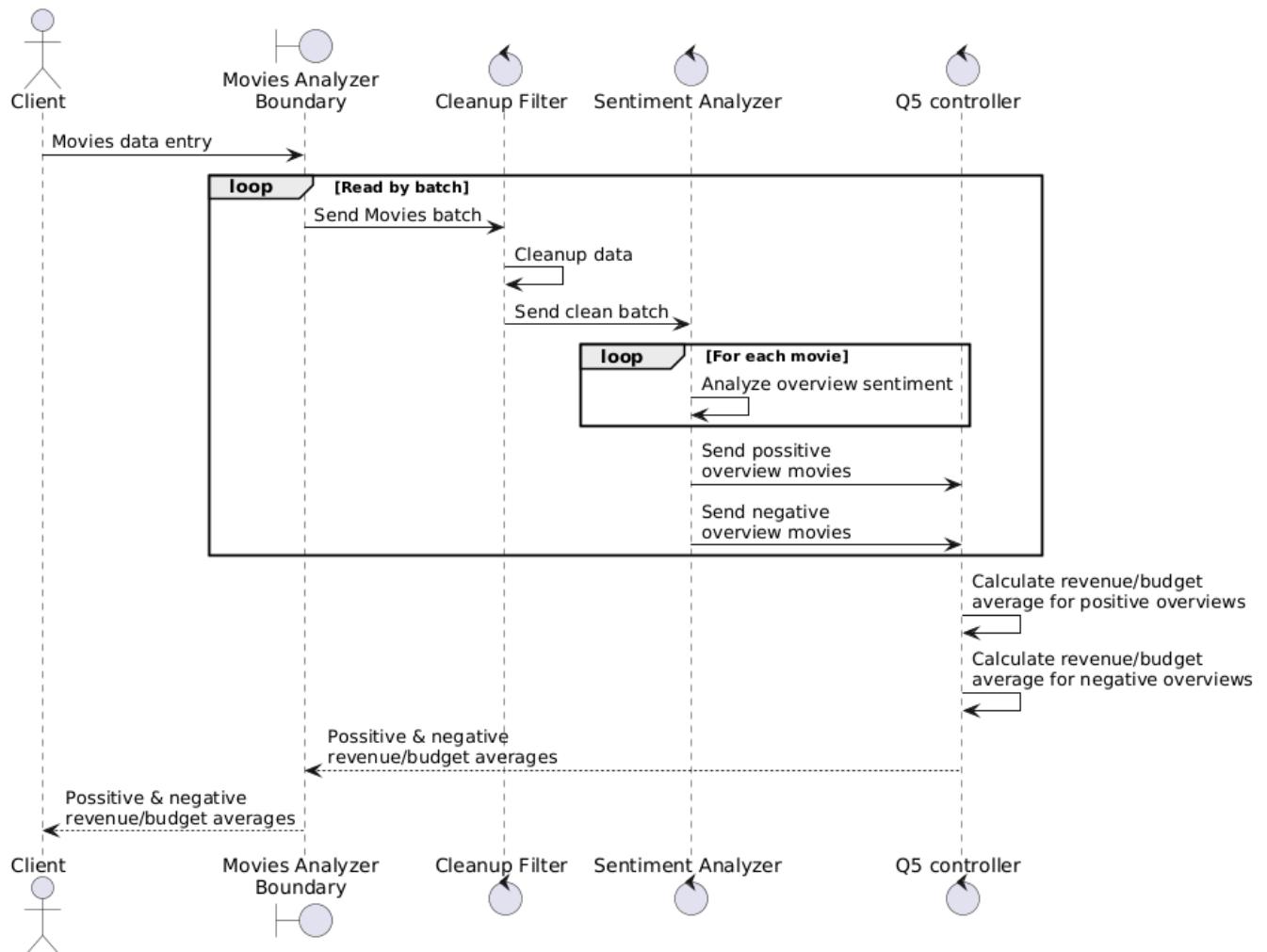
- **Escenario 3:** Película de producción Argentina con mayor y menor promedio de rating



- **Escenario 4:** Top 10 de actores con mayor participación en películas argentinas post-2000



- **Escenario 5:** Promedio del ratio ingreso/presupuesto para películas según sentimiento del overview



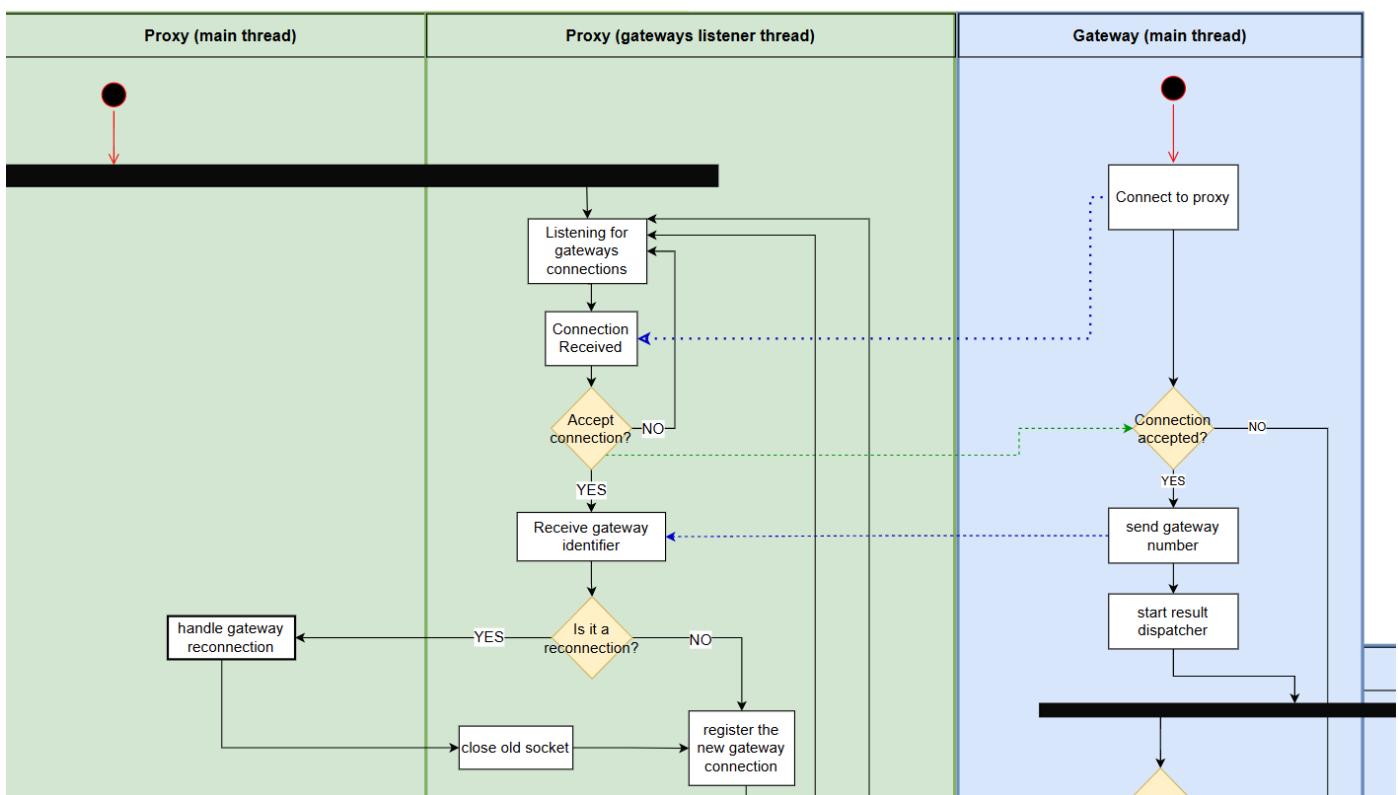
Cada diagrama refleja los pasos clave de procesamiento, incluyendo etapas de filtrado, transformaciones, joins entre datasets, análisis de sentimiento y almacenamiento de los resultados en una cola final de salida.

5.2. Diagramas de actividades

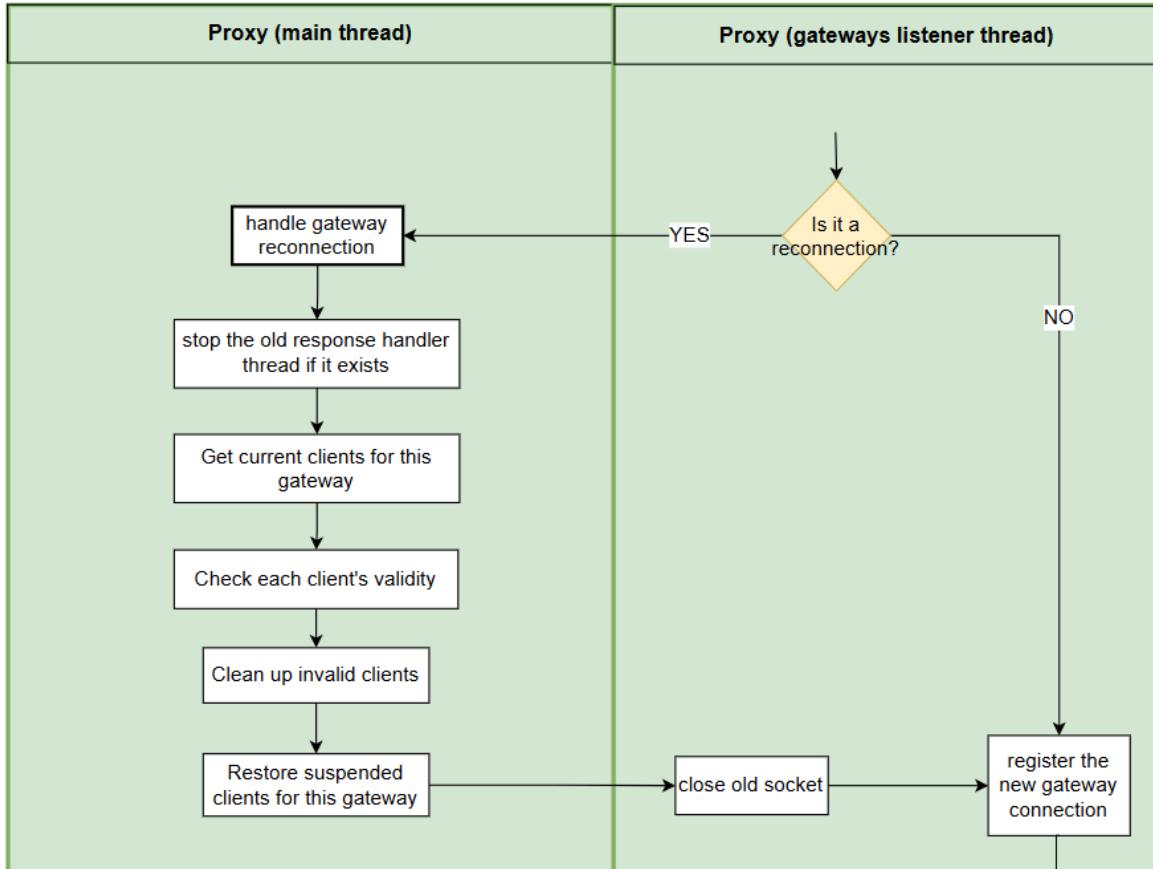
5.2.1. Funcionamiento del proxy

5.2.1.2. Conexiones y reconexiones de gateways

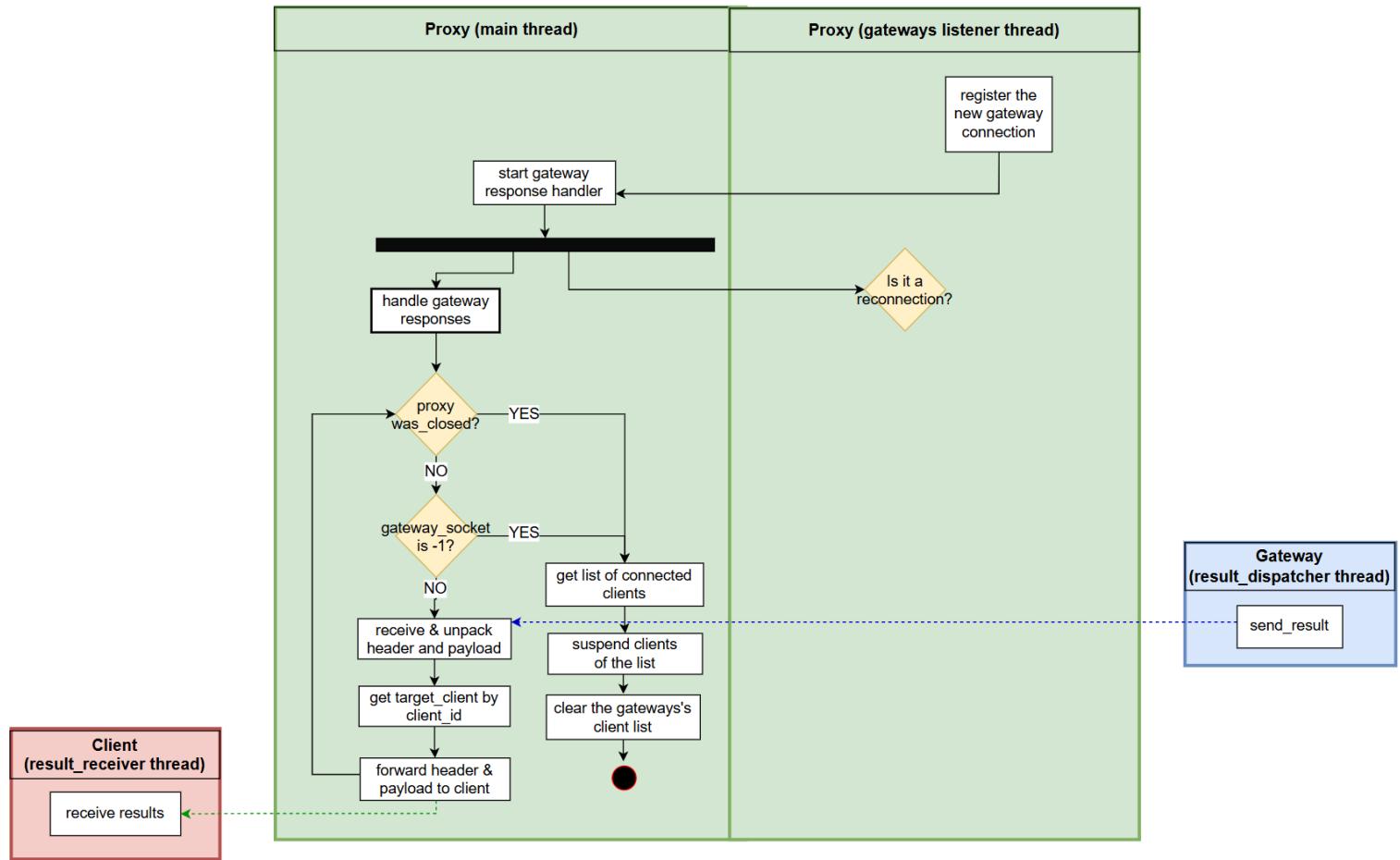
En este apartado se detalla el ciclo completo de conexión de un nodo gateway al nodo proxy, incluyendo tanto la primera conexión como el proceso de reconnexión en caso de fallo. El proxy escucha de forma pasiva en un puerto específico, acepta nuevas conexiones entrantes desde gateways, valida su identidad y registra la conexión en su tabla interna. En el siguiente recorte, podremos ver la lógica común que tienen tanto las conexiones nuevas como las reconexiones: toda conexión (sea nueva o reconnexión) debe antes que nada identificarse con su número de gateway.



Si el gateway listener detecta que es un número de ID que ya tenía registrado, lo toma como una reconnexión por lo que llama a la función `handle_gateway_reconnection` del proxy y cierra el socket anterior asociado a ese gateway. En la siguiente imagen haremos un “zoom” a la función `handle_gateway_reconnection` del proxy para ver lo que hace internamente para manejar las reconexiones de los clientes asociados al gateway antes de haberse desconectado:

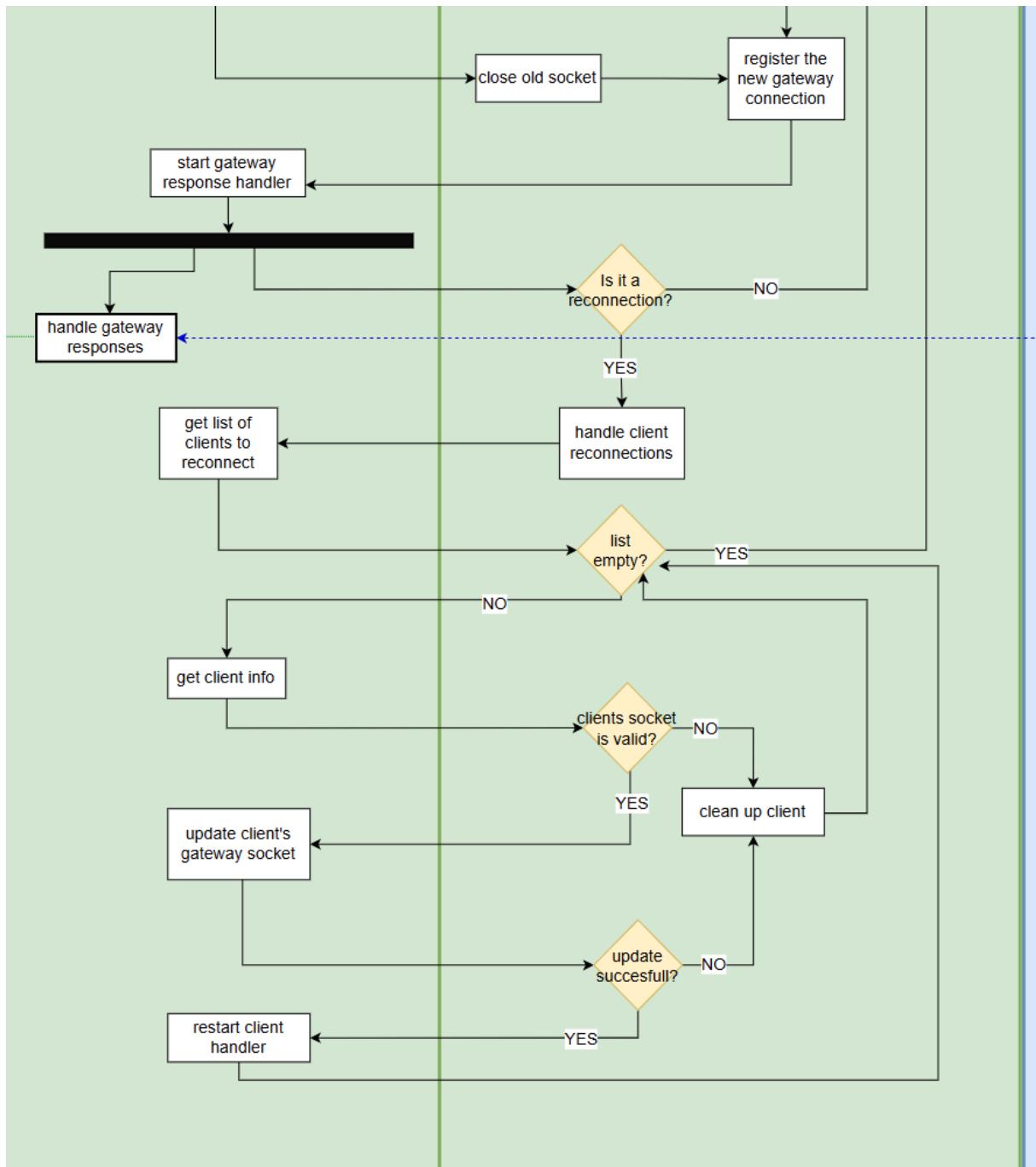


Una vez que ambos casos registraron su nueva conexión, se inicia el gateway_response_handler del proxy. Esta función es clave, ya que se encarga del forwarding de respuestas del gateway al cliente. En la próxima imagen, haremos énfasis sobre esta etapa. Aquí podremos ver como los 3 componentes (Gateway, Proxy y cliente) se comunican efectivamente:



Mientras el hilo del proxy quedará corriendo manejando los paquetes a forwardear del gateway al cliente, tendremos el hilo original del gateways_listener que continuará con el proceso de conexión/reconexión. En la siguiente porción del diagrama, podremos ver como:

- En caso de ser una conexión nueva, se termina el proceso y se vuelve al inicio del ciclo para seguir escuchando por nuevas conexiones de Gateways.
- En caso de ser una reconexión, se pasa reconectar todos los clientes que estaban conectados al anterior Gateway. Este proceso implica, primero verificar que los sockets de los clientes sigan siendo válidos (es decir, que los clientes sigan conectados). En caso de no serlo, se limpian y se continua con el siguiente. En caso de que el socket sea válido, se continua por actualizarles el socket del gateway asociado con el nuevo. Si este cambio es exitoso, se reinicia el client_handler.



