Trabajo Práctico Diseño:
Movies Analysis

[75.74] Sistemas Distribuidos I

Grupo 13

Bianchi Fernandez, Marcos	108921	mbianchif@fi.uba.ar
Gentilini, Franco	108733	fgentilini@fi.uba.ar
Ghosn, Lautaro Gabriel	106998	lghosn@fi.uba.ar

Índice

Escenarios

El proyecto Movies Analysis tiene como objetivo analizar diversas variables relacionadas con un conjunto de datos de películas, utilizando un enfoque basado en sistemas distribuidos. El propósito es obtener los siguientes resultados:

CU1 - Producción Argentina y España (2000s)

Utilizando el dataset de películas el sistema obtiene los títulos y géneros de cada una de las películas estrenadas en la década de los años 2000 con una producción Argentina y Española.

CU2 - Top 5 Inversores en Producciones Nacionales

Utilizando el dataset de películas el sistema obtiene los 5 nombres de países que hayan invertido más dinero en producción tomando en cuenta solamente aquellas películas que hayan sido producidas por un único país.

CU3 - Mejor y Peor Rating – Cine Argentino 2000+

Utilizando los datasets de películas y ratings el sistema obtiene el título y rating de las películas con mayor y menor rating, siendo estas producidas en Argentina a partir del año 2000.

CU4 - Actores Más Frecuentes en Argentina desde 2000

Utilizando los datasets de películas y créditos, el sistema obtiene los nombres de los actores que hayan tenido más apariciones en películas argentinas con una fecha de estreno posterior al año 2000.

CU5 - Tasa Ingreso/Presupuesto: Positivo vs. Negativo

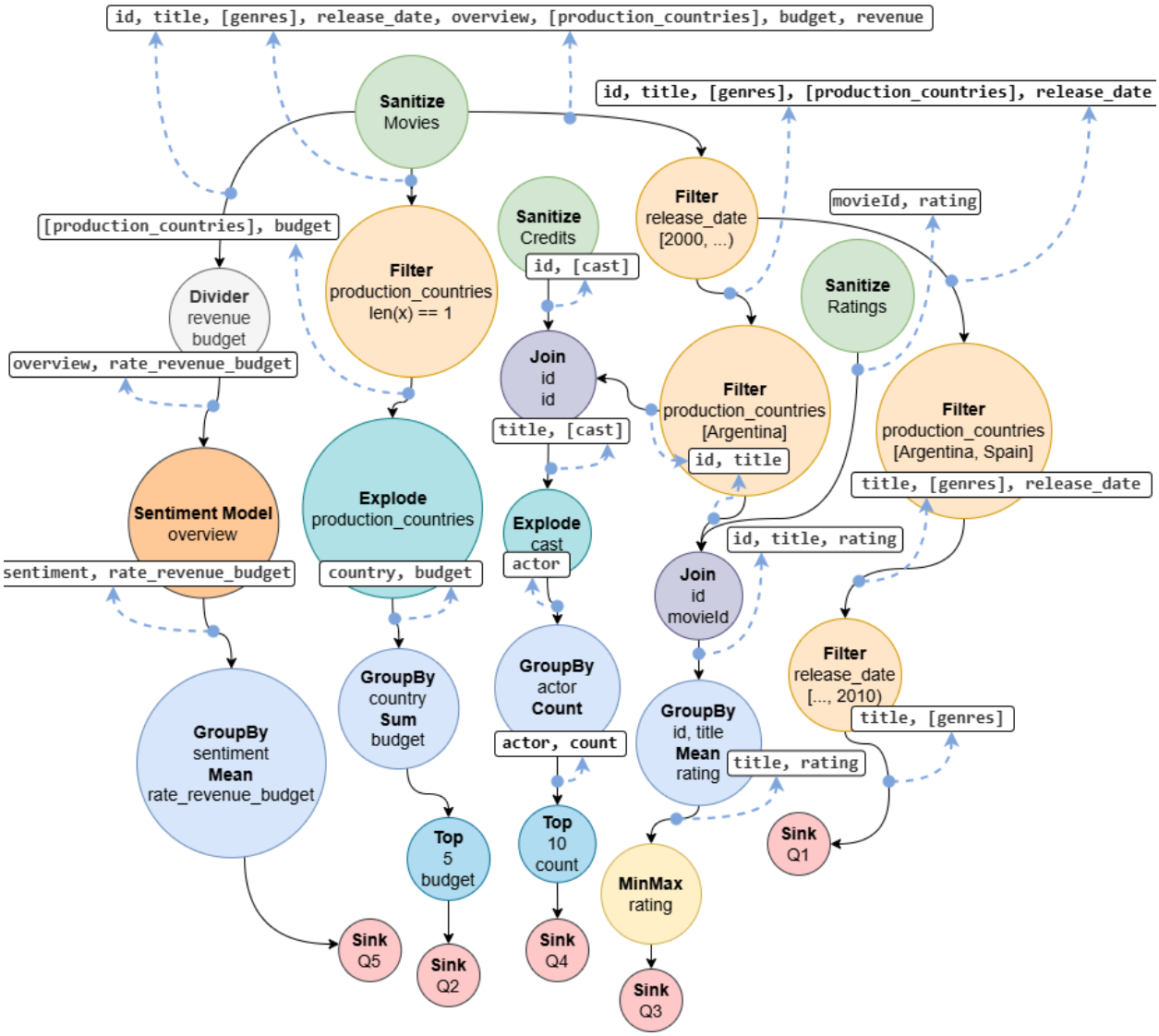
Utilizando el dataset de películas, el sistema obtiene el promedio de la tasa de ingreso/presupuesto por cada sentimiento obtenido de las reseñas de cada película utilizando un modelo de machine learning.