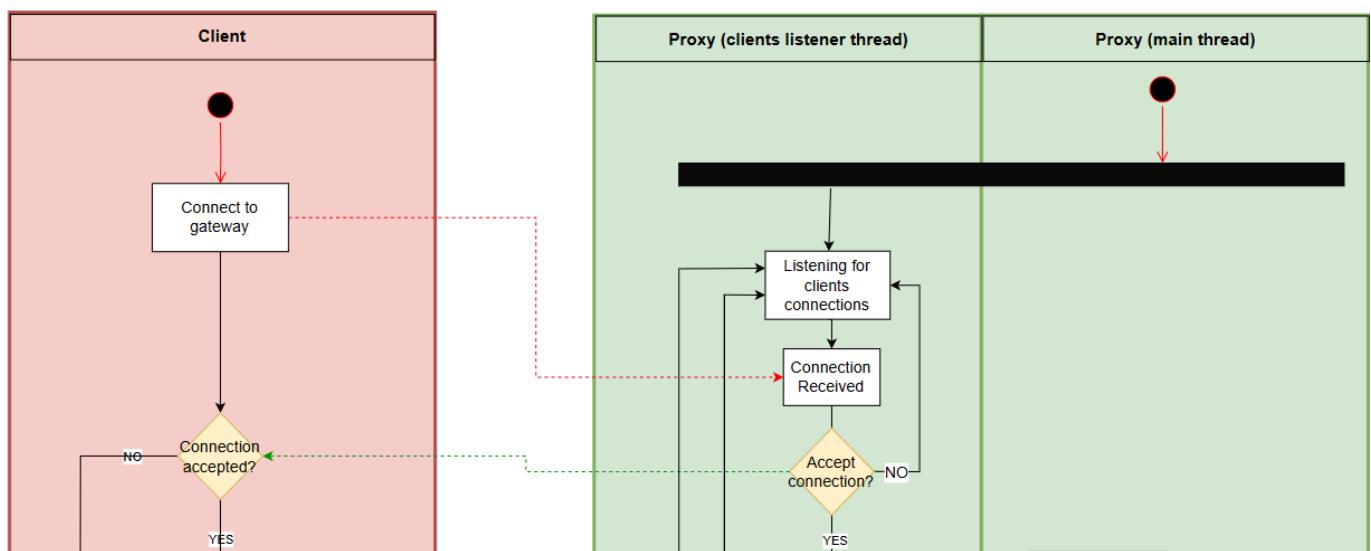
Una vez completada la lista, se vuelve a la escucha de conexiones.¹

¹ En el anexo puede encontrarse el diagrama completo (sección [13.2.4](#)).

5.2.1.2. Conexiones de clientes²

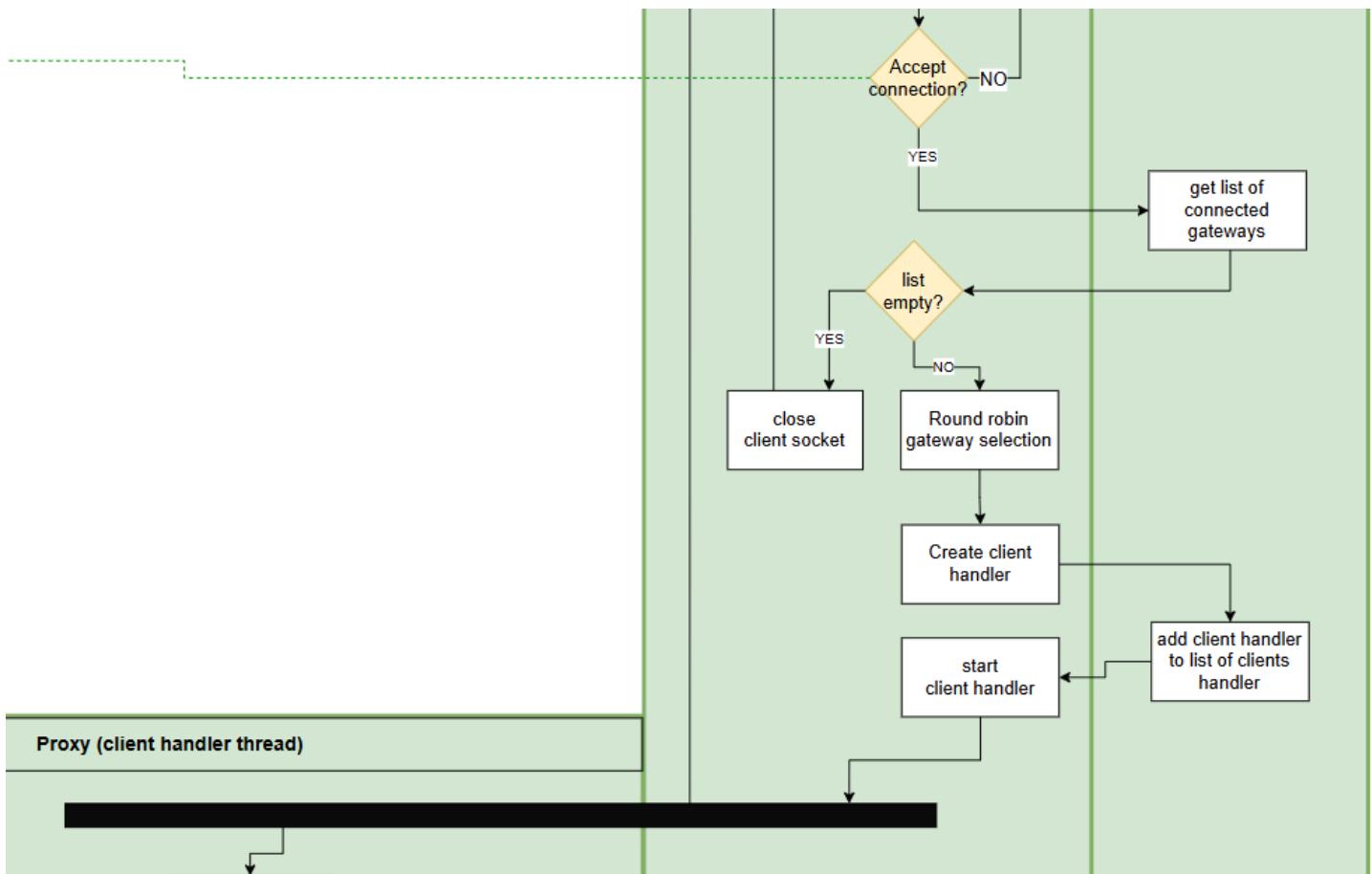
Esta sección ilustra cómo el proxy maneja las conexiones entrantes desde los distintos clientes. En particular, haremos el seguimiento cuando se conecta un cliente.

Una vez iniciado el Proxy, este inicializa el hilo de `clients_listener` que se encargará de recibir los pedidos de conexión de los clientes. Semejante a como lo hace el `gateways_listener`, este hilo escucha de forma pasiva en un puerto específico y va aceptando nuevas conexiones entrantes de clientes.

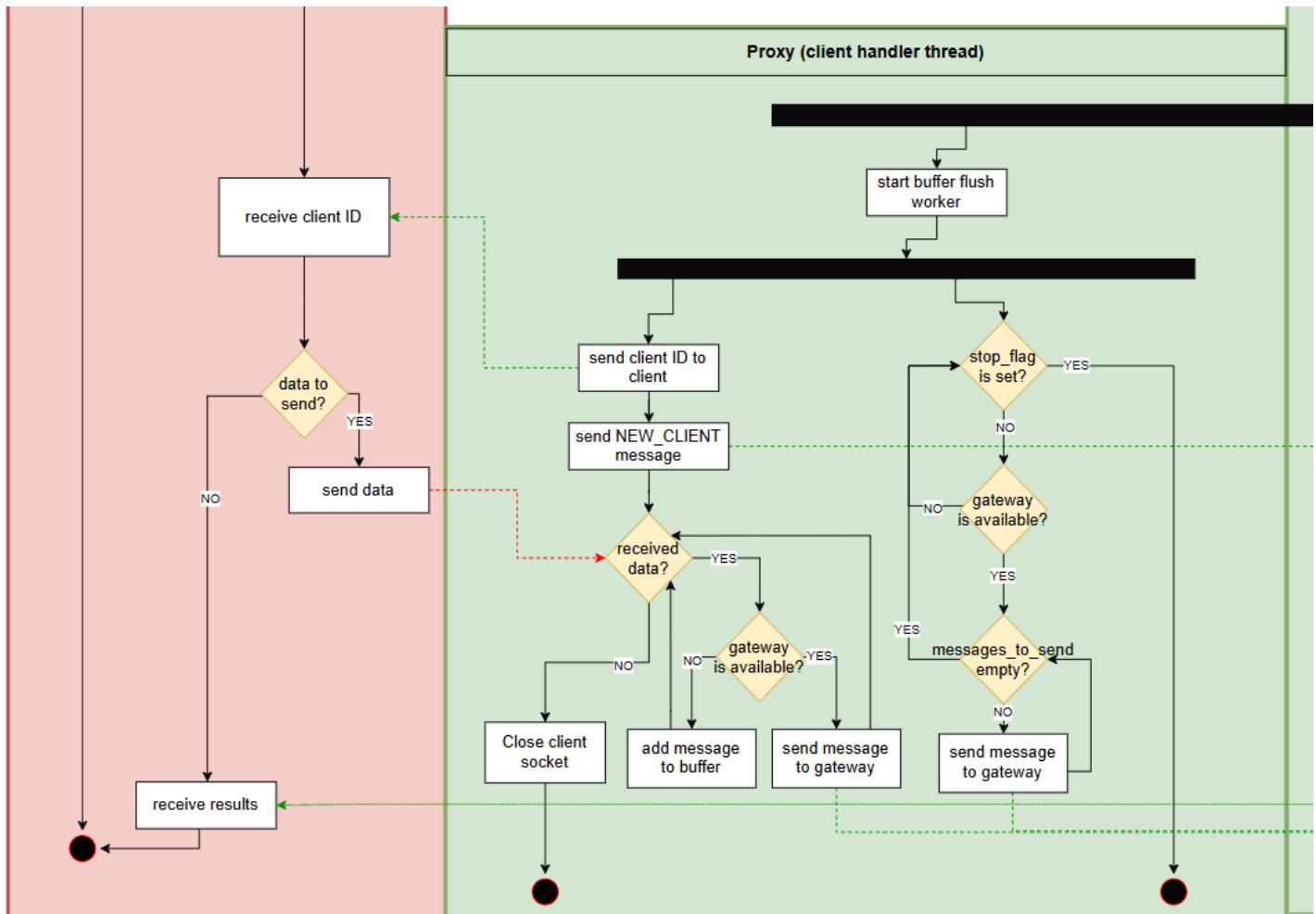


En caso de aceptar la conexión, pasamos a la siguiente etapa que se ve en la próxima imagen. El `clients_listener` le pide al hilo principal proxy que le de una lista de los gateways conectados. Mediante un algoritmo round-robin selecciona alguno de los gateways y crea una instancia del `ClientHandler` con el socket del cliente conectado y el gateway. Luego agrega esa instancia al diccionario de `_client_handlers` del proxy y le da start al hilo `client_handler`.

² En el anexo puede encontrarse el diagrama completo (sección [13.2.5](#)).

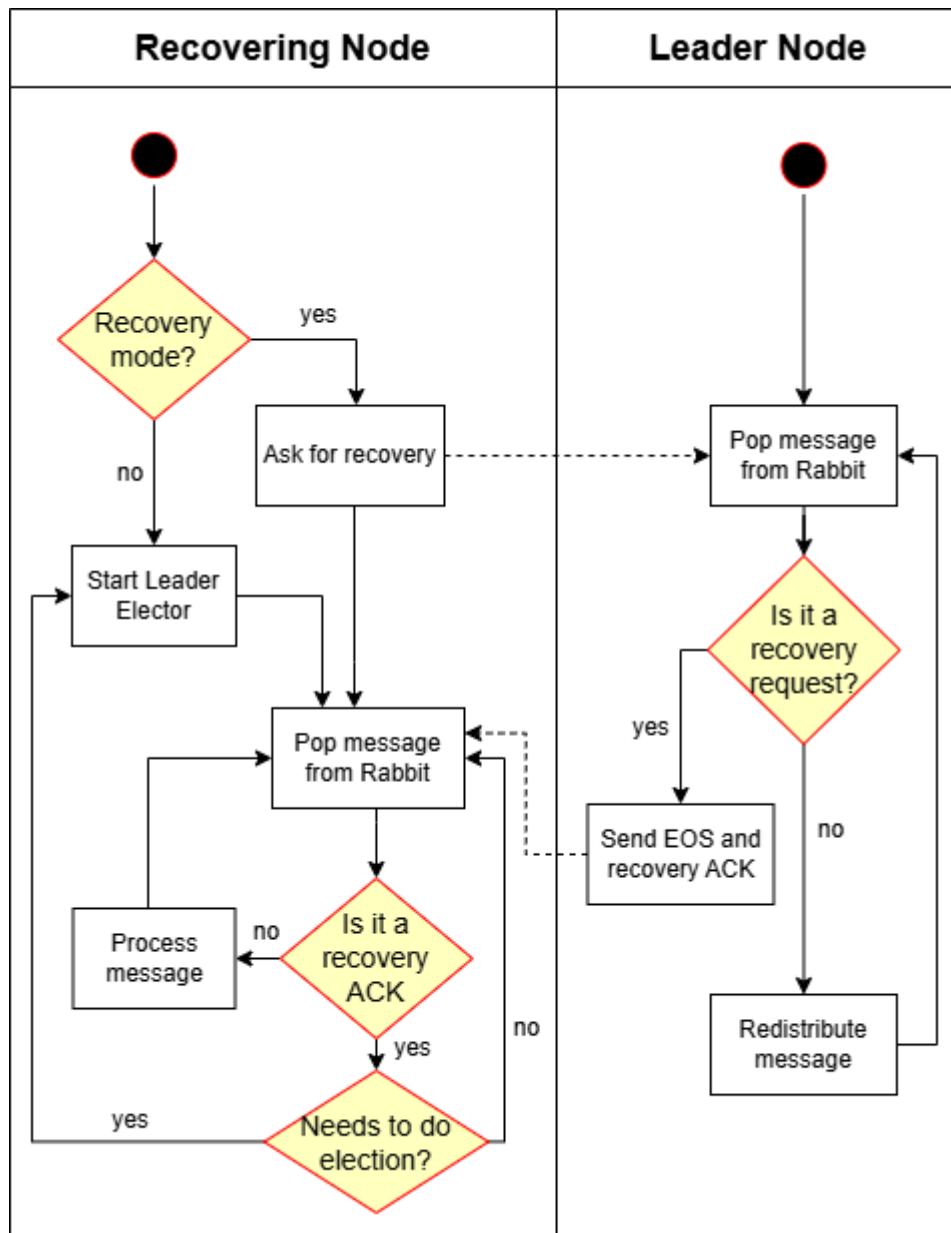


Cuando se le da inicio al hilo del client_handler, se le da inicio también a otro hilo encargado de enviar los mensajes almacenados en el buffer una vez este disponible el gateway. El hilo principal es el que se encarga de los pasos de registro y forwardeo “basico” de mensajes desde el cliente hacia el gateway conectado. La registración consiste en crear el client UUID y enviarlo tanto al cliente como al gateway (en el caso del gateway, se manda como un tipo de mensaje específico: “NEW_CLIENT”). Luego se iterá hasta que se mande la señal de stop al hilo donde se recibirán mensajes del cliente y se procederá a intentar enviarlo. En caso de que el envío falle, se almacenará ese mensaje en la cola buffer para que el otro hilo intente enviarlo.



Una vez que falle el receive (es decir, se corte por timeout o desconexión del cliente) o se mande la señal de stop al Proxy, se cerrará el socket del cliente y se procederá a la detención del hilo.

5.2.2. Inicio de nodo y recuperación



Como se mostró en el diagrama, el proceso de recuperación comienza cuando un nodo identifica que se encuentra en modo recuperación y solicita asistencia al nodo líder. Durante este período, consume mensajes desde RabbitMQ hasta recibir un ACK de recuperación, momento en el cual evalúa si debe iniciar una nueva elección. En caso contrario, continúa procesando los mensajes normalmente. Si el nodo no está en modo recuperación desde el inicio, directamente inicia el proceso electoral. Por su parte, el nodo líder permanece atento a los mensajes entrantes; si detecta una solicitud de recuperación, responde con sus mensajes EOS y un ACK de recuperación, mientras que los mensajes comunes son redistribuidos. Este flujo asegura una reintegración controlada de los nodos recuperados, manteniendo la coherencia del sistema distribuido.

6. Vista de Desarrollo

6.1. Diagrama de Paquetes

El sistema se encuentra organizado en múltiples paquetes distribuidos, cada uno con una responsabilidad específica, lo que permite una arquitectura desacoplada, escalable y fácil de mantener. A través del uso de contenedores y un broker de mensajería, los distintos componentes del sistema se comunican de forma asincrónica, garantizando una ejecución eficiente y paralela.

El paquete Client contiene los componentes necesarios para iniciar una consulta desde el lado del cliente. Incluye la lógica principal del cliente: En primer lugar, conectarse al gateway, recibir su número de cliente y enviar la cantidad de consultas a realizar; luego envía los datasets y espera los resultados de las consultas. Este paquete también incorpora un protocolo que encapsula la lógica de comunicación hacia el servidor y un componente llamado Results Receiver, encargado de escuchar los resultados que llegan desde el servidor.

El paquete Common agrupa las clases y funciones compartidas entre los distintos módulos del sistema. Aquí se destacan los componentes que permiten enviar y recibir mensajes de manera segura (Sender y Receiver), es decir, evitando la posibilidad de short reads o short writes; la lógica de protocolo compartida de mensajería (Protocol), donde se tienen los tamaños esperados para header y payload; y el decodificador de mensajes (Decoder). Este paquete asegura una base común y coherente en todos los puntos del sistema, promoviendo la reutilización y la consistencia.

El paquete Gateway cumple la función de punto de entrada principal al sistema. Aquí se establece la comunicación con cada cliente, se reciben los datasets, los cuales se interpretan mediante un componente de protocolo, y se reenvía a los distintos servicios responsables de procesar los datos. Una vez que las respuestas han sido calculadas, el componente Result Dispatcher del Gateway se encarga de enviarlas de vuelta al cliente, asegurando así una comunicación fluida de extremo a extremo.

Los paquetes agrupados bajo la categoría de Filter contienen los componentes encargados de aplicar los filtros necesarios sobre los datos originales. Entre estos se encuentran Clean Up, que realiza una limpieza inicial de los datos; Production, que filtra

las películas según el país productor; y Year, que permite seleccionar las películas por fecha de estreno. Estos módulos preparan los datos para su posterior análisis, reduciendo la cantidad de información procesada en las etapas siguientes.

El paquete Sentiment Analyzer incluye un worker especializado en el análisis de sentimientos. Este componente se encarga de procesar las sinopsis de las películas, y determinar el tono emocional de los mismos. A partir del resultado obtenido, se enviará a una cola u otra para su posterior análisis.

Por otro lado, los paquetes agrupados dentro de Query son los responsables de procesar las consultas específicas detalladas en el alcance. Cada worker, identificado como Q1 Controller hasta Q5 Controller, implementa la lógica de resolución correspondiente a una pregunta del enunciado. Estos componentes reciben los datos ya filtrados y procesados, y generan las respuestas finales que serán encoladas en la cola de resultados para luego ser enviadas al cliente.

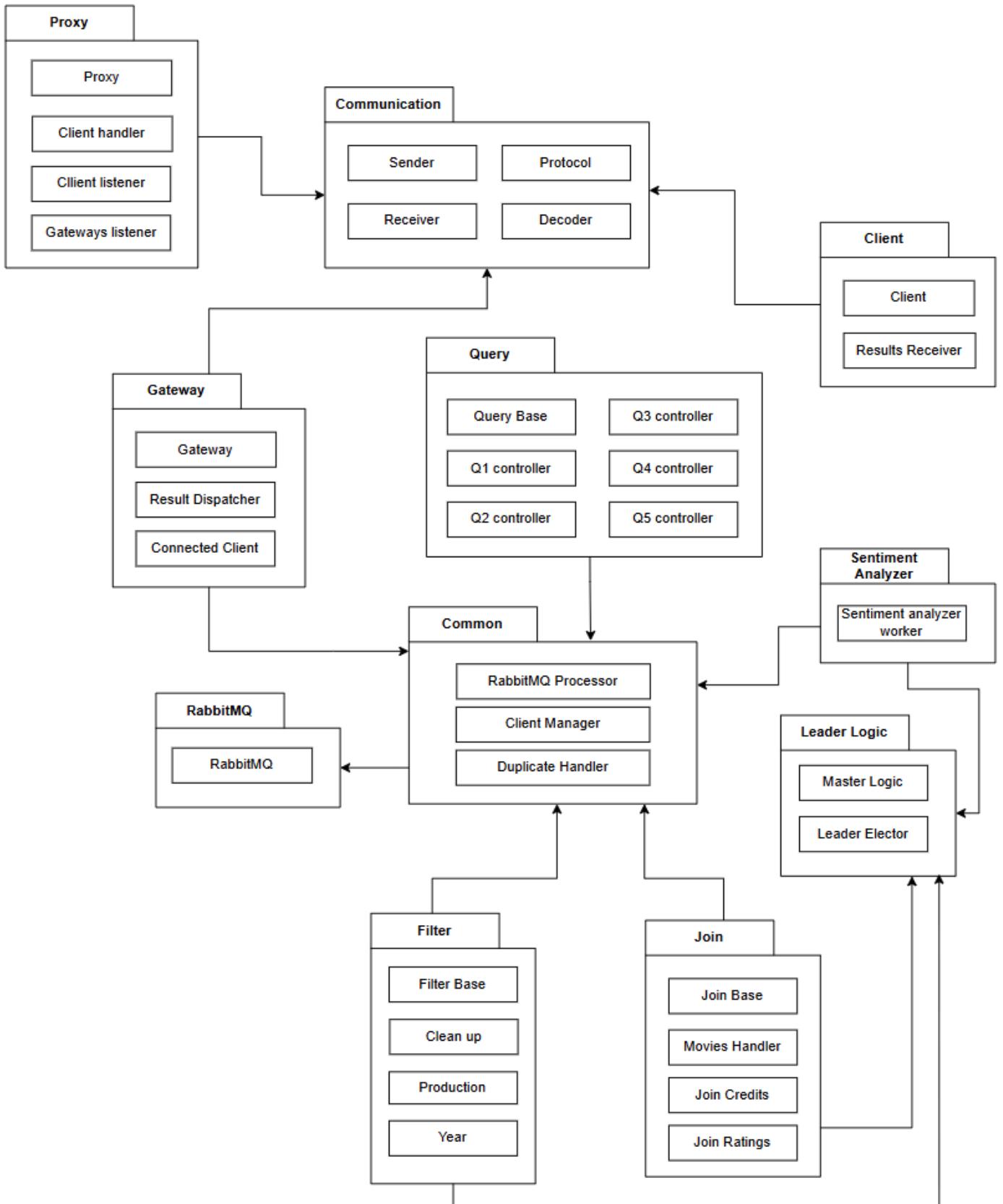
Dentro del paquete Join se encuentran los procesos especializados en realizar operaciones de join, específicamente Join Credits y Join Ratings. Ambos heredan de un módulo base llamado Join Base, que contiene la lógica general para llevar a cabo los joins. Además, estos nodos cuentan con un proceso adicional denominado Movies Handler, responsable de recibir la información de películas transmitida por los nodos Year Filter y de armar las tablas correspondientes para cada cliente. Una vez que disponen de alguna tabla de películas para un cliente, comienzan a procesar los batches de *credits* y *ratings*, respectivamente.