Arquitectura del sistema

Flujo de información en las tareas

A continuación se encuentra el diagrama de dígrafo acíclico del sistema, donde se pueden diferenciar los distintos tipos de nodos que interactúan para la correcta resolución de queries pedidas en el enunciado.

Los nodos de color verde son aquellos que producen la información correspondiente al nombre, funcionando como fuentes de información. Los nodos rojos (sinks) son consumidores finales de la información, llegado este punto las queries fueron resueltas y deben ser informadas al cliente.



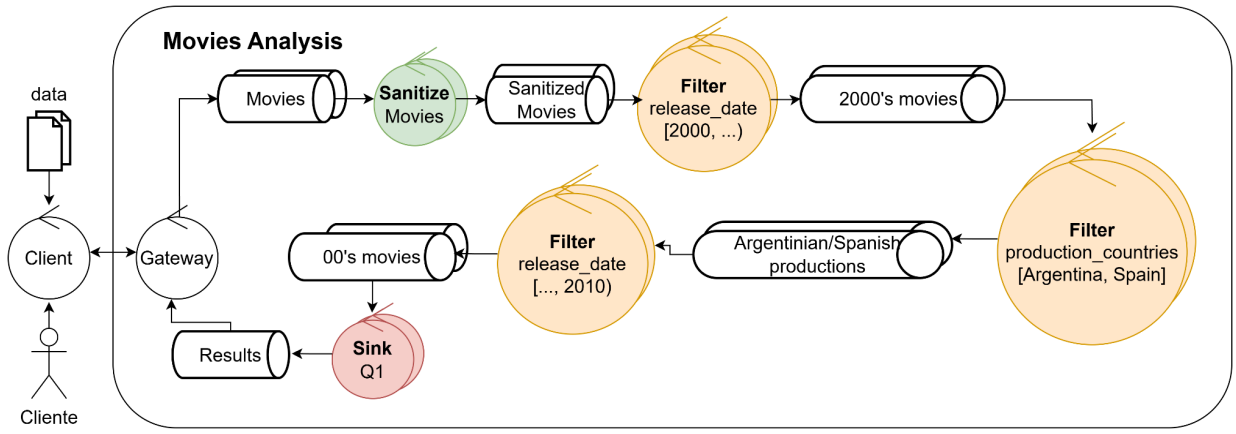
Comunicación entre procesos

El diagrama presentado permite visualizar de forma clara cómo interactúan las distintas entidades del sistema, en conjunto con las colas de mensajes que facilitan la comunicación asincrónica entre componentes.

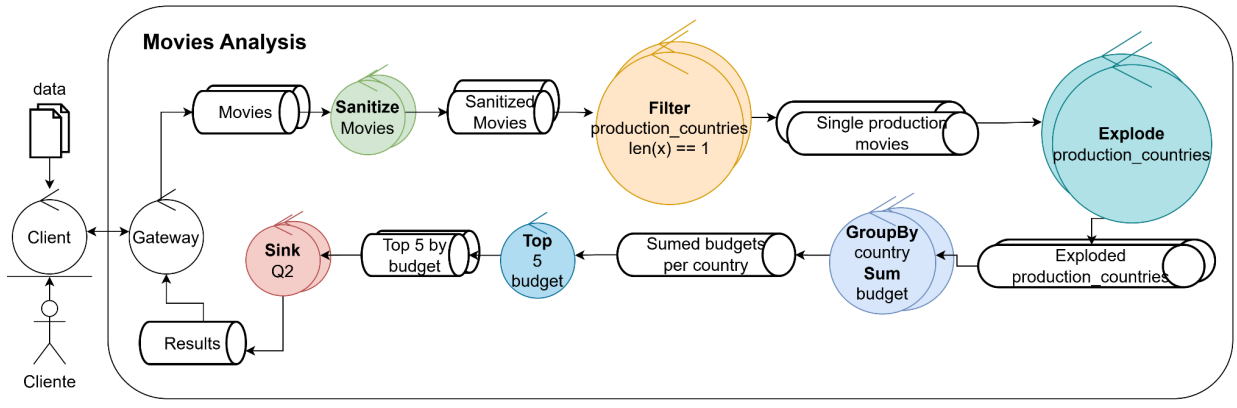
El sistema cuenta con varios tipos de workers, los cuales pueden ejecutarse en múltiples instancias para asegurar escalabilidad y paralelismo. Algunos ejemplos de estos workers son

filtros, explosiones, analizador de sentimientos, joins y agrupamientos. Particularmente para comunicar con las colas en estos dos últimos se hace mediante sharding permitiendo la distribución correcta de trabajo.
Por otro lado, hay otras que de momento creemos deberían correr de forma única y llevar consigo un estado interno para calcular sus resultados, estas instancias son por ejemplo los top-k y el worker minmax que calcula el mínimo y el máximo de una de las columnas.
Con el objetivo de facilitar la comprensión del flujo general del sistema, el diagrama ha sido estructurado en diferentes casos de uso, permitiendo una visualización más segmentada y enfocada de las operaciones involucradas.

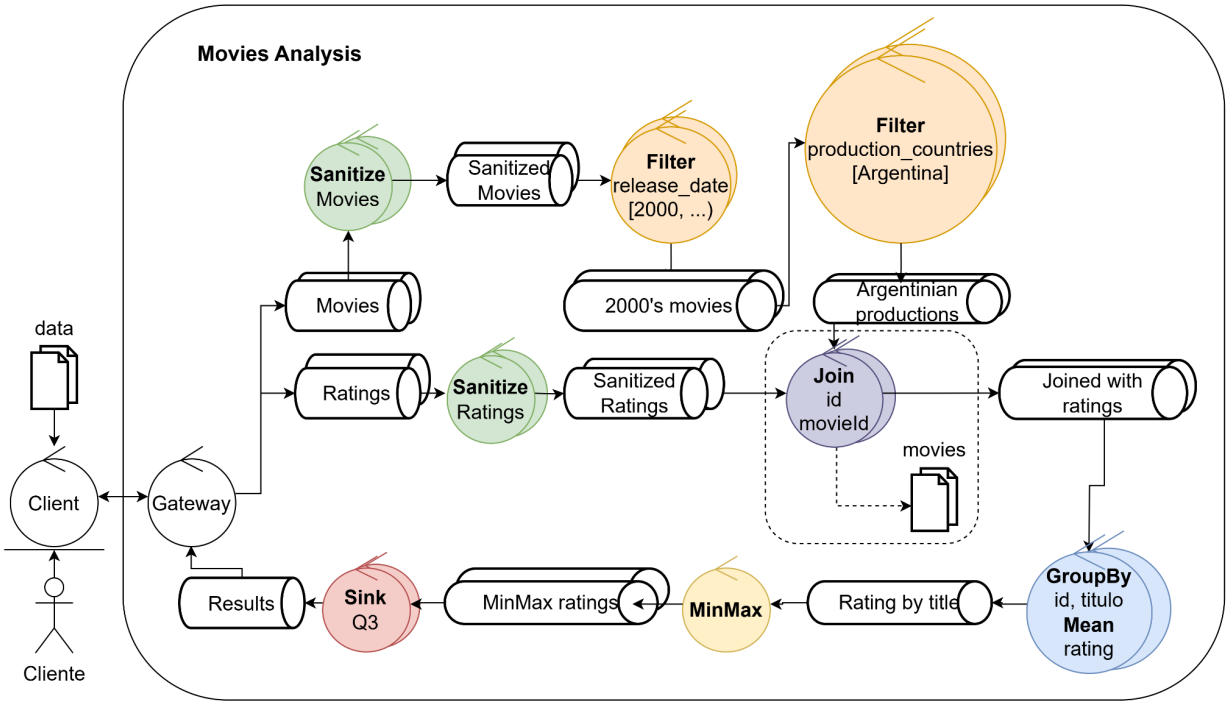
CU1 - Producción Argentina y España (2000s)