El paquete correspondiente a la lógica de líder contiene al módulo Leader Elector, principal encargado de la sincronización y toma de decisiones para la elección de líder entre distintos nodos de un mismo tipo, y también a Master Logic, utilizado para el comportamiento de “load balancer” del líder seleccionado.

El broker de mensajes, implementado con RabbitMQ, permite una arquitectura orientada a eventos. Funciona como middleware, gestionando la distribución de mensajes entre todos los servicios del sistema. Este enfoque no solo desacopla la comunicación entre módulos, sino que también facilita la escalabilidad horizontal y la tolerancia a fallos, al permitir múltiples instancias de un mismo servicio operando en paralelo.

En resumen, el diseño modular del sistema, basado en paquetes bien definidos y comunicados mediante eventos asincrónicos, permite una solución robusta, mantenible y

altamente escalable, capaz de adaptarse a distintos entornos de ejecución y demandas de procesamiento.

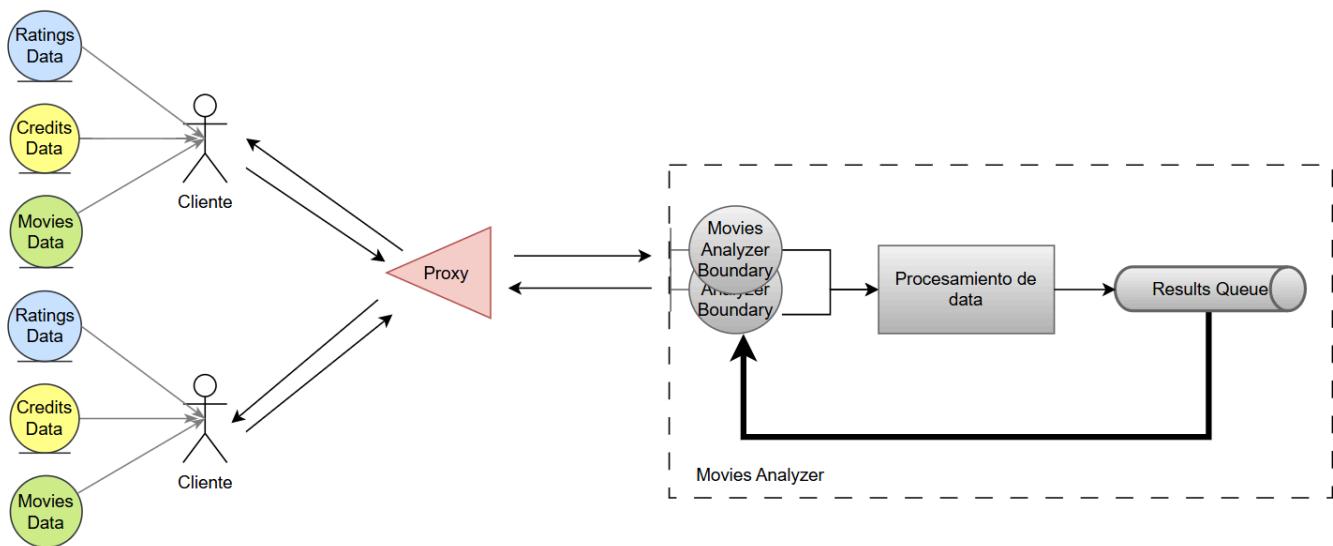


7. Vista Física

7.1. Diagrama de Robustez³

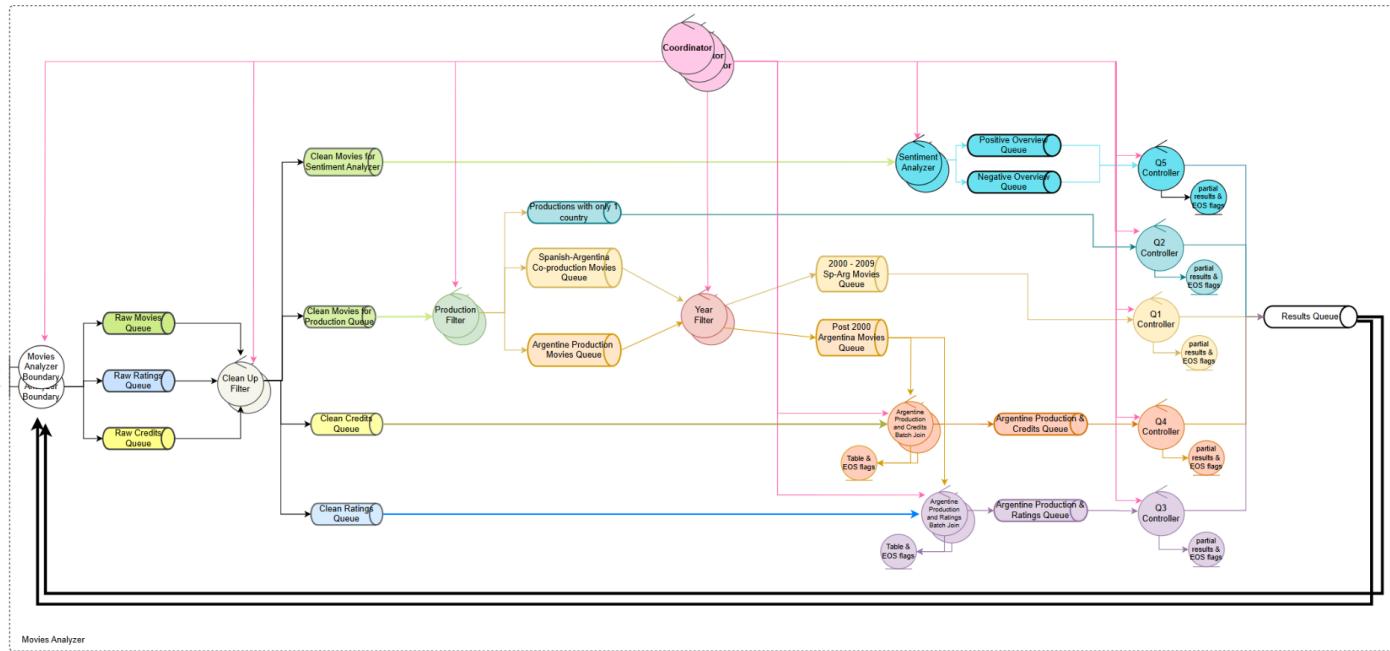
El diagrama de robustez expone la interacción entre las diferentes entidades involucradas en el sistema, mostrando cómo se conectan actores externos, límites del sistema, procesos y datos durante el flujo de ejecución general. En particular, permite visualizar cómo se transforman y enrutan los datos desde su carga inicial hasta la producción de las respuestas a las consultas específicas.

Como se explicará en profundidad en los puntos [11.1](#) y [11.3](#), tendremos 2 entidades principales: El sistema y el cliente. En esta sección nos enfocaremos en el sistema y todos sus componentes, pero una idea general de la interacción entre ellos se puede representar en el siguiente diagrama:



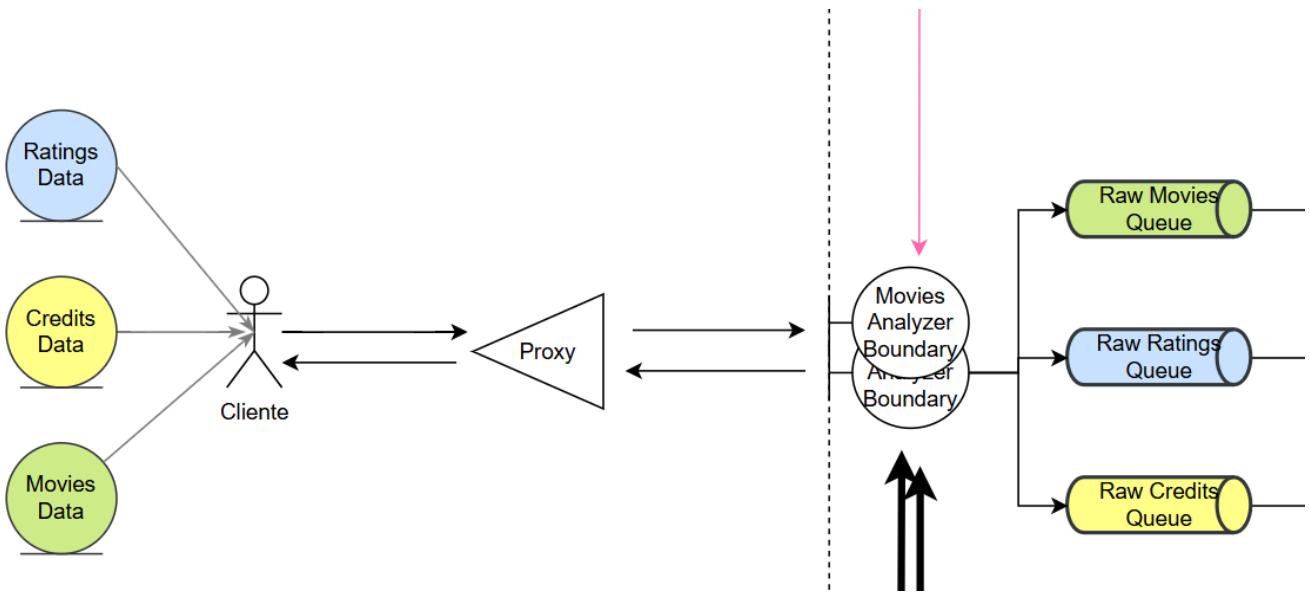
La primera acción que realiza el sistema es levantar todos los nodos que lo componen: *Proxy*, *Movies Analyzer Boundary*, *Clean Up Filter*, *Production Filter*, *Year Filter*, *Sentiment Analyzer*, *Argentine Production and Credits Batch Join*, *Argentine Production and Ratings Batch Join*, *Q1*, *Q2*, *Q3*, *Q4*, *Q5* y *coordinator*. Nótese que, como lo muestra el diagrama a continuación, algunos de ellos pueden ser más de 1 (como el *Movies Analyzer Boundary*, *Clean Up*, *Production*, *Year*, *Coordinator* y los joins).

³ En el anexo puede encontrarse el diagrama completo (sección [13.2.1](#))



Una vez levantado el sistema, el nodo **Coordinator** se encargará de monitorear el resto de los nodos que podrían sufrir caídas y restaurarlos en casos de las mismas. Este proceso autónomo estará corriendo siempre que se mantenga el sistema corriendo. Cabe destacar que se tienen replicas de Coordinator donde, una vez definido el líder, este se encargará de monitorear el sistema y reiniciar los nodos que detecte caídos. El resto de los nodos Coordinator que no sean líderes, simplemente estarán en modo “Stand by” esperando a alguna falla del líder para comenzar el proceso de liderazgo.

A medida que los actores externos —los clientes— se conecten al proxy, este les asignará un **Movies Analyzer Boundary/gateway** asociado al cual le forwardeará los paquetes. Estos paquetes serán batches provenientes de los tres conjuntos de datos base: **Movies Data**, **Credits Data** y **Ratings Data**.

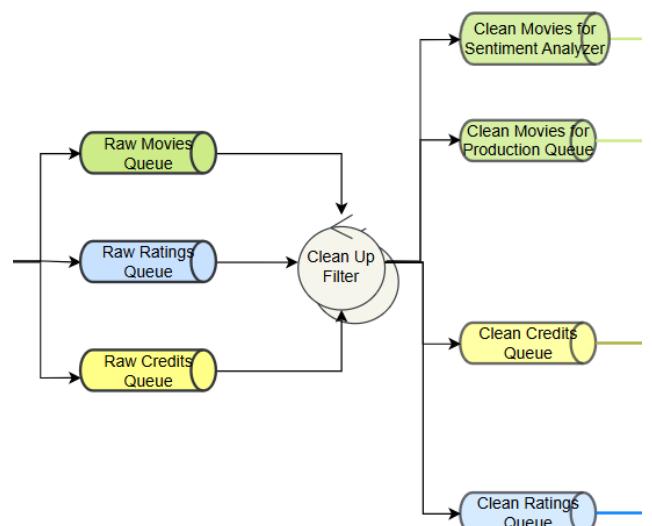


Una vez dentro del sistema, los datos son distribuidos en distintas colas de mensajes (una por cada tipo de dato) que funcionan como buffers entre etapas, permitiendo desacoplar los componentes e incrementar la paralelización. De esta manera, los batches de información correspondientes al dataset de `Movies_metadata` se irán encolando en la `Raw Movies Queue`, los correspondientes al dataset de `Credits` lo harán en la `Raw Credits Queue` y por último, los correspondientes al dataset de `Ratings` lo harán en la `Raw Ratings Queue`.

A partir de allí, de acuerdo al análisis que se realizó sobre los filtros necesarios para poder realizar las cinco consultas que se esperan poder resolver, se decidió tener los siguientes controladores:

- Para los procesos de filtrado:

- Clean Up Filter: Este filtro toma de las colas de Raw data y por cada dato que levanta, se fija que tenga toda la información esperada y solo propaga la que será relevante para futuras consultas⁴. Una vez realizado el filtro, encola en la queue correspondiente al tipo de dato que refiere (Movie, Credit o Ratings). Para el caso de las películas, debido a que

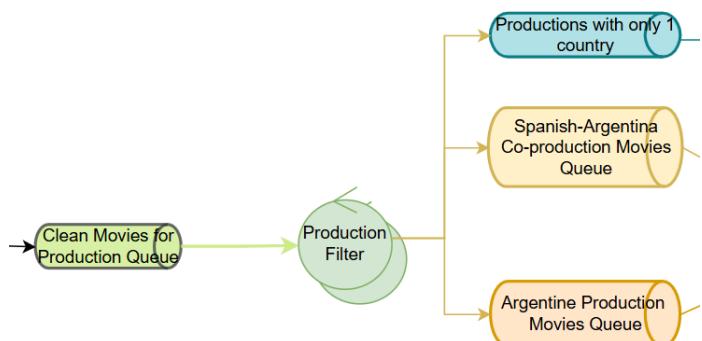


⁴ Para más información sobre las columnas que se utilizan a de DAG donde se especifica.

distintos procesos las necesitarán más adelante, se decidió por publicar los resultados a 2 queues: `Clean Movies for Sentiment Analyzer` y a `Clean Movies for Production`.

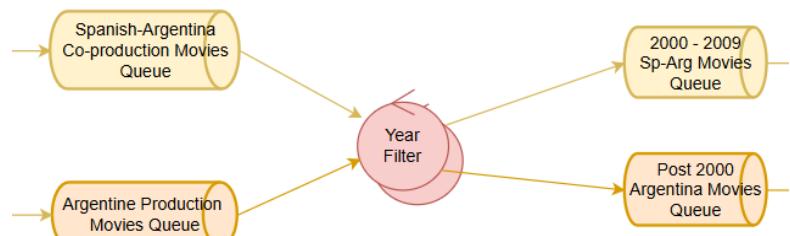
- **Production Filter:** Este filtro solo toma elementos de la cola `Clean Movies for Production Queue`, y analiza tanto la cantidad como el contenido de la columna `production_countries` de los tipo de datos `Movies`. Por un lado, se cuentan la cantidad de países que aparecen en estos diccionarios. Si solo aparece un país, entonces el elemento es encolado a la cola `Productions with only 1 country`. Por otro lado, se analizan cuáles países aparecen mencionados: En caso de tener como productor a Argentina, se lo encola a `Argentine Production Movies Queue`, y,

si además tiene como co-producción a España, se lo encola en `Spanish-Argentina Co-production Movies Queue`.



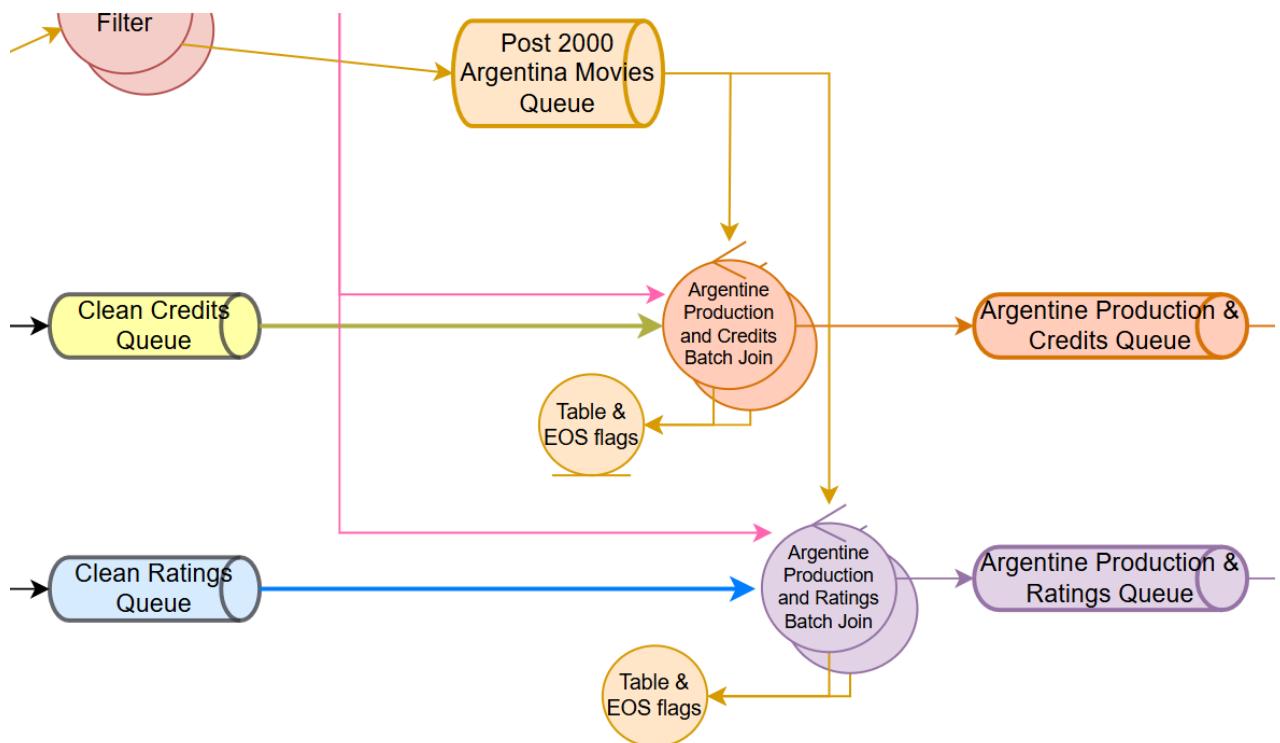
- **Year Filter:** Este filtro toma elementos tanto de la `Spanish-Argentina Co-production Movies Queue` como de la `Argentine Production Movies Queue`. Para el caso de la primera, se queda con los elementos cuya columna `release_date` informe una fecha dentro de los años 2000 y 2009 y los encola en la cola `2000 - 2009 Sp-Arg Movies Queue`. Para el caso de la segunda, simplemente se fija si la fecha de estreno es posterior al 2000 y la encola en `Post 2000 Argentina Movies Queue`.

`Sp-Arg Movies Queue`. Para el caso de la segunda, simplemente se fija si la fecha de estreno es posterior al 2000 y la encola en `Post 2000 Argentina Movies Queue`.