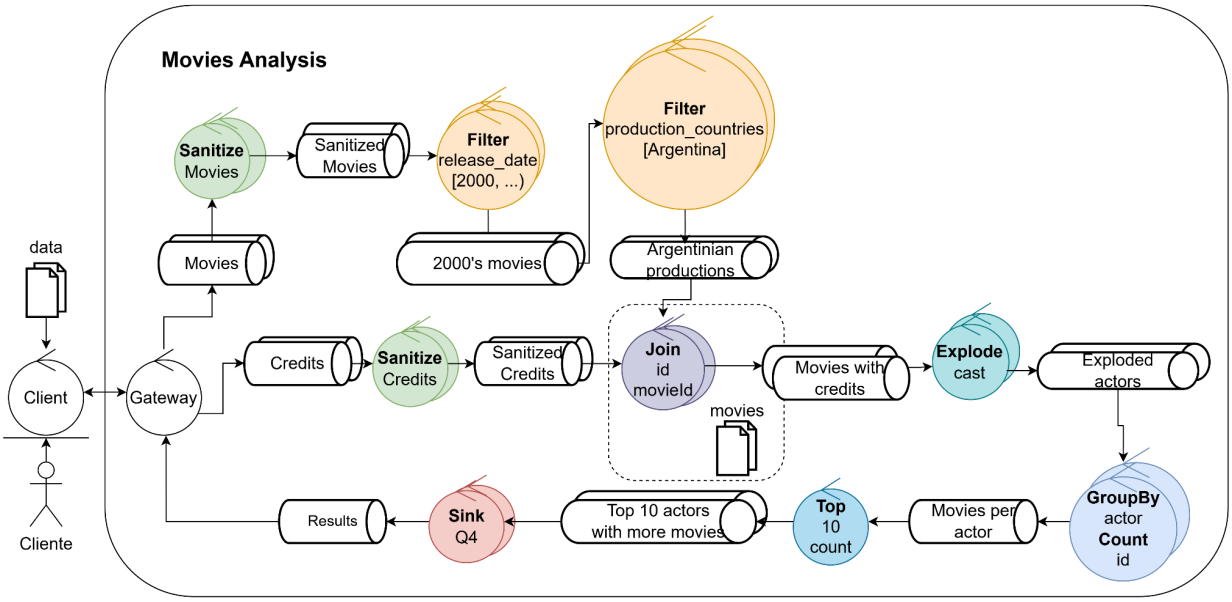
CU2 - Top 5 Inversores en Producciones Nacionales



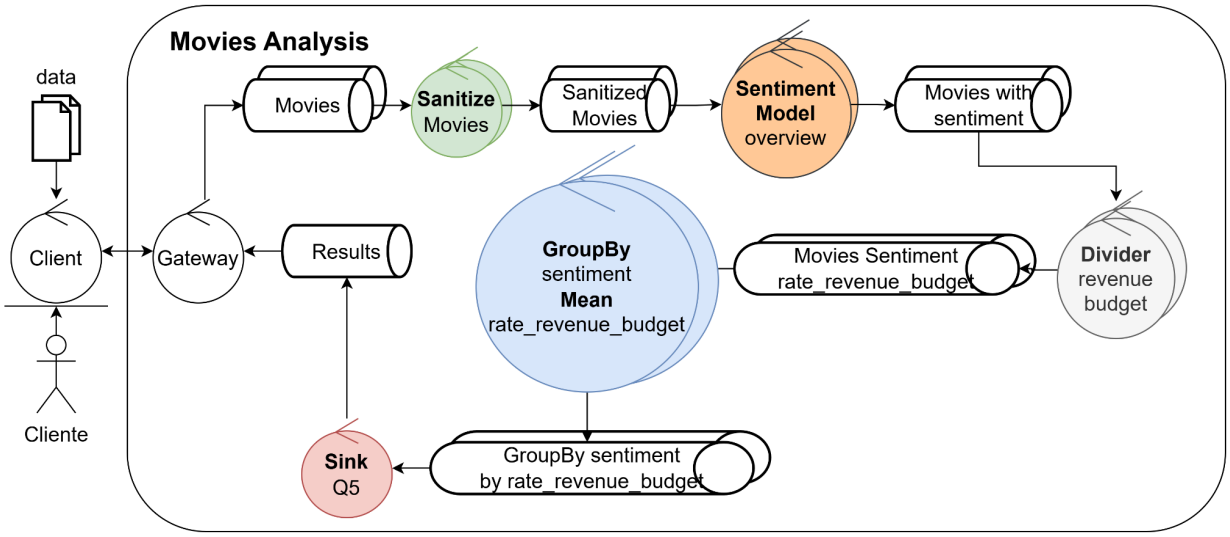
CU3 - Mejor y Peor Rating – Cine Argentino 2000+