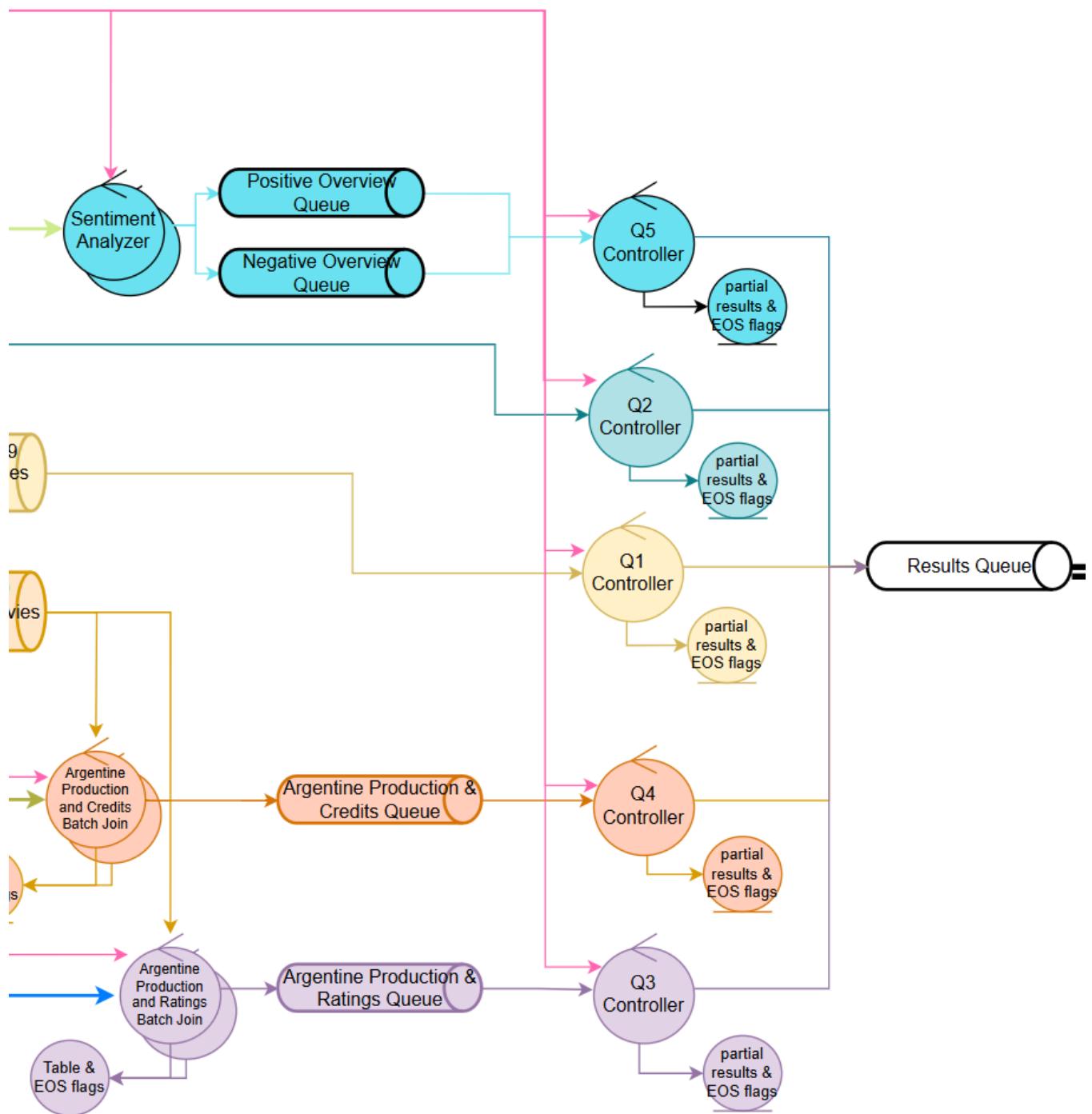
- Para las operaciones de join:

- **Argentine Production and Credits Batch Join:** Este proceso recibe, por un lado, los datos de créditos limpios desde la **Clean Credits Queue** en forma de batches, y por otro, accede a la tabla auxiliar generada previamente con un subprocesso **Movies Handler**, el cuál obtiene las películas a través de los mensajes producidos por el **Year Filter**. Por cada batch de créditos, se realiza una operación de join con las producciones argentinas. Los resultados del cruce son encolados en la **Argentine Production & Credits Queue** para su posterior uso. Este nodo es *stateful*, ya que debe guardar en disco la información de su sidetable y los EOS correspondientes a ella.
- **Argentine Production and Ratings Batch Join:** Este componente opera de manera análoga al anterior, pero trabajando sobre batches de ratings provenientes de la **Clean Ratings Queue**. Cada lote de datos de ratings se cruza con la sidetable también generada por su subprocesso **Movies Handler**. Los resultados se colocan en la **Argentine Production & Ratings Queue**. De la misma manera que los nodos de unión con créditos, estos también son *stateful*.



- Para el procesamiento de cada consulta:
 - **Q1 Controller:** Para obtener el resultado final para la consulta 1, el Q1 Controller tomará elementos de la cola **2000-2009 Sp-Arg Movies Queue** y de cada uno tomará el *original_title* y el *genre* para luego encolarlos en la cola de resultados final **Results Queue**. El nodo es *statefull*, guarda resultados parciales en disco para poder recuperarse en caso de caída.
 - **Q2 Controller:** Para obtener el resultado final para la consulta 2, el Q2 Controller tomará elementos de la cola **Productions with only 1 country** y de cada uno tomará el *budget* y el *producion_countries* para luego ir acumulando la cantidad de dinero invertido (*budget*) por cada país productor (*producion_countries*). Una vez que nos aseguremos que la cola ya no tiene más elementos, se toma el top 5 y se encolan los resultados en la **Results Queue**. El nodo es *statefull*, guarda resultados parciales en disco para poder recuperarse en caso de caída.
 - **Q3 Controller:** Para obtener el resultado final para la consulta 3, el Q3 Controller tomará elementos de la cola **Argentine Production & Ratings Queue** y de cada uno tomará el *original_title* y el *rating*. Una vez que estén todos los elementos se tomarán las películas con mayor y menor rating para luego encolarlos en la cola de resultados final **Results Queue**. El nodo es *statefull*, guarda resultados parciales en disco para poder recuperarse en caso de caída.
 - **Q4 Controller:** Para obtener el resultado final para la consulta 4, el Q4 Controller tomará elementos de la cola **Argentine Production & Credits Queue** y de cada uno tomará el *original_title* y el *cast*. Una vez que estén todos los elementos se agrupará por actor (*cast.id*) para saber en cuántas películas actuó cada uno. Una vez completada la agrupación, se tomarán los 10 actores con mayor aparición (*cast.name*) y se los encolará en la cola de resultados final **Results Queue**. El nodo es *statefull*, guarda resultados parciales en disco para poder recuperarse en caso de caída.
 - **Q5 Controller:** Para obtener el resultado final para la consulta 5, el Q5 Controller tomará elementos tanto de la cola **Positive Overview Queue**

como de la **Negative Overview Queue**. Por cada uno, calculará la tasa de ingreso/presupuesto (*revenue/budget*) y luego hará el promedio para todas aquellas películas con overview positivo vs. el negativo. Una vez obtenidos los promedios, se los encolará a la cola de resultados final **Results Queue**. El nodo es *statefull*, guarda resultados parciales en disco para poder recuperarse en caso de caída.



- Para el analizador de sentimientos:
 - **Sentiment Analyzer**: Para poder aplicar el modelo de procesamiento de lenguaje natural (PNL) a las *overview* de las películas este controlador recibirá lotes de películas de la cola **Clean Movies for Sentiment Analyzer Queue** y a cada uno le aplicará el modelo. Una vez completado el análisis, se obtendrá una valoración positiva o negativa. Para el caso de las valoraciones positivas, se encolarán las películas en **Positive Overview Queue**, mientras que las negativas se encolarán en **Negative Overview Queue**.

7.2. Diagrama de Despliegue⁵

El diagrama de despliegue ilustra la distribución física de los distintos componentes del sistema en nodos de ejecución, destacando cómo se asignan los procesos o workers a cada uno de ellos. En este sistema, se utiliza una arquitectura basada en microservicios con procesamiento distribuido, orquestado mediante un Message Oriented Middleware (RabbitMQ) que actúa como intermediario para desacoplar los distintos módulos.

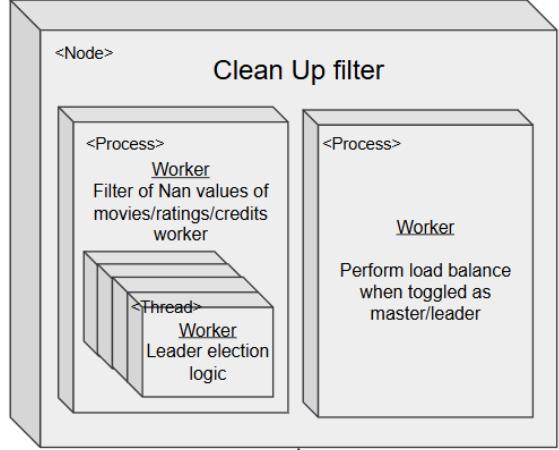
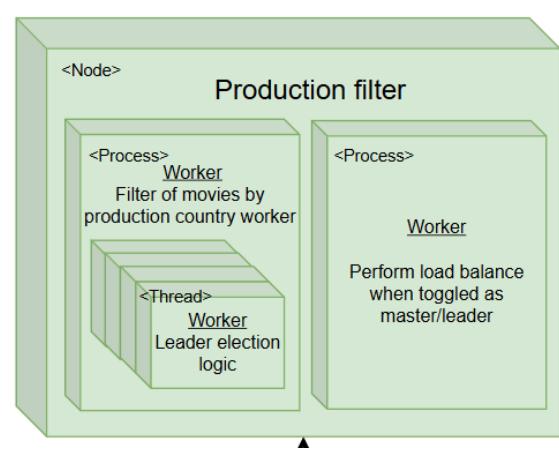
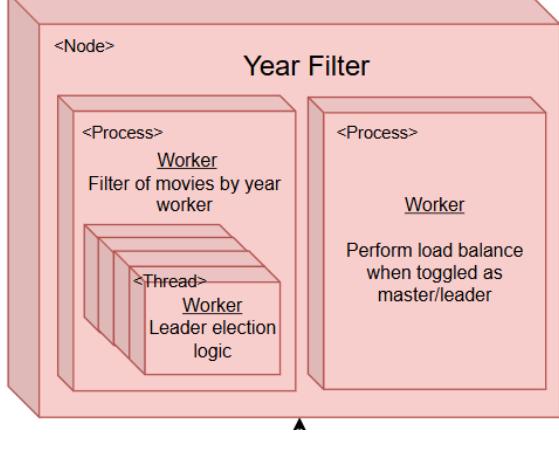
Cada nodo representa una unidad lógica de despliegue que puede correr en contenedores, máquinas virtuales o servidores físicos, dependiendo del entorno. Los procesos dentro de cada nodo representan workers especializados que consumen mensajes desde colas específicas, procesan los datos y producen nuevos mensajes para etapas posteriores del pipeline.

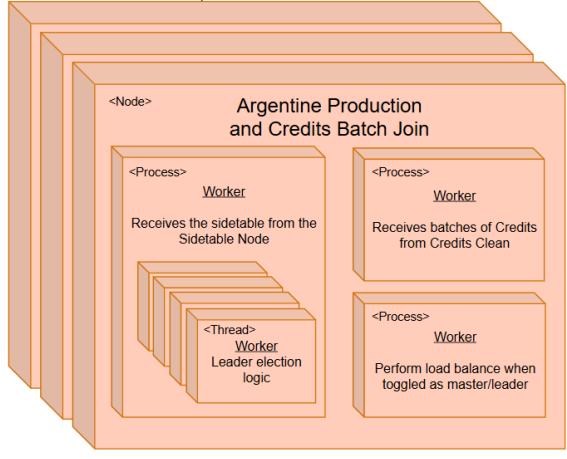
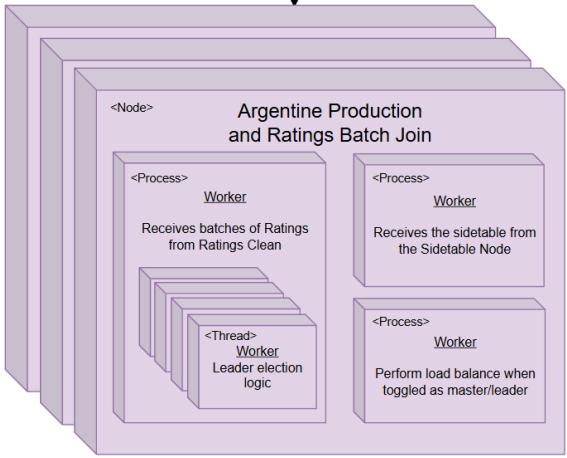
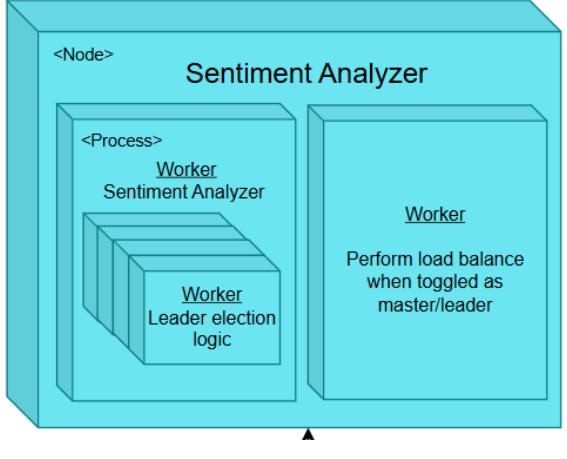
La arquitectura refleja la separación por responsabilidades, agrupando workers por dominio funcional: filtros, análisis de sentimiento, joins, y resolución de queries. Esta organización permite tanto paralelismo como escalabilidad, clave para soportar grandes volúmenes de datos y múltiples consultas simultáneas.

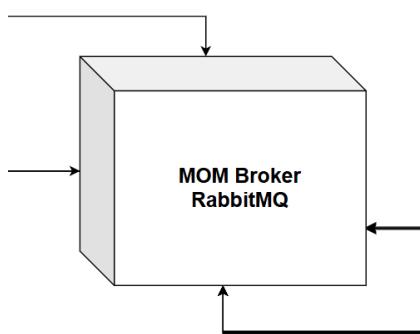
7.2.1. Descripción de Nodos

Diagrama	Detalle
	<p><u>Descripción:</u> Este nodo está compuesto por:</p> <ul style="list-style-type: none">• message_receiver : Recibe todos los mensajes enviados por el proxy• message_processor: Procesa cada mensaje• result_dispatcher: Consume de la cola results_queue los resultados ingresados por los nodos queries• result_sender: Envía al proxy las respuestas de todos los clientes conectados a ese gateway.• por cada cliente nuevo registrado se lanza un thread connected_client encargado de registrar cuántas respuestas asociadas al cliente fueron enviadas al Proxy. <p><u>Objetivo:</u> Actuar como intermediario entre los clientes y el sistema distribuido, asegurando una correcta recepción, distribución y reenvío de los mensajes.</p>

⁵ En el anexo puede encontrarse el diagrama completo (sección [13.2.3](#))

 <pre> <Node> Clean Up filter <Process> Worker Filter of Nan values of movies/ratings/credits worker <Thread> Worker Leader election logic <Process> Worker Perform load balance when toggled as master/leader </pre>	<p>Descripción: Este nodo está compuesto por uno o más workers que filtran los valores nulos (NaN) de los datasets de películas, ratings y créditos.</p> <p>Objetivo: Asegurar que los datos que continúan en el pipeline estén limpios y sean procesables, evitando errores en etapas posteriores (ej: no se puede calcular un promedio entre budget y presupuesto si alguno de los 2 valores es 0).</p>
 <pre> <Node> Production filter <Process> Worker Filter of movies by production country worker <Thread> Worker Leader election logic <Process> Worker Perform load balance when toggled as master/leader </pre>	<p>Descripción: Nodo responsable de filtrar las películas según el país de producción y la cantidad.</p> <p>Objetivo: Encolar según corresponda solo aquellas producciones cuyo país sea Argentina, Argentina y España o solo un único país (sin coproducción).</p>
 <pre> <Node> Year Filter <Process> Worker Filter of movies by year worker <Thread> Worker Leader election logic <Process> Worker Perform load balance when toggled as master/leader </pre>	<p>Descripción: Nodo que filtra las películas según el año de lanzamiento.</p> <p>Objetivo: Seleccionar únicamente aquellas películas cuya fecha de estreno sea a partir del año 2000 y/o pertenezcan a la década del 2000.</p>

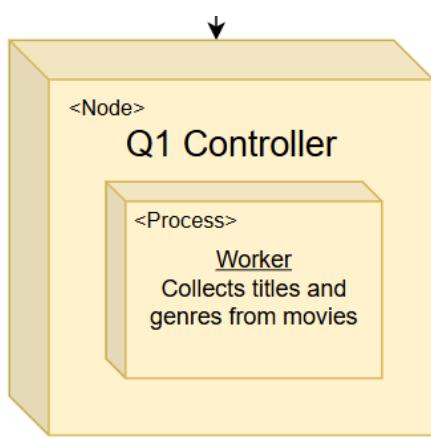
 <p>The diagram shows a process node titled "Argentine Production and Credits Batch Join". It contains three worker processes: one that receives a sidetable from the Sidetable Node, another that receives batches of Credits from Credits Clean, and a third that performs load balance when toggled as master/leader. A thread process labeled "Worker Leader election logic" is also shown.</p>	<p>Descripción: Realiza un join por ID entre la sidetable y batches de créditos.</p> <p>Objetivo: Enriquecer los datos de películas con información de los actores que participaron de dicha película.</p>
 <p>The diagram shows a process node titled "Argentine Production and Ratings Batch Join". It contains three worker processes: one that receives batches of Ratings from Ratings Clean, another that receives the sidetable from the Sidetable Node, and a third that performs load balance when toggled as master/leader. A thread process labeled "Worker Leader election logic" is also shown.</p>	<p>Descripción: Join por ID entre la sidetable y los ratings.</p> <p>Objetivo: Asociar los ratings con la película que evalúan.</p>
 <p>The diagram shows a process node titled "Sentiment Analyzer". It contains two worker processes: one for the Sentiment Analyzer and another for performing load balance when toggled as master/leader. A thread process labeled "Worker Leader election logic" is also shown.</p>	<p>Descripción: Aplicar el modelo NPL a el campo overview de cada película para analizar su sentimiento.</p> <p>Objetivo: Clasificar las sinopsis como positivas, negativas o neutras.</p>



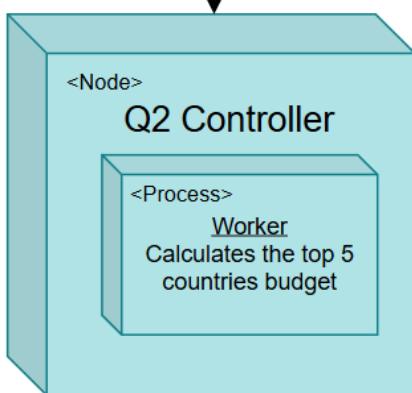
Descripción: Componente de intermediación basado en colas de mensajes.

Ventajas: Facilita desacoplamiento, paralelismo y tolerancia a fallos.

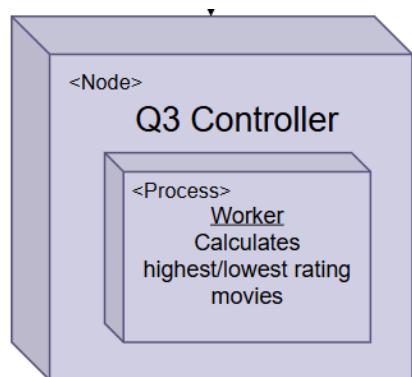
Objetivo: Orquestar el flujo de datos entre los nodos de procesamiento.



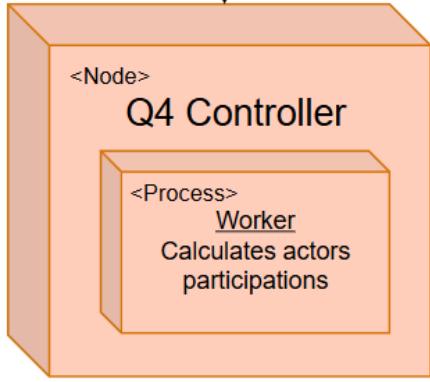
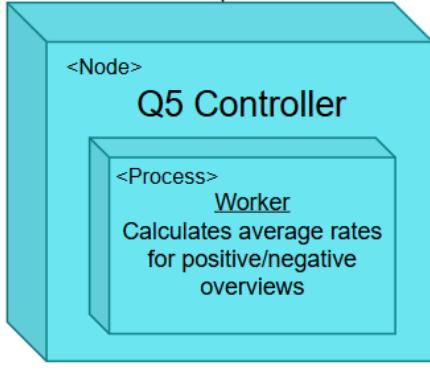
Descripción: Calcula los títulos y géneros de las películas argentinas post-2000.



Descripción: Determina los 5 países con mayor presupuesto acumulado.



Descripción: Calcula la película con mayor y menor rating.

	<p><u>Descripción:</u> Determina la cantidad de participaciones de actores por película.</p>
	<p><u>Descripción:</u> Calcula promedios de ratings diferenciados por polaridad del overview.</p>

7.2.2. Importancia del MOM Broker en la Arquitectura

El Message Oriented Middleware (MOM), en este caso **RabbitMQ**, es un componente central en la arquitectura del sistema. Todos los nodos y workers están conectados directa o indirectamente a través de este broker, lo que permite una comunicación desacoplada, asincrónica y escalable entre los distintos componentes del pipeline de procesamiento.