CU4 - Actores Más Frecuentes en Argentina desde 2000



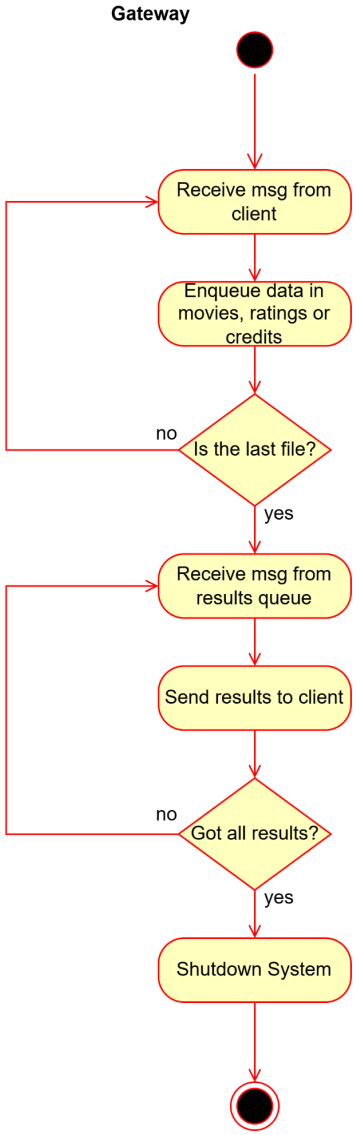
CU5 - Tasa Ingreso/Presupuesto: Positivo vs. Negativo



Casos particulares

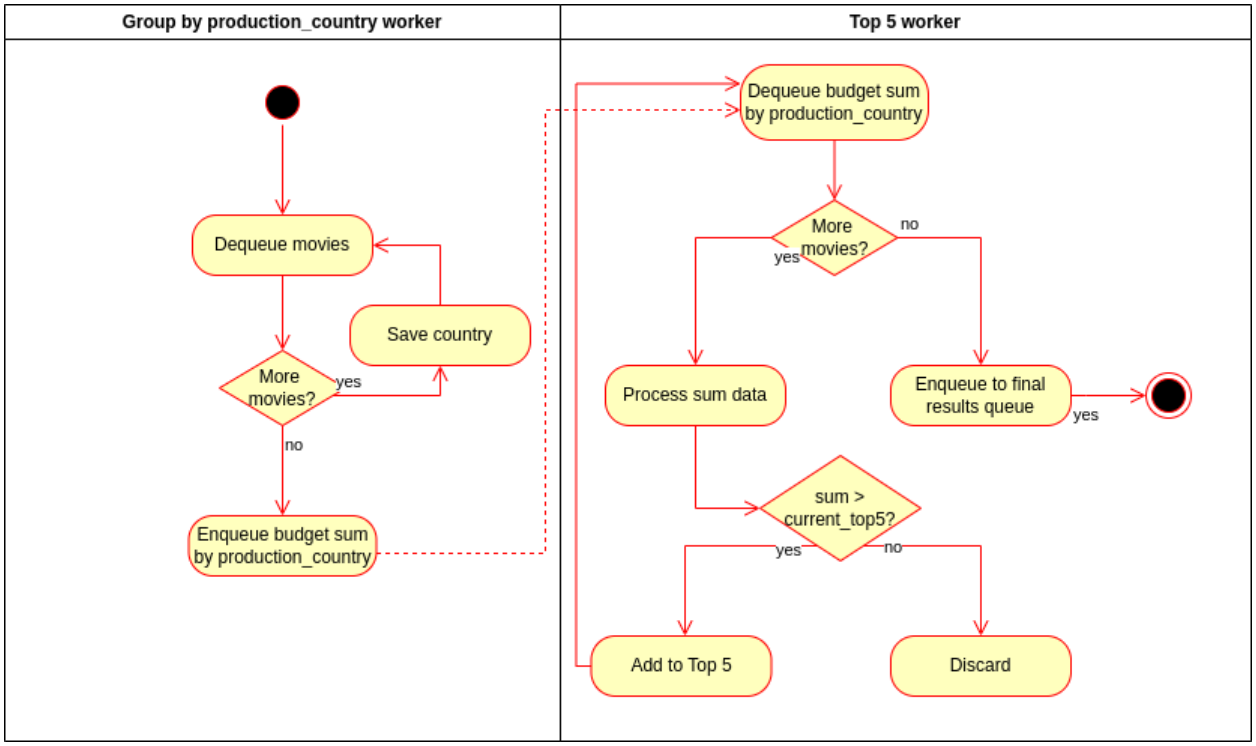
Manejo de mensajes del gateway

El siguiente flujo muestra el comportamiento del gateway, primero espera a recibir todos los datos necesarios del cliente para luego esperar por los resultados de las distintas queries para a medida que llegan ser enviados al cliente. Una vez termina, se hace shutdown al sistema.



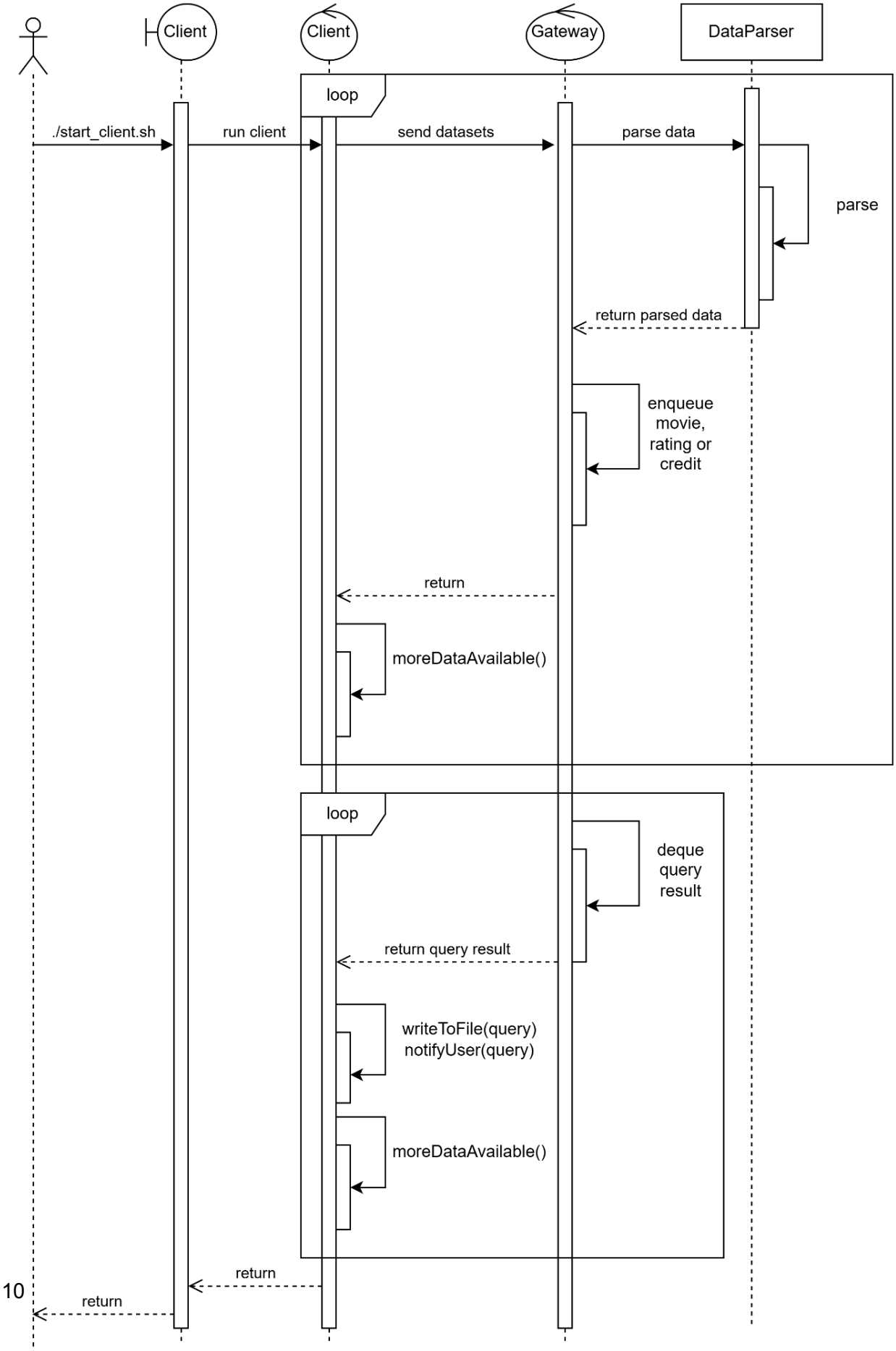
Top 5 de países

En este diagrama se puede observar como es el flujo para obtener el top 5 países con mayor dinero invertido. El Group by production_country worker va desencolando películas y las agrupa por país sumando el dinero invertido, para luego encolarlas en la cola del top 5 worker para que pueda obtener el top de países con mayor dinero invertido.