Ventajas de utilizar un MOM como RabbitMQ:

- **Desacoplamiento entre módulos:** Los productores de mensajes (por ejemplo, filtros o joins) no necesitan conocer la lógica ni la existencia de los consumidores (como los controladores de queries). Esto facilita la independencia entre equipos de desarrollo y la evolución del sistema por partes.
- **Escalabilidad horizontal:** Se pueden agregar más workers a una cola para procesar en paralelo, lo que permite manejar grandes volúmenes de datos sin modificar el código del sistema.

- **Tolerancia a fallos:** Si un worker falla o se reinicia, los mensajes permanecen en la cola hasta que puedan ser procesados nuevamente. Esto garantiza una mayor **resiliencia** frente a errores temporales.
- **Buffer natural ante picos de carga:** Si la velocidad de entrada de datos supera la de procesamiento, las colas actúan como un amortiguador, almacenando temporalmente los mensajes hasta que los workers los puedan procesar.
- **Flexibilidad en el diseño de flujos:** Permite construir pipelines complejos con múltiples pasos, reintentos automáticos, colas de prioridad, enrutamiento condicional, etc.

Desventajas o desafíos:

- **Mayor complejidad operativa:** Se requiere configurar, monitorear y mantener RabbitMQ, incluyendo aspectos como encolado, manejo de errores, confirmaciones y persistencia.
- **Latencia adicional:** El paso intermedio por la cola puede agregar algo de latencia en comparación con llamadas directas entre servicios, aunque se ve compensado por los beneficios en robustez.
- **Gestión de orden:** Si se requiere procesar mensajes en un orden estricto, se necesita un diseño cuidadoso ya que múltiples consumidores pueden parallelizar el procesamiento.
- **Dependencia central:** RabbitMQ se vuelve un punto crítico del sistema, por lo que debe estar correctamente replicado y monitoreado para evitar cuellos de botella o caídas.

8. Direct Acyclic Graph (DAG)

Aunque no forma parte estricta de las vistas tradicionales del modelo 4+1, la representación mediante un **DAG (Directed Acyclic Graph)** se incluye en este documento con el objetivo de **ilustrar el flujo de datos y dependencias entre etapas del procesamiento distribuido**.

El DAG presentado refleja cómo los distintos filtros y controladores del sistema interactúan entre sí, siguiendo una secuencia de transformación de datos desde su origen (los archivos de películas y ratings) hasta los distintos módulos de consulta que producen las respuestas finales.

Cada nodo del grafo representa un proceso lógico (filtro, unión o controlador de consulta), mientras que las aristas indican el flujo de datos, respetando la asincronía y la separación de responsabilidades que caracteriza al diseño general del sistema. El hecho de que sea un grafo acíclico garantiza que los datos fluyan en una única dirección, sin ciclos, lo que facilita el razonamiento sobre el orden de ejecución y evita problemas de dependencia circular.

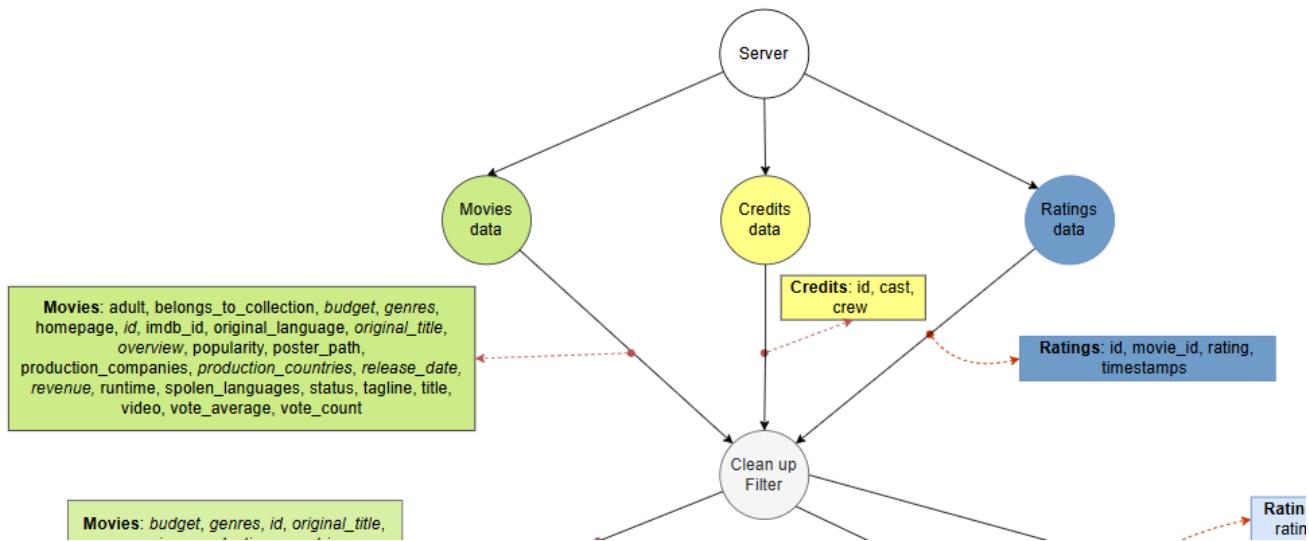
Además, el DAG permite identificar fácilmente puntos de paralelismo, cuellos de botella potenciales, y caminos críticos dentro del pipeline de procesamiento. Esta visualización resulta particularmente útil tanto para validar la arquitectura como para planificar el despliegue y la distribución de cargas en un entorno de ejecución real.

8.1. Descripción General⁶

El flujo comienza con la ingestión de tres datasets proporcionados por el servidor:

- **Movies data**
- **Credits data**
- **Ratings dat**

⁶ En el anexo puede encontrarse el diagrama completo (sección [13.2.2](#))



A continuación, los datos son procesados por un componente central denominado **Clean up Filter**, que se encarga de limpiar, homogeneizar y filtrar las entradas. Posteriormente, los datos limpios se derivan a distintos módulos de análisis, filtrado y combinación que permiten responder a cinco consultas principales.

8.2. Componentes del DAG

1. Ingesta de Datos

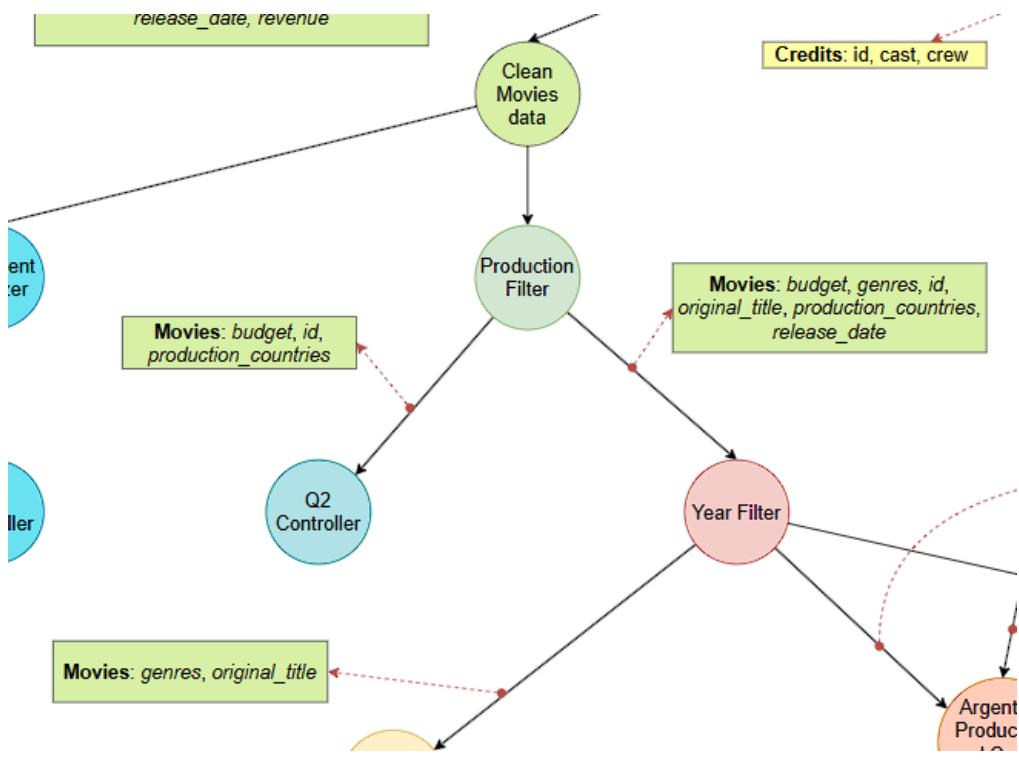
- **Server:** Fuente de los tres datasets iniciales.
- **Movies / Credits / Ratings data:** Datos originales, cada uno con campos relevantes como ***budget***, ***genres***, ***cast***, ***rating***, entre otros.

2. Limpieza y Normalización

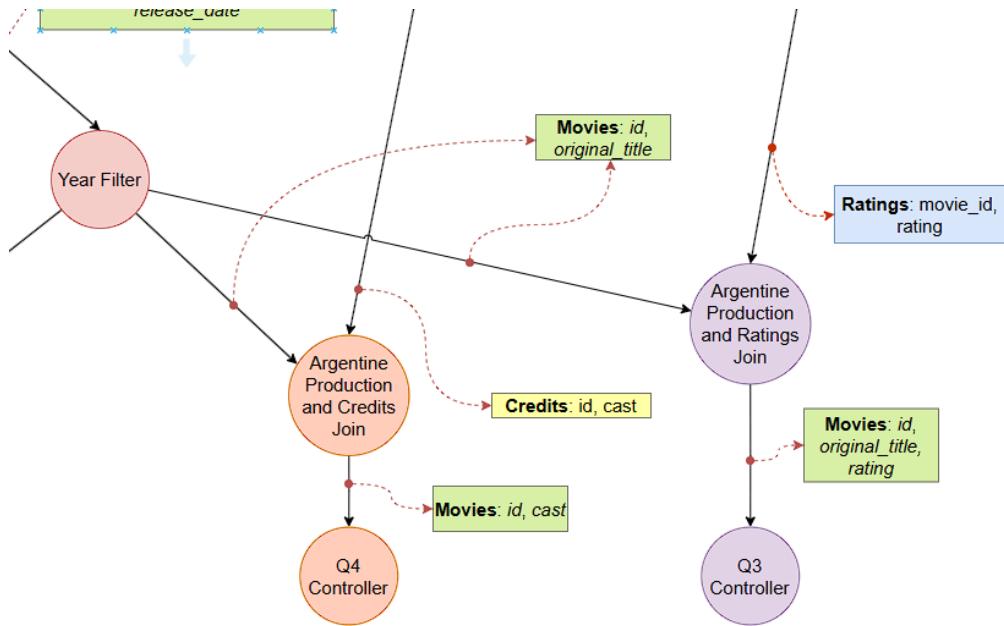
- **Clean up Filter:** Componente encargado de la limpieza de los datasets, eliminando valores nulos, corrigiendo formatos y asegurando la consistencia.
- **Clean Movies / Credits / Ratings data:** Versiones limpias de los datasets, listas para ser consumidas por las siguientes etapas.

3. Filtrado y Enriquecimiento

- **Production Filter:** Selecciona películas según su país de producción.
- **Year Filter:** Filtra películas por año.

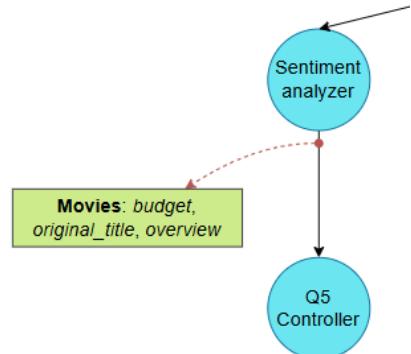


- **Argentine Production and Credits Join:** Junta las películas argentinas con los datos de cast y crew.
- **Argentine Production and Ratings Join:** Junta las películas argentinas con sus ratings.



4. Análisis

- **Sentiment Analyzer:** Analiza el sentimiento del campo `overview` de las películas, generando metadatos adicionales que enriquecen las respuestas.



5. Controladores de Consultas

Cada controlador responde a una consulta específica:

- **Q1 Controller:** Consulta películas por `genres` y `original_title`.
- **Q2 Controller:** Consulta películas según presupuesto y país de producción.
- **Q3 Controller:** Relaciona películas argentinas con sus ratings.
- **Q4 Controller:** Relaciona películas argentinas con el equipo técnico (cast).
- **Q5 Controller:** Presenta resultados enriquecidos con análisis de sentimiento.

9. Escalabilidad

9.1. Gestión de mensajes EOS

Importante: esta versión de manejo de EOS fue cambiada en la entrega de *Tolerancia a Fallos*, ver sección [11.5.2.2](#).

Para manejar correctamente los mensajes **EOS (End Of Stream)** en nuestro entorno escalable con múltiples nodos ejecutándose en paralelo, analizamos distintos patrones de comunicación entre nodos y desarrollamos lógicas específicas para cada caso. Dado que usamos RabbitMQ con un enfoque *productor-consumidor*, estos casos requieren un tratamiento cuidadoso para garantizar que el sistema detecte el final del flujo correctamente sin dejar nodos colgados ni perder datos.

A continuación una explicación de cada caso que debemos evaluar:

1 nodo → 1 nodo

En este escenario, el flujo de datos es lineal y directo. Solo hay un productor y un consumidor, por lo que el manejo del EOS es trivial: el nodo productor simplemente envía un único mensaje EOS, y el nodo consumidor, al recibirla, sabe que no recibirá más datos. No se necesita lógica adicional de control o coordinación.

Muchos nodos → 1 nodo

Aquí múltiples nodos están enviando mensajes a un único nodo de destino. Para detectar correctamente el fin del flujo, el nodo receptor mantiene un **diccionario o conjunto de nodos que debe esperar**. Cada vez que recibe un EOS, marca a ese emisor como "finalizado". Solo una vez que recibió el EOS de *todos* los emisores esperados, emite su propio EOS hacia el siguiente paso del DAG. Este enfoque asegura que el nodo no finalice prematuramente mientras aún puede haber mensajes en camino desde otros emisores.

1 nodo → muchos nodos

Este es uno de los casos más problemáticos en nuestro esquema con RabbitMQ, ya que usamos **distribución round robin** para balancear carga. Cuando el productor envía un único mensaje EOS, solo uno de los nodos consumidores lo recibe; los demás no lo ven y

por lo tanto *no terminan nunca*. Nuestra solución es que el nodo que consume el EOS:

1. Lo procese normalmente (cerrando su flujo de trabajo),
2. Pero **vuelva a colocar el mensaje EOS en la cola de entrada** antes de finalizar,
3. Y luego recién envíe su propio EOS al siguiente paso.

Esto permite que, con el tiempo, cada uno de los nodos consumidores eventualmente reciba el EOS. Es una forma simple y efectiva de simular un broadcast con las limitaciones del esquema round robin.

Muchos nodos → muchos nodos

Este último caso combina los desafíos de los dos anteriores. Además de tener que asegurarse de que **cada nodo reciba el EOS**, también tienen que saber **cuántos EOS esperar**. Aplicamos una lógica doble:

- Cada nodo mantiene un **diccionario de emisores esperados**, igual que en el caso "muchos → uno".
- Además, el mensaje EOS incluye un **contador**, que incrementa cada vez que pasa por un nodo que todavía necesitaba recibirla.

Si un nodo recibe un EOS y aún espera EOS de otros emisores, lo **vuelve a colocar en la cola**. El contador del mensaje ayuda a evitar que se encole indefinidamente. Cuando el contador alcanza el número esperado de nodos, se considera que el EOS ya "circuló" por todos los que debía, y no se reencola más.

Este esquema, aunque funcional, puede llegar a generar un **livelock**, donde varios nodos se pasan el EOS entre sí sin necesidad. Sin embargo, gracias al comportamiento probabilístico del round robin de RabbitMQ, usualmente todos los nodos terminan recibiendo el EOS. Como alternativa más robusta, podríamos implementar una **cola de broadcast específica**

para EOS, donde todos los nodos se suscriben y reciben directamente la señal de fin de flujo.

9.2. Escalado de nodos

El sistema fue diseñado para soportar un modelo de ejecución distribuida, con la posibilidad de escalar horizontalmente sus componentes mediante la duplicación de nodos. Esta capacidad está soportada por un archivo de configuración centralizado (`global.config`) que permite ajustar de forma flexible la cantidad de instancias que deben levantarse para cada uno de los workers involucrados en el procesamiento de datos.

Durante el desarrollo y evaluación del sistema, se realizaron pruebas sistemáticas con distintas configuraciones de cantidad de nodos para cada componente. En todos los casos se observó que, al incrementar la cantidad de nodos, el tiempo total de ejecución se reducía notablemente. Esto validó la efectividad del enfoque distribuido y la capacidad de escalar los workers para mejorar el rendimiento general del sistema. Sin embargo, también se identificó un límite práctico en este beneficio. A medida que se agregaban más nodos, especialmente en etapas donde se tenían múltiples nodos consumiendo de múltiples colas, comenzó a observarse un efecto negativo vinculado al mecanismo de detección de fin de batch (End Of Stream, EOS). Dado que cada nodo recibe batches de manera independiente, el sistema requiere múltiples mensajes de EOS para determinar cuándo una etapa ha finalizado completamente. Con una cantidad excesiva de nodos, este proceso de sincronización se volvió costoso en tiempo, generando demoras en la detección del fin de los batches y, por ende, en la continuación del flujo de procesamiento. Por esta razón, se buscó un punto de equilibrio entre paralelismo y eficiencia en la configuración final.

Uno de los mayores desafíos de rendimiento se presentó en el componente de análisis de sentimientos (Sentiment Analyzer). Este módulo utiliza un modelo de procesamiento de lenguaje natural (NLP) que resulta particularmente costoso en términos computacionales. Ejecutar este modelo de forma secuencial para cada película implicaba tiempos de espera prolongados, que se acumulaban significativamente al procesar batches completos. Para solucionar este cuello de botella, se implementó un **pool de workers internos** mediante un `ThreadPoolExecutor`, permitiendo procesar múltiples películas de un mismo batch en paralelo dentro de un único nodo. A lo largo de este proceso, se realizaron pruebas comparando el rendimiento entre el uso de **threads** y **procesos** para la ejecución paralela. Los resultados mostraron que **threads** ofrecieron una mejora significativa en el tiempo de ejecución.

9.2.1. Pruebas con Threads vs. Procesos

Al realizar las pruebas, se observó que el uso de **ThreadPoolExecutor** resultó ser más eficiente que el uso de un **ProcessPoolExecutor**. Esto se debe a varias razones clave:

1. **Menor sobrecarga de memoria:** Los threads comparten el mismo espacio de memoria, lo que significa que **no es necesario duplicar la carga del modelo NLP** en cada proceso, como ocurre con los procesos, que requieren cargar una instancia separada del modelo por cada nodo. Esto redujo significativamente el consumo de memoria y la latencia asociada con la inicialización de procesos.
2. **Menor overhead en la creación y comunicación de threads:** Crear un thread es significativamente más liviano que crear un proceso, especialmente cuando se usan pools de threads, lo que permitió un inicio más rápido y menor costo en términos de gestión del sistema operativo.
3. **Modelo NLP optimizado para multihilos:** Muchas de las bibliotecas de procesamiento de lenguaje natural (como las de HuggingFace y **tokenizers**) están optimizadas para operar eficientemente con múltiples threads, ya que internamente manejan paralelismo para operaciones como la tokenización y la inferencia en GPU. Al usar **múltiples threads dentro de un solo proceso**, se aprovechó el paralelismo natural de estas bibliotecas sin generar conflictos o overhead adicional.

Gracias a estas optimizaciones, el tiempo promedio de procesamiento de una consulta completa se redujo de aproximadamente **45 minutos** a tan solo **25 minutos** al usar threads. Esta mejora no solo hizo el sistema más ágil, sino que también permitió liberar recursos de cómputo más rápidamente, mejorando la eficiencia global.

Finalmente, también se realizaron pruebas incrementando la cantidad de nodos del Sentiment Analyzer. Sin embargo, se observó que, más allá de cinco nodos, los beneficios adicionales eran marginales. La combinación de un número moderado de nodos y el uso de ejecución en paralelo dentro de cada uno resultó ser la estrategia más efectiva para balancear tiempo de respuesta y utilización de recursos.

9.2.1.1. Actualización por tolerancia a fallos

Durante las pruebas de desempeño y robustez del sistema, nuestro corrector nos recomendó reemplazar el modelo de NLP utilizado inicialmente por una alternativa más liviana: TextBlob. Esta sugerencia estuvo motivada por la necesidad de mejorar la velocidad de procesamiento y reducir la complejidad general del sistema.

Al implementar TextBlob, se observó una reducción drástica en los tiempos de ejecución, pasando de aproximadamente 20 minutos a tan solo 2 minutos para procesar el mismo conjunto de datos. Esta mejora se debió principalmente a la simplicidad del modelo y a su enfoque basado en reglas, en contraste con modelos más pesados y dependientes de infraestructura GPU o multihilo.