Interacción entre componentes del sistema

En este diagrama de secuencia se muestra la interacción entre el cliente y el sistema. El cliente inicia el proceso mediante un script que envía la información al gateway y queda a la espera de una respuesta. El gateway, por su parte, se encarga de parsear los datos recibidos y encolarlos para que los workers los procesen. Una vez que ha finalizado la recepción y el encolado de la información, el gateway comienza a desencolar los resultados procesados y los envía en conjunto al cliente.

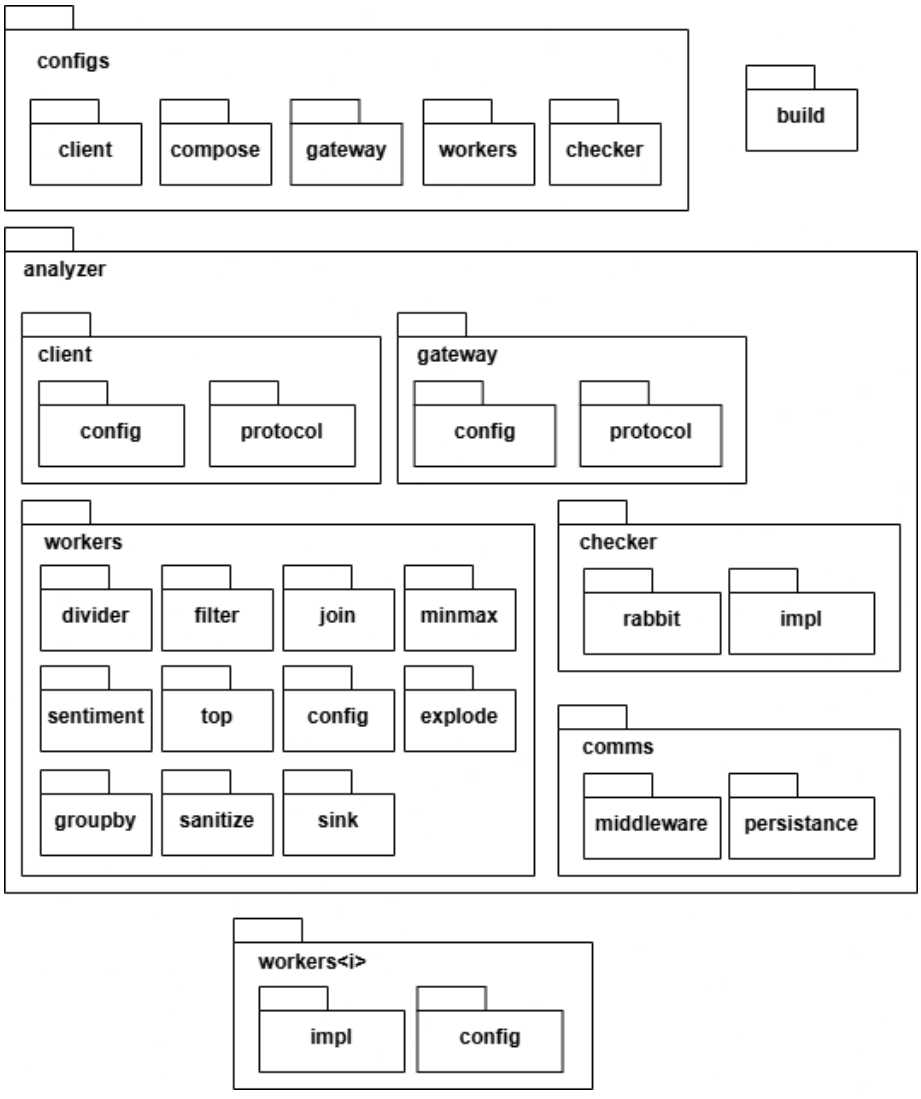

Client
Client
Gateway
DataParser
loop
./start_client.sh
run client
send datasets
parse data
parse
return parsed data
enqueue movie, rating or credit
return
moreDataAvailable()
loop
deque query result
return query result
writeToFile(query)
notifyUser(query)
moreDataAvailable()
return
return

Estructura y organización interna

Organizamos los directorios del proyecto de manera que cada componente tenga responsabilidades claramente separadas. Existe una carpeta general llamada configs, encargada de almacenar las variables de entorno utilizadas por cada worker.