Sin embargo, durante la fase de integración con la arquitectura multithread del Sentiment Analyzer, se descubrió que TextBlob no es thread-safe. Esto generaba comportamientos inesperados o errores cuando se intentaba procesar múltiples consultas en paralelo utilizando ThreadPoolExecutor. En la siguiente imagen, puede notarse como la misma película “*Guardian Angel*” obtiene una polaridad **Negativa** en un hilo, y una polaridad **Neutral** en otra:

```
[SentimentAnalyzer] Guardian Angel movie overview detected, polarity: -0.145
[SentimentAnalyzer] [Worker] Guardian Angel movie sentiment: negative
[SentimentAnalyzer] [Worker] Guardian Angel overview: Detective – turned – bodyguard
s hired by a psychotic icy seductress, Nina Lindell (Lydie Denier), the same woman who
er. With vendetta in her heart McKay accompanies the flamboyantly playful womanizer t
super-rich as his protector. In a unique role reversal, it is the woman protecting t
g into a deadly triangle of passion, suspense and action.
[SentimentAnalyzer] Guardian Angel movie overview detected, polarity: -0.066
[SentimentAnalyzer] [Worker] Guardian Angel movie sentiment: neutral
[SentimentAnalyzer] [Worker] Guardian Angel overview: Detective – turned – bodyguard
s hired by a psychotic icy seductress, Nina Lindell (Lydie Denier), the same woman who
er. With vendetta in her heart McKay accompanies the flamboyantly playful womanizer t
super-rich as his protector. In a unique role reversal, it is the woman protecting t
g into a deadly triangle of passion, suspense and action.
```

Como solución, se optó por ejecutar el modelo únicamente en el hilo principal, deshabilitando la parallelización interna.

A pesar de esta limitación, el rendimiento general siguió siendo sobresaliente: incluso sin parallelización, el tiempo total de procesamiento no superó los 5 minutos para el set completo de datos. Esta solución demostró ser altamente efectiva, permitiendo un sistema más simple, confiable y aún eficiente, cumpliendo con los requisitos de tolerancia a fallos y manteniendo un rendimiento aceptable.

10. Multi-Client

10.1. Gestión de proceso de mensajes para múltiples clientes

En nuestro sistema la coordinación de múltiples flujos de datos y nodos de procesamiento requiere una gestión del estado de cada cliente (identificado por el id del cliente y id de solicitud). Para esto, se implementaron dos componentes clave: ClientManager y ClientState.

10.1.1. ClientManager

El módulo ClientManager actúa como un registro centralizado de los clientes activos en el sistema. Su responsabilidad principal es mantener una estructura de datos que almacena, por cada cliente, un objeto ClientState. Esto permite rastrear el progreso de los mensajes y del estado de finalización (EOS) por cliente.

Funciones principales:

- `add_client(client_id, request_id):`
 - Si el cliente no existe en el registro, se crea un nuevo ClientState.
 - Devuelve la instancia ClientState correspondiente.
- `remove_client(client_id, request_id):`
 - Elimina del registro al cliente identificado, liberando recursos una vez que el procesamiento ha finalizado.

10.1.2. ClientState

El módulo ClientState representa el estado individual de un cliente durante una solicitud de procesamiento. Su foco principal es el seguimiento del estado de finalización (EOS) recibido por cola y por nodo.

Atributos clave:

- `client_id` y `request_id`: identificadores únicos del cliente y de la solicitud.
- `eos_flags`: diccionario que almacena por cada cola de origen los nodos que ya han enviado el mensaje de EOS.

- `amount_of_eos`: cantidad esperada de nodos del mismo tipo que deben enviar EOS antes de considerar finalizada la cola.

Métodos principales:

- `mark_eos(queue_name, node_id)` :
 - Marca que un nodo específico (`node_id`) ha enviado EOS para una cola dada.
- `has_queue_received_eos(queue_name)` :
 - Verifica si se han recibido todos los EOS esperados para una cola específica.
- `has_received_all_eos(expected_queues)` :
 - Determina si se han recibido los EOS de todas las colas de origen definidas, es decir, si el flujo completo ha concluido para ese cliente.

10.1.2.1. Actualización por tolerancia a Fallos

Como parte de la entrega de tolerancia a fallos, se refactorizó la clase `ClientManager` para incorporar directamente la funcionalidad que antes residía en `ClientState`, la cual quedó deprecada. Ahora, `ClientManager` mantiene un **diccionario de eos_flags por cliente**, administrado por un `multiprocessing.Manager` para permitir el acceso compartido entre procesos. Esta modificación fue necesaria para que los nodos líderes, que ejecutan el proceso `MasterLogic`, puedan consultar estos flags y reenviar los mensajes EOS a los nodos que se encuentren en proceso de recuperación (ver sección [11.5.2.2](#)).

10.1.3. Manejo de Side Tables para clientes

Como se mencionó anteriormente, los nodos *Join* ahora incorporan un proceso denominado *Movies Handler*, encargado de llevar un registro de las películas recibidas para cada cliente. Una vez que se recibe al menos una tabla de películas para un cliente, se habilita el inicio de las operaciones de *join*. En caso de que llegue un *batch* de *credits* o *ratings* correspondiente a un cliente cuya tabla aún no ha sido recibida, el mensaje se reencola en la *queue* de *batches*. De esta forma, solo se procesan los *batches* para los cuales el nodo *Join* ya dispone de su respectiva tabla de películas.

10.2. Adaptación del Gateway

10.2.1. Separación de responsabilidades

En la versión inicial, el Gateway tenía una arquitectura monolítica: se encargaba simultáneamente de manejar conexiones de red, mantener el estado del cliente conectado, interpretar y procesar sus mensajes, y ejecutar la lógica de negocio asociada. Este diseño, aunque funcional para un entorno de baja complejidad, presentaba múltiples inconvenientes en contextos multicliente:

- Acoplamiento elevado, dificultando el mantenimiento y la claridad del código.
- Baja escalabilidad, ya que una única clase era responsable de múltiples tareas críticas.
- Mayor riesgo de errores en condiciones de alta concurrencia por la falta de encapsulamiento y control del estado compartido.

Por lo tanto, el Gateway fue refactorizado siguiendo principios de diseño orientados a objetos y buenas prácticas de ingeniería de software como Single Responsibility Principle (SRP) y encapsulamiento. Se introdujeron los siguientes componentes:

- **ClientRegistry**: es una clase dedicada exclusivamente a la gestión de los clientes conectados. Su responsabilidad incluye registrar nuevas conexiones, eliminarlas al desconectarse y mantener un mapeo actualizado entre IDs de cliente y sus respectivas instancias. Esto permite centralizar el control de clientes de forma segura y coherente.
- **ConnectedClient**: encapsula la lógica asociada a una única conexión con un cliente. Aísla detalles como el socket de comunicación, la gestión de mensajes entrantes y salientes y el cierre ordenado de la conexión. Esto facilita el manejo independiente de cada cliente y la extensión del comportamiento individual en el futuro.
- **ResultDispatcher**: mantiene su rol de responsable del envío de resultados procesados. Su desacoplamiento del resto de la lógica permite enfocar su funcionamiento en el formato, enrutamiento y entrega de respuestas, sin depender

del estado interno del Gateway.

- **RabbitMQProcessor:** se extrae toda la lógica relacionada con la mensajería de RabbitMQ desde el Gateway, aislando la creación de conexiones, manejo de colas y consumo de mensajes. Esta separación reduce el acoplamiento con la infraestructura de mensajería y permite una mejor reutilización o sustitución de esta capa en el futuro.

Esta nueva estructura modular mejora significativamente la mantenibilidad del código, la escalabilidad para múltiples clientes concurrentes, y la capacidad de pruebas unitarias de cada componente de forma aislada.

10.2.2. Manejo explícito de nodos del sistema

En un sistema distribuido, es fundamental garantizar que todos los nodos del sistema estén listos antes de iniciar el procesamiento coordinado de tareas. Para esto, se implementó un mecanismo explícito de detección y sincronización entre nodos:

- Se introduce la variable de entorno `SYSTEM_NODES`, que especifica cuántos nodos se espera que estén activos antes de que el Gateway inicie su ciclo de vida normal. Esta variable actúa como un parámetro configurable que permite adaptar el sistema a diferentes despliegues.
- Se crea una cola específica en RabbitMQ denominada `nodes_ready_queue`, donde cada nodo, al iniciar correctamente (es decir, una vez que se confirma la conexión con RabbitMQ), publica un mensaje `<node_name>_ready` indicando que está listo. El Gateway se suscribe a esta cola y mantiene un contador de nodos listos.

Este mecanismo mejora notablemente la sincronización distribuida, evitando situaciones de carrera o inconsistencias donde algunos nodos comienzan a operar antes de que el sistema esté en condiciones estables. Se garantiza así un inicio coordinado del sistema distribuido.

10.2.3. Mejor manejo del estado

La adaptación del Gateway también implicó mejoras importantes en el control del estado interno, particularmente en un entorno multicliente y distribuido, donde es esencial evitar condiciones de carrera y garantizar consistencia.

Se introduce la variable interna `_ready_nodes`, junto con un **lock explícito**, para llevar un conteo seguro y atómico de los nodos que se van declarando listos. Esto asegura que el chequeo de si el sistema puede empezar no se vea afectado por concurrencia o errores de sincronización.

Se mantiene la variable `_was_closed`, que indica si el Gateway fue cerrado o no, permitiendo controlar de forma ordenada el ciclo de vida del servidor. Esto evita que se sigan aceptando conexiones o procesando mensajes en estados inconsistentes o después de una interrupción.

Gracias a estas mejoras, el Gateway no solo se adapta a múltiples clientes, sino que también adquiere un comportamiento más robusto, seguro y coherente dentro de una arquitectura distribuida y concurrente.

10.2.3.1. Actualización por tolerancia a Fallos

Para la última versión se quitó esta variable interna, ya que se pasó a tener 2 scripts separados de docker-compose donde se levanta primero el sistema y una vez que está estabilizado, recién allí se levanta el de los clientes. Para más información al respecto vaya al punto [11.1](#))

11. Tolerancia a fallos

11.0 Recepción de películas mediante sharding en los nodos Join

El sistema ha implementado un mecanismo de distribución basado en sharding para gestionar el flujo de películas (movies) hacia los nodos de join. Esta arquitectura permite una distribución eficiente y balanceada de la carga de trabajo, donde cada nodo join procesa únicamente las películas que corresponden a su rango de shard asignado.

11.0.1 Configuración de Sharding

La configuración de sharding se define completamente en el archivo docker-compose.yml mediante variables de entorno. El sistema utiliza tres variables principales:

- **SHARD_MAPPING**: Define el mapeo completo de todos los nodos y sus rangos
- **SHARD_RANGE_START**: ID mínimo que procesa el nodo específico
- **SHARD_RANGE_END**: ID máximo que procesa el nodo específico

11.0.2. Proceso de recepción de películas

Una vez comenzado el proceso de envío de películas, el Movies Handler es el encargado de recibirlas.

11.0.2.1 Filtrado por Rango

Solo se retienen aquellas películas cuyo id se encuentra dentro del rango de shard asignado al nodo. Este paso asegura que ningún nodo procese información fuera de su dominio.

11.0.2.2 Almacenamiento en Memoria

Las películas válidas son almacenadas en una estructura compartida self.movies[client_id], la cual está gestionada por un multiprocessing.Manager. Esta estructura mantiene las películas agrupadas por cliente (client_id).

11.0.2.3 Persistencia a Disco

Posteriormente, las películas almacenadas se persisten en disco mediante el método `write_storage`. Esto garantiza que, ante una caída del sistema, se puede restaurar el estado previamente alcanzado sin pérdida de información.

11.0.2.4 Manejo de Mensajes EOS

Cuando un nodo recibe un mensaje de tipo EOS (End Of Stream), se actualizan los indicadores internos para señalar que otro nodo del sistema ha finalizado el envío de películas para un cliente determinado. Estos indicadores también se escriben a disco.

11.0.3 Coordinación desde el Nodo Líder

El reparto inicial de películas hacia los nodos joins lo realiza el nodo líder a través del componente `MasterLogic`. Este componente evalúa, para cada película, a qué nodo le corresponde de acuerdo al `SHARD_MAPPING` y publica el mensaje en la cola correspondiente. Esta distribución se realiza de forma automática cuando se encuentra activado el modo `sharded`. Se detallará este componente más adelante.

11.1. Separación de contenedores: Sistema y Clientes

Como parte de la estrategia de **tolerancia a fallos** del sistema distribuido, se diseñó una separación explícita entre los contenedores que componen el sistema core (nodos de procesamiento, coordinador, proxy, gateways, etc.) y los contenedores que representan a los **clientes**. Esta separación se implementa generando dos archivos `docker-compose` distintos:

- `docker-compose.system.yml`: incluye todos los servicios que forman el sistema distribuido (nodos de filtrado, join, análisis de sentimientos, consultas, gateway, proxy, coordinador y broker de mensajes RabbitMQ).
- `docker-compose.clients.yml`: define exclusivamente los contenedores de los clientes que generan cargas de trabajo y envían datasets al sistema y esperan luego por sus correspondientes respuestas.

Esta división tiene múltiples beneficios en términos de **tolerancia a fallos**:

- **Aislamiento de fallos del cliente:** Si uno o más clientes fallan, se puede reiniciar su entorno sin afectar el sistema en producción ni interrumpir la ejecución de los nodos que procesan los datos.
- **Reinicio independiente:** Los clientes pueden ser lanzados o reiniciados de forma independiente del sistema. Esto resulta útil durante pruebas de carga, validaciones o reinicios por fallos.
- **Escalabilidad controlada:** Permite instanciar múltiples clientes sin necesidad de modificar la configuración del sistema principal, facilitando pruebas de stress o escenarios con distintos niveles de concurrencia.
- **Entorno limpio por generación:** Antes de cada generación de los docker-compose, se eliminan los resultados previos y el contenido del storage, reduciendo problemas derivados de estados inconsistentes o resultados residuales.

Esta arquitectura modular, combinada con una red compartida (testing_net), asegura que los contenedores puedan comunicarse adecuadamente, manteniendo a la vez un **desacople lógico y operativo** entre componentes críticos del sistema y actores externos (clientes).

11.2. Introducción de nodo Coordinator

Con el objetivo de garantizar la disponibilidad y continuidad del sistema ante posibles caídas de procesos, se introdujo un componente central denominado **Coordinator**. Este nodo actúa como un monitor y controlador de los distintos contenedores Docker que ejecutan los nodos del sistema distribuido, tales como gateways, filters, queries y joins.

El Coordinator está implementado como un proceso autónomo que ejecuta un ciclo de monitoreo continuo sobre una lista configurable de contenedores Docker. Su función principal es verificar periódicamente el estado de ejecución de cada contenedor especificado, detectando cuándo alguno de ellos se encuentra detenido o no responde correctamente. En caso de detectar un contenedor caído, el Coordinator procede a

detenerlo formalmente (en caso de que quede en estado inconsistente) y luego lo reinicia para restaurar su operatividad.

Esta funcionalidad de reinicio automático contribuye significativamente a la tolerancia a fallos del sistema, ya que permite la recuperación rápida y automática de nodos caídos sin intervención manual. El Coordinator maneja además distintos tiempos de espera configurados según el tipo de nodo, permitiendo por ejemplo diferenciar entre gateways y otros nodos, ajustando los retardos antes del reinicio para facilitar el proceso de recuperación.

A diferencia de herramientas externas o sistemas de orquestación completos, esta solución es ligera, específica y orientada a las necesidades particulares del entorno implementado, formando parte de una arquitectura de monitoreo y recuperación diseñada para asegurar la resiliencia del sistema distribuido.

Además, el Coordinator cuenta con un esquema de réplicas para reforzar la alta disponibilidad del sistema. A través de un mecanismo de elección de líder (detallado en el punto [11.5.1](#)), uno de los Coordinators activos es designado como líder, y es el único encargado de monitorear todos los nodos del sistema y sus hermanos Coordinators. Los Coordinators que no resultan elegidos como líder permanecen en modo *stand-by*, sin ejecutar acciones de monitoreo ni reinicio, salvo por el envío periódico de mensajes de tipo "ALIVE" al líder para indicar que siguen activos. Esta estrategia permite evitar redundancia de acciones, reducir la carga de supervisión distribuida y mantener la coherencia en la gestión del sistema.

Por último, la configuración del Coordinator se realiza a través de variables de entorno que especifican los nombres de los contenedores a monitorear, facilitando la adaptabilidad y extensibilidad del sistema. Esta aproximación permite escalar o modificar el conjunto de nodos supervisados sin cambios en el código, manteniendo un control centralizado y eficiente sobre la infraestructura desplegada.

11.3. Introducción de nodo Proxy

Con el objetivo de desacoplar responsabilidades, escalar horizontalmente y mejorar la resiliencia del sistema, se desarrolló un **Proxy** multigateway, encargado de recibir conexiones tanto de gateways como de clientes, y enrutar mensajes entre ellos de forma segura y eficiente.

En versiones anteriores, todos los clientes se conectaban directamente a un único Gateway. Este enfoque, aunque funcional en entornos controlados o de baja concurrencia, presentaba importantes limitaciones:

- **Punto único de falla:** si el gateway fallaba, se perdía toda la capacidad de interacción cliente-sistema.
- **Escalabilidad limitada:** la sobrecarga de múltiples clientes podía saturar el gateway.
- **Dificultad para balancear carga o reenrutar en caso de fallos.**
- **Falta de flexibilidad para gestionar reconexiones o transiciones suaves ante reinicios del sistema.**

Para resolver estos problemas, se incorporó un nuevo **contenedor Proxy** con las siguientes responsabilidades centrales:

- **Gestión de múltiples clientes:** escucha conexiones de clientes en un puerto dedicado y las enruta dinámicamente hacia gateways disponibles.
- **Gestión de múltiples gateways:** mantiene conexiones activas con múltiples gateways, permitiendo un balanceo de carga más eficiente y mayor tolerancia ante fallos individuales.
- **Aislamiento de roles:** desacopla completamente la lógica de comunicación cliente-gateway, liberando al gateway de la necesidad de manejar múltiples conexiones directamente.

11.3.1. Arquitectura del proxy⁷

El proxy cuenta con una arquitectura modular basada en hilos concurrentes, cada uno con responsabilidades bien definidas. El diseño se centra en aceptar y coordinar conexiones tanto de clientes como de gateways de forma eficiente y desacoplada.

11.3.1.1. Hilo de escucha de clientes

⁷ Para mayor claridad, se dispone de una explicación guiada por un diagrama de actividad en la sección [5.2.1](#).

Este hilo (ClientsListener) opera sobre un puerto configurable (por defecto el **8000**) y se encarga de aceptar múltiples conexiones entrantes de clientes. Cada vez que un cliente se conecta:

- El proxy selecciona un **gateway disponible** mediante una política de balanceo round-robin.
- Se instancia un objeto ClientHandler, el cual encapsula completamente la lógica de comunicación con ese cliente.
- Este handler se ejecuta en un hilo independiente, garantizando que cada cliente pueda operar de manera concurrente y sin bloquear al resto del sistema.

El ClientHandler se encarga de:

- Registrar al cliente en las estructuras internas del proxy (client_id, socket, dirección).
- Notificar al gateway asignado que se ha conectado un nuevo cliente (NEW_CLIENT).
- Escuchar los mensajes del cliente, reenviarlos al gateway correspondiente, y manejar fallos de red o desconexiones.

Esto permite mantener una gestión encapsulada por cliente, simplificando el manejo de errores, reconexiones y el monitoreo individualizado.

11.3.1.2. Hilo de escucha de gateways

Este hilo (GatewaysListener) escucha en un puerto distinto (por defecto el **9000**) y acepta conexiones desde gateways que desean registrarse activamente en el sistema. Al conectarse un gateway:

- Se lee su identificador (gateway_id) para registrarla en el sistema.
- Si ya estaba conectado, se considera una **reconexión**: el proxy cierra el socket anterior y reasocia clientes suspendidos.
- Se inicia un **hilo dedicado** para recibir respuestas desde ese gateway y reenviarlas a los clientes correspondientes.

Este esquema permite que múltiples gateways se conecten y operen en paralelo, y que el sistema sea resiliente a reconexiones y caídas parciales.

11.3.2. Tolerancia a Fallos de Gateways

La disponibilidad de los gateways es fundamental para el funcionamiento correcto del sistema distribuido. Para evitar que la caída de uno de estos nodos derive en pérdida de servicio, se diseñó un **mecanismo resiliente de detección y reconexión automática** mediante el módulo GatewaysListener.

Este componente forma parte del contenedor Proxy, y se encarga de:

- Escuchar conexiones entrantes desde gateways.
- Identificarlos por un gateway_id único.
- Detectar **reconexiones**.
- Realizar la **limpieza y actualización** de clientes afectados.
- **Restaurar automáticamente** la conectividad de clientes suspendidos.

11.3.2.1. Identificación y Reconexión

Cuando un nuevo gateway se conecta al proxy, el GatewaysListener:

1. **Acepta la conexión** en el socket TCP (puerto 9000).
2. Espera el **identificador del gateway** (un byte que representa su gateway_id).
3. Verifica si ese gateway_id ya estaba registrado:
 - Si es una **nueva conexión**, se registra directamente.
 - Si es una **reconexión**, se ejecuta un procedimiento de **recuperación** detallado.

Este mecanismo garantiza que **el proxy siempre mantenga una única conexión activa por gateway** y evita inconsistencias ante reinicios o fallos parciales.

11.3.2.2. Protocolo de Reconexión

En caso de reconexión, se siguen estos pasos:

1. **Cierre limpio del socket anterior** del gateway (si estaba abierto).
2. Llamada a `proxy.handle_gateway_reconnection`, que:
 - Verifica la validez de todos los clientes asociados.
 - Descarta clientes inactivos.

- Restaura los clientes suspendidos para ese gateway_id.
- 3. Reasignación del nuevo gateway_socket a los clientes activos.
- 4. Reinicio del hilo ClientHandler si era necesario.

Este proceso se realiza en tiempo de ejecución, sin reiniciar el proxy ni perder la conexión con los clientes.

11.3.2.3. Restauración de Clientes Suspendidos

Una de las funcionalidades clave del diseño es la **suspensión temporal de clientes** cuando un gateway falla. En vez de desconectarlos por completo, el proxy:

- Mueve al cliente a una estructura self._suspended_clients, manteniendo su socket.
- Elimina la asociación temporal con el gateway caído.

Luego, en cuanto el gateway se reconecta, estos clientes se restauran a:

- self._connected_clients
- self._clients_per_gateway
- Se actualiza su nuevo gateway_socket

Esto permite mantener una **sesión persistente a pesar de reinicios transitorios** del backend, algo esencial en sistemas que requieren **alta disponibilidad sin pérdida de estado**.

11.3.2.4. Robustez ante errores

El GatewaysListener implementa múltiples validaciones para garantizar estabilidad:

- Manejo de excepciones detallado en la aceptación de conexiones.
- Timeout de accept() no bloqueante para soportar detención con self._stop_flag.
- Validación de socket de cliente (fileno()) antes de intentar actualizarlo.
- Cierre ordenado de sockets en todas las ramas posibles.
- Logging exhaustivo para trazabilidad operativa.

11.3.2.5. Coordinación con el Proxy

Este componente colabora estrechamente con el objeto principal Proxy, con acceso y modificación segura a las siguientes estructuras compartidas:

- `_gateways_connected`: actualización de sockets activos.
- `_clients_per_gateway`: reubicación y purga de clientes según estado.
- `_connected_clients`: verificación y sustitución de conexiones.
- `_client_handlers`: reinicio de hilos de cliente según necesidad.

11.4. Tolerancia a fallos de los nodos Gateways

El diseño del proxy y los gateways incorpora mecanismos explícitos para garantizar la **resiliencia** y la **continuidad del servicio** frente a fallos o desconexiones inesperadas de los nodos gateways.

11.4.1. Reconexión y recarga de estado de clientes por parte del Gateway

Cuando un gateway pierde conexión y posteriormente se reconecta al proxy, se ejecuta un proceso de recuperación integral que garantiza la consistencia del sistema:

- El proxy detecta la reconexión del gateway mediante la identificación única (`gateway_id`) que este envía al establecer el socket.
- El proxy invoca el método `handle_gateway_reconnection`, que limpia los clientes inválidos asociados al gateway desconectado y reactiva los clientes previamente suspendidos.
- Para cada cliente activo, el proxy actualiza la referencia del socket del gateway al nuevo socket reestablecido.
- Se reinician los hilos encargados de manejar las respuestas del gateway y la comunicación bidireccional con cada cliente.

De esta manera, el gateway vuelve a tener el estado actualizado de sus clientes, sin pérdida de información ni necesidad de reestablecer manualmente las conexiones.

11.4.2. Persistencia de batches y respuestas

Para maximizar la tolerancia a fallos y evitar pérdida de datos ante desconexiones o reinicios, cada gateway implementa un mecanismo de almacenamiento persistente:

- Los batches (lotes de datos o mensajes) que cada cliente envía al gateway son almacenados en disco local, garantizando que en caso de caídas la información pueda ser recuperada.
- Las respuestas que el gateway debe enviar a los clientes también se guardan localmente antes de ser enviadas.
- Este almacenamiento en disco permite que, luego de una caída y posterior reconexión, el gateway pueda reanudar el procesamiento y envío de respuestas exactamente donde lo dejó, evitando duplicados o pérdidas.
- Esta estrategia asegura una alta confiabilidad y robustez, incluso en entornos con múltiples nodos y cargas concurrentes.

11.5. Tolerancia a fallos en nodos de procesamiento

11.5.1. Elección de líder

El sistema implementa un **Algoritmo de Elección Bully** (módulo LeaderElector en el programa) modificado para sistemas distribuidos, donde cada nodo tiene un ID único y el nodo con el ID más alto se convierte en líder. El algoritmo incluye mecanismos de detección de fallos mediante heartbeat y recuperación automática ante fallos del líder.

11.5.1.1. Criterio de Liderazgo

- Regla Principal: El nodo con el ID más alto se convierte en líder
- Identificación Única: Cada nodo tiene un node_id único.
- Jerarquía: Los nodos con IDs más altos tienen prioridad sobre los de IDs menores

11.5.1.2. Identificación y Comunicación entre Nodos

La comunicación entre nodos se realiza mediante mensajes UDP sobre sockets. Cada nodo escucha en un puerto definido (`election_port`), y se comunica con sus pares definidos en el diccionario `peers`, o nodos pares, que mapea `node_id → (host, port)`.

11.5.1.2.1. UDP

UDP fue elegido como protocolo de transporte por ser rápido, ligero y sin conexión, ideal para mensajes breves y frecuentes como **HEARTBEAT** y **ELECTION**. Sin embargo, UDP no garantiza la entrega, el orden ni la unicidad de los mensajes, por lo que se integraron varias estrategias para tolerar sus fallos:

- Reintentos de mensajes críticos: los mensajes **COORDINATOR** se envían tres veces, con espera de 2 segundos entre envíos.
- Timeouts: si un nodo no recibe respuestas (**ALIVE** o **LEADER**) en 5 segundos, asume la falta de liderazgo y actúa.
- Validaciones antes de aceptar un líder: cada nodo valida si el líder propuesto es legítimo antes de aceptarlo.
- Monitoreo de HEARTBEAT: la detección de fallos se basa en la ausencia continua de estos mensajes, no en la pérdida aislada.

11.5.1.2.2. Fuente de la Configuración: Variables de Entorno vía Docker Compose

Tanto el `node_id` (identificador único del nodo), la lista de `peers` (otros nodos con sus respectivos `host:port`) como el `election_port` (puerto de escucha) son proporcionados como variables de entorno a través de un archivo `docker-compose.yml`.

Definiciones:

- **NODE_ID**: El ID único de cada nodo,
- **PEERS**: Cadena con formato "nombre_1:puerto,nombre_2:puerto,..." que representa los otros nodos del sistema.
- **ELECTION_PORT**: Puerto UDP en el que cada nodo escuchará los mensajes de elección.

11.5.1.3. Flujo de Inicialización

- Al iniciar un nodo: Se crea el socket y se lanzan los hilos de escucha (`listen`) y `heartbeat` (`_heartbeat_sender`, `_heartbeat_monitor`).

- Se envía el mensaje **WHOISLEADER** a todos los peers, solicitando información sobre el líder actual.
- Si en 5 segundos no se recibe respuesta (**LEADER**), y no hay elección activa, se inicia la elección con `start_election()`.

11.5.1.4. Mecanismo de Heartbeat

El sistema implementa un mecanismo de latido (heartbeat) para detectar caídas de nodos de manera pasiva y sin intervención directa. Este mecanismo es fundamental para la estabilidad del clúster y para tomar decisiones sobre elecciones de líder o reintegración de nodos previamente fallidos.

11.5.1.4.1. Funcionamiento General

Cada nodo envía un mensaje **HEARTBEAT** cada 3 segundos a todos sus peers. Si un nodo no recibe un heartbeat de otro peer en un intervalo de 10 segundos, considera que dicho nodo ha fallado temporalmente o está desconectado.

11.5.1.4.2. Implementación con Threads

Para gestionar el envío y recepción de estos mensajes sin bloquear el resto del sistema, se utilizan **dos hilos (threads) en segundo plano**, iniciados automáticamente al arrancar el nodo:

- **Thread 1: _heartbeat_sender**
 - Envía periódicamente (cada 3 segundos) un mensaje **HEARTBEAT** a todos los peers conocidos.
 - Utiliza el mismo canal de comunicación UDP y evita validaciones de red innecesarias para reducir la sobrecarga.
 - Corre continuamente mientras `heartbeat_running` esté activo.
- **Thread 2: _heartbeat_monitor**
 - Monitorea los timestamps del último **HEARTBEAT** recibido por cada peer.
 - Si detecta que han pasado más de 10 segundos desde el último latido de un peer, marca ese nodo como fallido.

- En caso de que el nodo fallido fuera el líder actual o un nodo con ID superior, se lanza automáticamente una nueva elección, con un retardo aleatorio para evitar colisiones entre múltiples nodos.
- Si más adelante se vuelve a recibir un **HEARTBEAT** desde un nodo previamente marcado como fallido, este se reintegra automáticamente al sistema.

11.5.1.4.3. Robustez frente a falsos positivos

El sistema evita reaccionar ante una única pérdida de heartbeat (algo común con UDP) y sólo actúa si la ausencia persiste durante 10 segundos. Esta ventana permite cierta tolerancia a la pérdida de paquetes sin disparar falsamente una elección.

11.5.1.5. Recepción y Procesamiento de Mensajes

El método `handle_message()` gestiona los distintos tipos de mensajes:

- **WHOISLEADER**: si el nodo conoce al líder, responde con **LEADER**.
- **LEADER**: actualiza su `leader_id` si es válido.
- **ELECTION**: si el emisor tiene ID menor, responde con **ALIVE** y evalúa si debe iniciar una elección.
- **ALIVE**: indica que un nodo con mayor ID está activo; se detiene la elección local.
- **COORDINATOR**: actualiza el líder si pasa las validaciones.
- **HEARTBEAT**: actualiza el estado de conexión con el peer.

11.5.1.6. Proceso de Elección

Cuando un nodo decide iniciar una elección (por no recibir **LEADER**, detectar caída del líder o recibir **ELECTION** de un nodo), ejecuta el método para inicializar la elección.

11.5.1.6.1. Verificación Preliminar

Chequea si hay nodos con id mayor disponibles,

- Si ya hay un líder estable y no debe ser desafiado, es decir, su ID es mayor al nodo actual, no hace mas.
- Si no hay peers con ID mayor disponibles (según *heartbeat*), el nodo intenta proclamarse líder.

11.5.1.6.2. Envío de Mensajes ELECTION

El nodo envía **ELECTION** a todos los peers con `node_id` mayor y activos. Si recibe al menos un **ALIVE**, espera que ese peer se autoproclame líder. Si en 5 segundos no recibe nada, se declara líder enviando mensajes del tipo **COORDINATOR**.

11.5.1.7. Anuncio del Coordinador

Cuando un nodo se declara líder:

1. Verifica que no haya peers activos con mayor ID.
2. Establece su `leader_id` local.
3. Envía el mensaje **COORDINATOR** tres veces a todos los peers activos.
4. Ejecuta la lógica asociada con el nuevo liderazgo.

11.5.1.8. Detección y Manejo de Fallos

Cuando el monitor de **HEARTBEAT** detecta la caída de un nodo:

- Lo marca como fallido.
- Si ese nodo era el líder o tenía mayor ID, se inicia una nueva elección.
- Si el nodo era irrelevante (por ejemplo, con menor ID), se ignora.

Si un nodo que estaba marcado como fallido vuelve a enviar **HEARTBEAT**, se reintegra automáticamente al sistema.

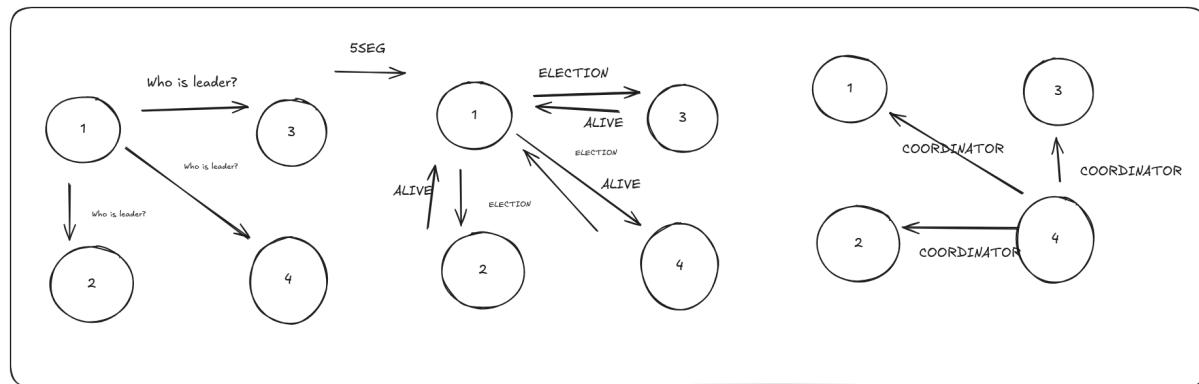
11.5.1.9. Escenarios

Escenario 1: Startup Normal (Inicio Simultáneo de Nodos)

Todos los nodos inician al mismo tiempo en una red sin líder preestablecido.

1. Cada nodo inicia su `LeaderElector` y comienza a escuchar mensajes.
2. Todos envían mensajes **WHOISLEADER** para consultar si algún nodo ya es líder.

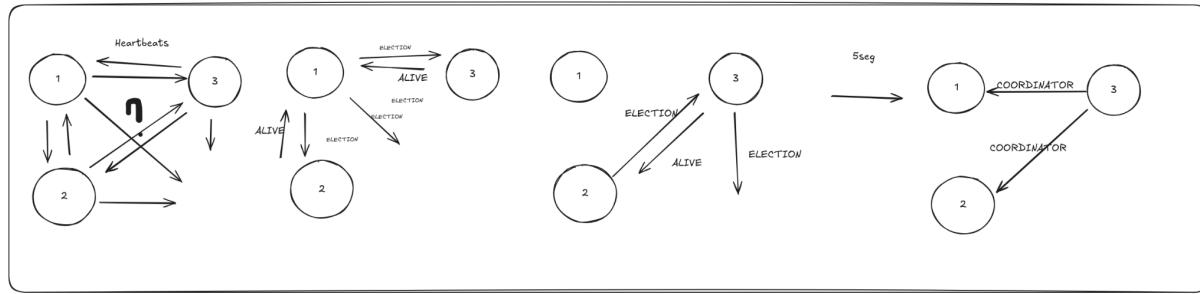
3. Como ninguno ha sido elegido aún, no se recibe ningún mensaje LEADER.
4. Cada nodo espera 5 segundos. Si no recibió un LEADER, y no hay una elección en curso, inicia una elección.
5. Los nodos envían mensajes **ELECTION** a peers con mayor node_id.
6. Sólo el nodo con ID más alto y que no recibe ninguna respuesta ALIVE espera el timeout de 5 segundos y se proclama líder.
7. El nuevo líder envía mensajes COORDINATOR a todos los peers activos.



Escenario 2: Fallo del Líder

El nodo que actualmente actúa como líder deja de enviar heartbeats por caída.

1. Los peers monitorean continuamente los **HEARTBEAT** del líder.
2. Cuando no reciben **HEARTBEAT** por más de 10 segundos, detectan su falla.
3. Cada nodo que detectó la falla lanza una elección (con retardo aleatorio entre 0.5 y 1.5s para evitar colisiones).
4. Cada uno envía **ELECTION** a sus peers.
5. Los nodos con ID menor no hacen nada más que responder **ALIVE**.
6. El nodo con mayor ID, sin recibir ningún **ALIVE**, se autoproclama líder y envía **COORDINATOR**.



Escenario 3: Fallo de un Nodo que NO es el Líder

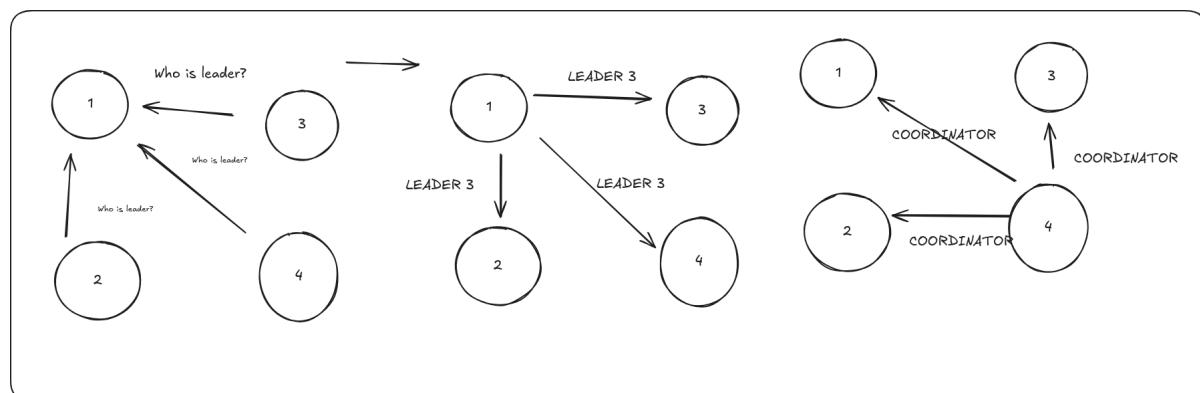
Un nodo que no es el líder falla o se desconecta.

1. Los demás nodos detectan la ausencia de **HEARTBEAT** y marcan al nodo como fallido.
2. Se actualiza el estado interno, pero no se lanza una elección, ya que el líder sigue activo y no se ve afectado.

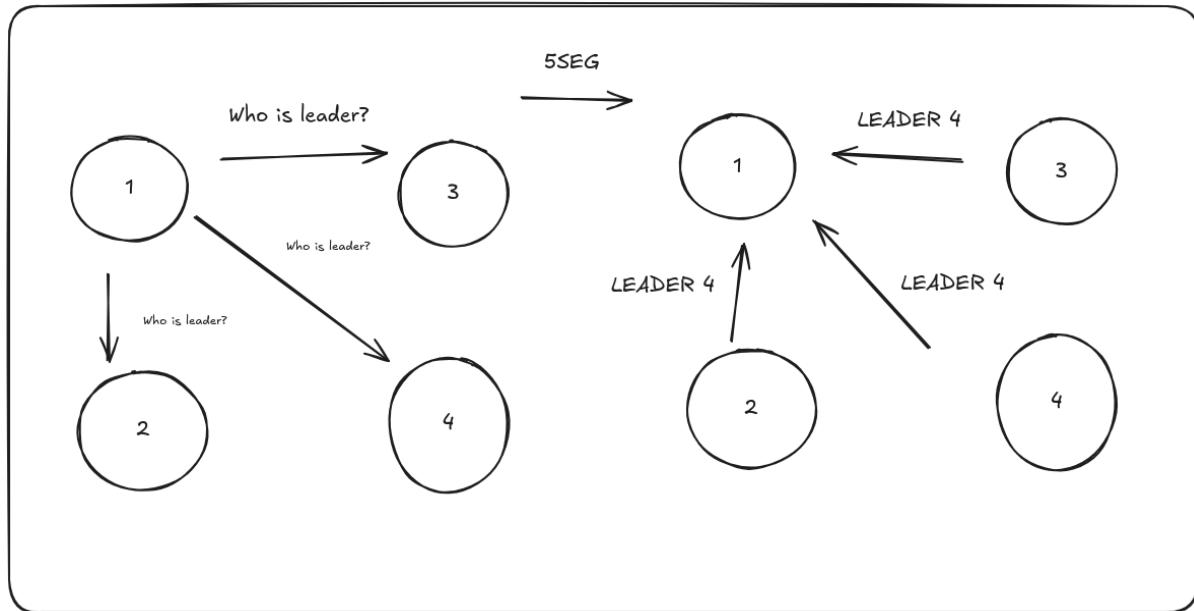
Escenario 4: Recuperación de Nodo Fallido

Un nodo previamente fallido vuelve a la red.

1. El nodo reinicia su LeaderElector, comienza a escuchar y envía **WHOISLEADER**.
2. Si recibe respuesta **LEADER**, actualiza su estado interno y se reintegra normalmente al clúster.
3. Si tiene un ID mayor que el actual líder, puede iniciar una nueva elección, donde el será el ID mayor y mandará **COORDINATOR**.



4. Si no lo hace, simplemente se mantiene como seguidor.

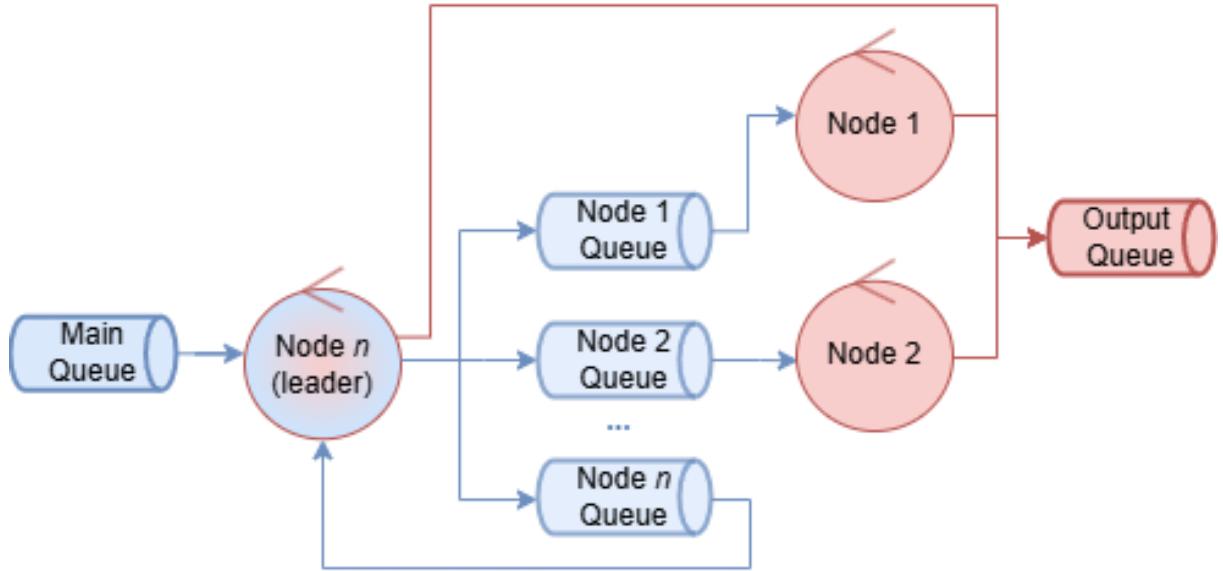


11.5.2. Lógica Master-Slave

11.5.2.1 Balanceo de carga

Una vez elegido un nodo líder entre un grupo de nodos del mismo tipo, este asume el rol de **Master** y comienza a encargarse del **balanceo de carga** (load balance). Todos los nodos incluyen un proceso llamado MasterLogic, que se mantiene a la espera de que se defina el liderazgo según la lógica de elección previamente implementada.

Cada nodo posee su propia **cola de entrada** desde la cual consume mensajes. El nodo líder, en cambio, consume desde una **cola principal compartida** y se encarga de redistribuir los mensajes entre sus pares. Para determinar a qué nodo reenviar cada mensaje, se utiliza su identificador (message_id), lo que garantiza **consistencia determinista** en caso de reprocesamientos o duplicados.



En el caso particular de nodos join con **sharding** de las tablas de películas, el líder también es responsable de redirigir los mensajes al shard adecuado.

Importante: aunque el nodo líder tiene un rol especial, **sigue actuando como un nodo común**, es decir, también publica mensajes hacia su propia cola como cualquier otro nodo del conjunto. Esta separación entre las funcionalidades de liderazgo y de procesamiento podría implementarse mediante comunicación entre procesos internos. Sin embargo, se optó por mantener al líder como un proceso abstracto que interactúa exclusivamente a través de colas, tratándose a sí mismo como cualquier otro nodo.

Ventajas

- **Abstracción de responsabilidades:** el líder no necesita saber si está enviando a sí mismo o a otro nodo. Simplemente escribe a una cola destino, sin lógica especial de comunicación interna.
- **Uso del sistema de colas como frontera de confiabilidad:** al enviar siempre por RabbitMQ, se aprovecha el mecanismo de confirmación (ACK) del broker. Si el líder falla antes de enviar el mensaje a otro nodo, el mensaje simplemente no se “acknowledges”, por lo que RabbitMQ lo redirige a otro líder. Esto facilita la recuperación segura ante fallos.

- **Escalabilidad y modularidad:** esta arquitectura permite que los nodos escalen horizontalmente y sean reemplazables o reiniciables sin romper la lógica general del sistema.

Desventajas

- **Recorrido adicional del mensaje:** incluso si el nodo líder es el destino del mensaje, este debe pasar por RabbitMQ y luego ser reconsumido por el mismo. Esto introduce latencia y overhead innecesario.
- **Sobrecarga en el líder:** el nodo líder debe consumir, reenviar y seguir procesando sus propios mensajes, lo que puede convertirse en un cuello de botella.

11.5.2.2. Manejo y recuperación de EOS (End Of Stream)

Cuando el nodo líder recibe un mensaje de tipo EOS, lo distribuye a todos sus nodos pares. Si un nodo se cae y luego vuelve a iniciar, el Coordinator le notificará que se ejecute en modo de recuperación mediante un archivo a modo de *flag*. Una vez lanzado el nodo en ese modo, le envía un mensaje de tipo RECOVERY al líder. Este responde reenviando los EOS pendientes a la cola del nodo recuperado, junto con un mensaje de confirmación.

Un nodo en estado de recuperación **no participa en elecciones de líder** hasta que se le notifica que su proceso de recuperación ha finalizado. Dado que según nuestras restricciones siempre consideramos que habrá al menos un nodo activo, la recuperación del sistema está garantizada (siempre y cuando no se caiga el último nodo líder mientras sucede la recuperación).

11.5.2.3. Fallo del nodo líder durante la redistribución

Si el líder falla mientras está procesando un mensaje (y todavía no ha enviado el ACK a RabbitMQ), el mensaje será entregado a un nuevo líder. Esto puede causar **duplicación de mensajes**, pero asegura que **el mensaje será eventualmente procesado al menos una vez**. La duplicación es gestionada de forma local por cada nodo.

11.5.2.4. Control de duplicados

Tanto los EOS como los mensajes para las tablas de películas son **idempotentes**, quiere decir que de llegar duplicados no afectarán al estado actual de los nodos si es que ya recibieron esa información.

Para los mensajes comunes, cada nodo implementa un objeto `DuplicateHandler`, responsable de detectar y descartar mensajes repetidos. Este componente utiliza una **LRU cache (Least Recently Used)** para almacenar los identificadores de los últimos n mensajes procesados. Los mensajes más antiguos se eliminan a medida que llegan nuevos.

No se usa un sistema de numeración secuencial (por ejemplo, mantener el mayor `message_id` recibido) debido a la **naturaleza asincrónica y fuera de orden** del sistema, donde los mensajes pueden llegar en cualquier orden y desde distintas fuentes.

12. Iteraciones futuras

Persistencia de duplicados en disco

Actualmente, los duplicados se manejan en memoria mediante una LRU cache. Se propone una mejora que permita **volcar estos identificadores a disco**, lo que ofrecería una mayor persistencia frente a reinicios o fallos prolongados. Esto permitiría que los nodos puedan **reconocer y descartar mensajes duplicados incluso tras haberse caído y recuperado**, sin depender exclusivamente de la memoria RAM.

Comunicación de recuperación por UDP

Los mensajes de tipo RECOVERY se envían a través de RabbitMQ, y si el sistema de colas tiene problemas, podría afectar la recuperación de nodos. Se propone migrar esta comunicación a **mensajes ligeros vía UDP**, lo cual reduciría la latencia y eliminaría la dependencia del sistema de colas para esta etapa crítica. La confirmación podría implementarse con una lógica simple de reintentos sobre UDP.

Otra alternativa para manejar la recuperación podría ser utilizar una **memoria compartida distribuida**, que sea accesible por todos los nodos a través de un sistema externo junto con algún mecanismo de **coordinación distribuida** (por ejemplo locks). Esto permitiría a los nodos registrar y consultar su estado de forma eficiente, evitando la dependencia de RabbitMQ para la coordinación de recuperación, aunque implicaría mayor complejidad e infraestructura adicional.

13. Anexo

13.1. Referencias

- Link a los diagramas:

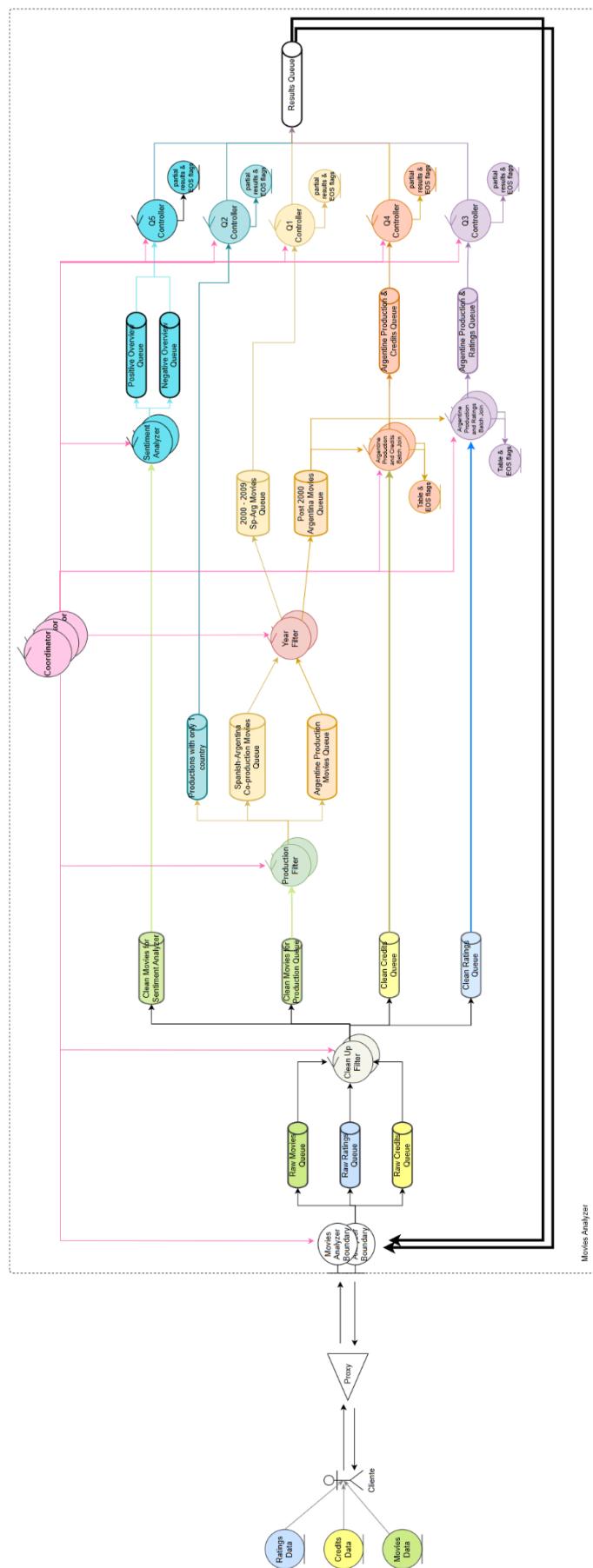
https://drive.google.com/file/d/15dcFuXlb_mMzxmrfxLuxFFdnBSae8ah3/view?usp=sharing

- Link al repositorio:

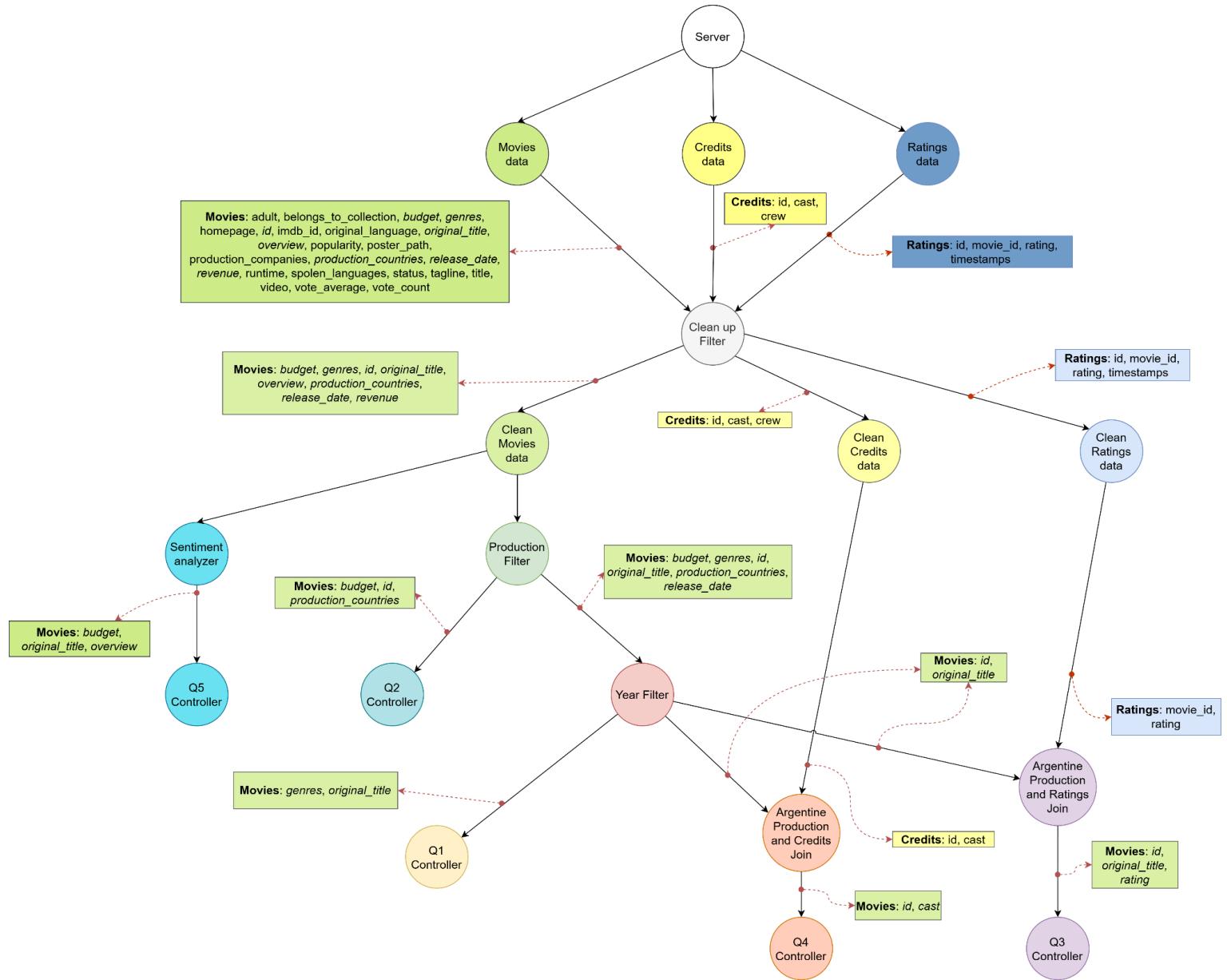
<https://github.com/JuanPF56/TP-SDI>

13.2. Diagramas completos

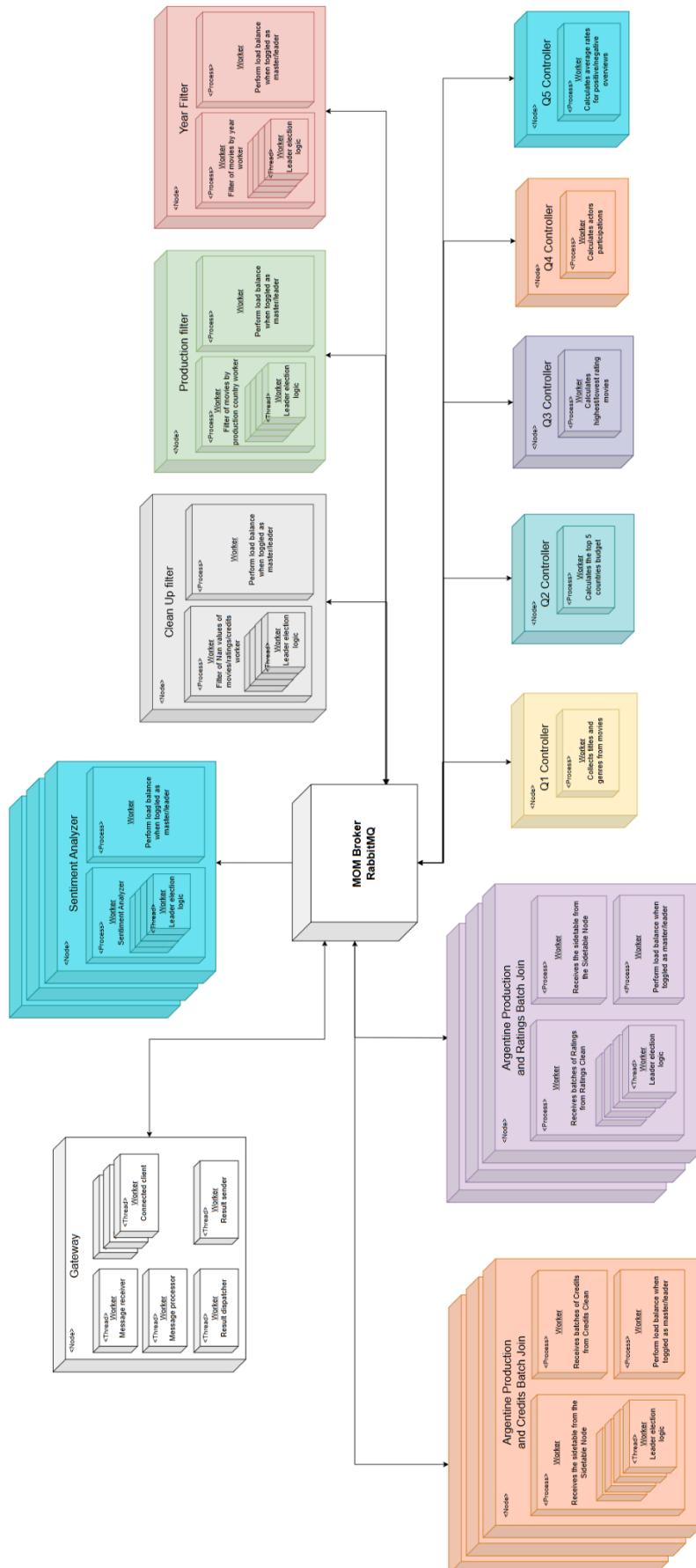
13.2.1. Diagrama de robustez completo



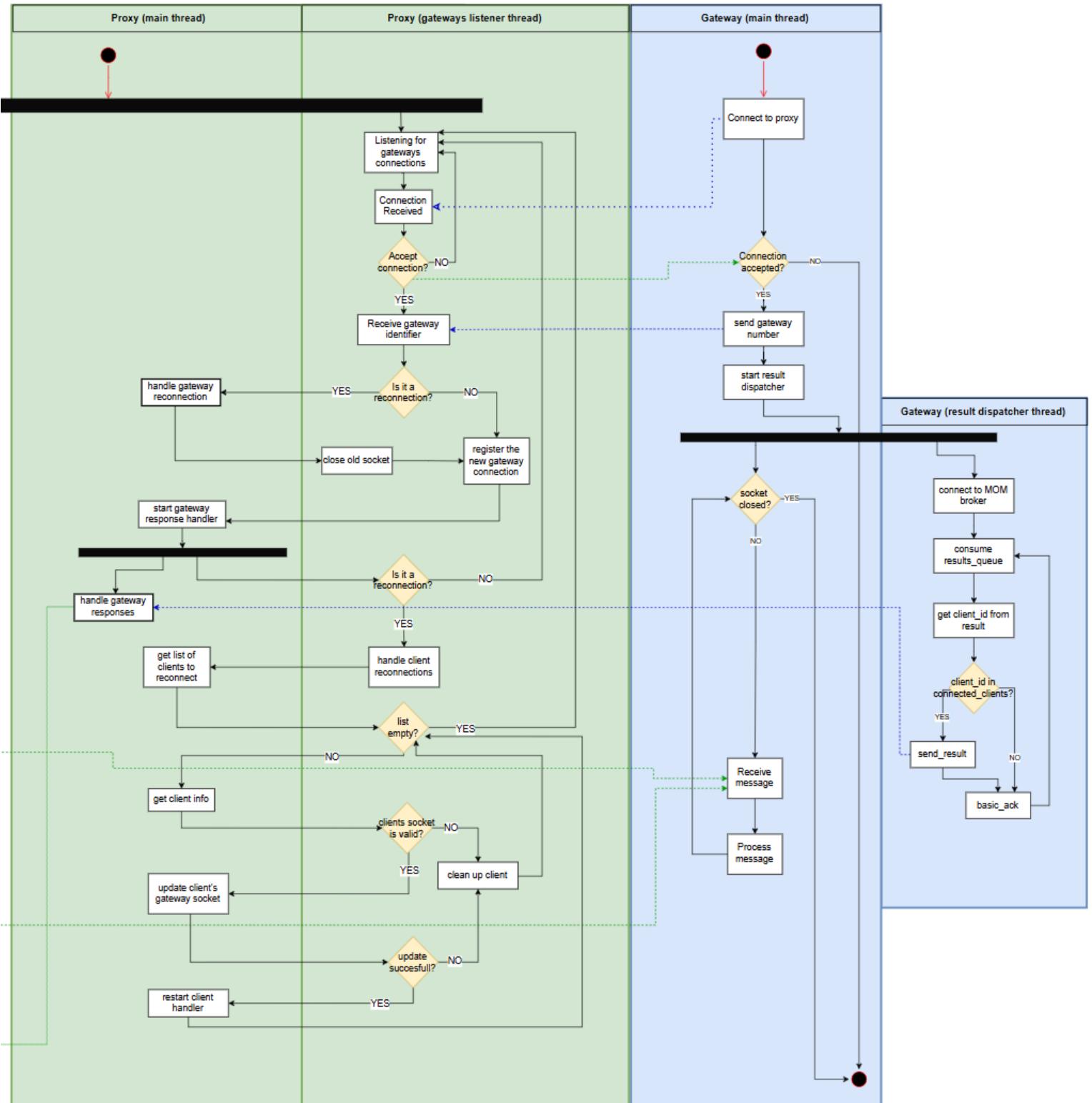
13.2.2. Diagrama DAG completo



13.2.3. Diagrama de despliegue completo



13.2.4. Diagrama de actividades: Proxy - Gateway listener completo



13.2.5. Diagrama de actividades: Proxy - Clients listener completo