Por su parte, el cliente cuenta con su propia configuración y protocolo para comunicarse con el gateway. El gateway, a su vez, posee su propia configuración, define el protocolo de comunicación con el cliente y contiene una carpeta adicional llamada protocol, responsable de manejar la comunicación con el cliente y los workers.

Cada worker comparte una configuración general común y, además, incluye una configuración específica junto con su implementación, representada en el paquete Worker<i>.

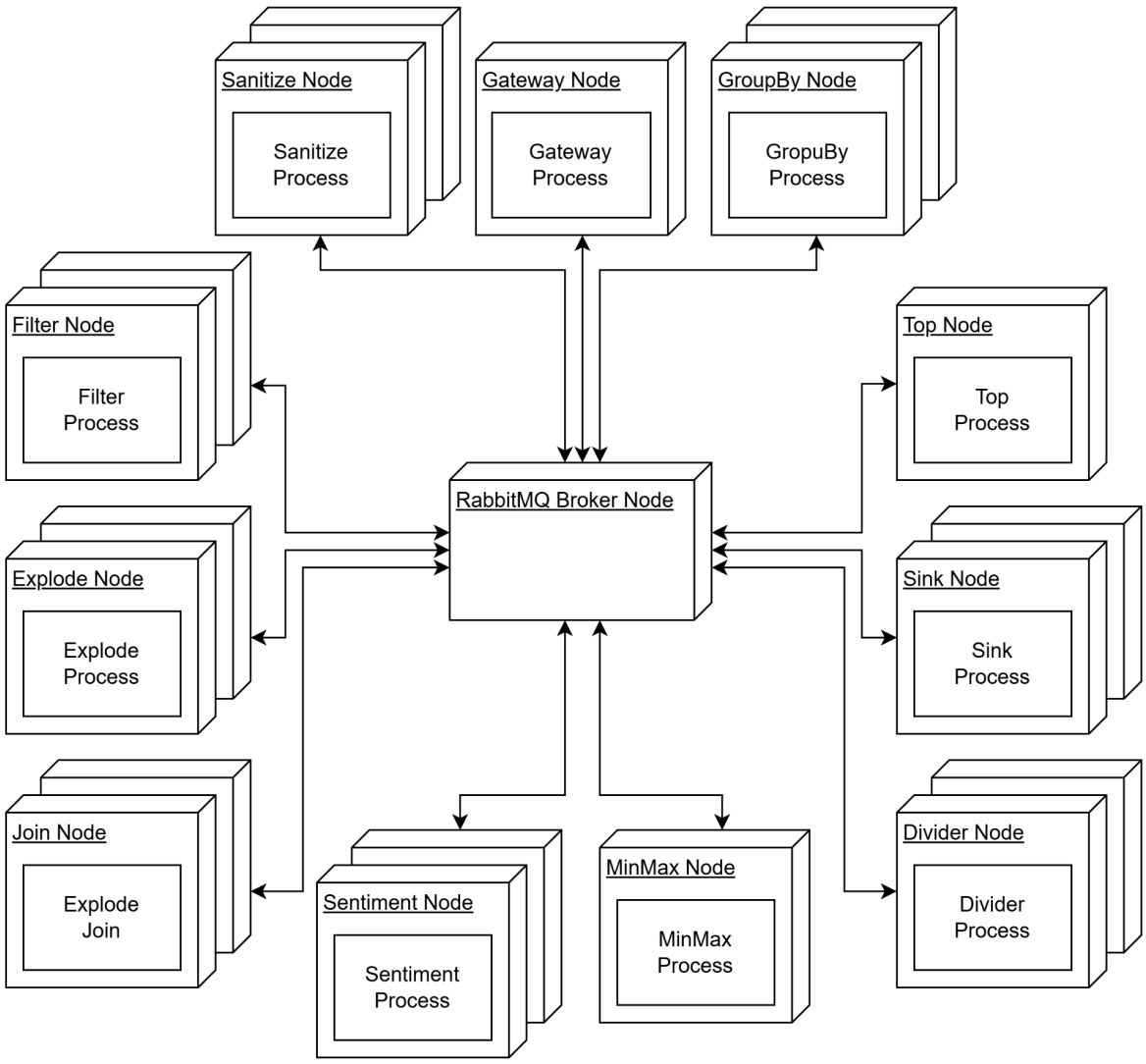


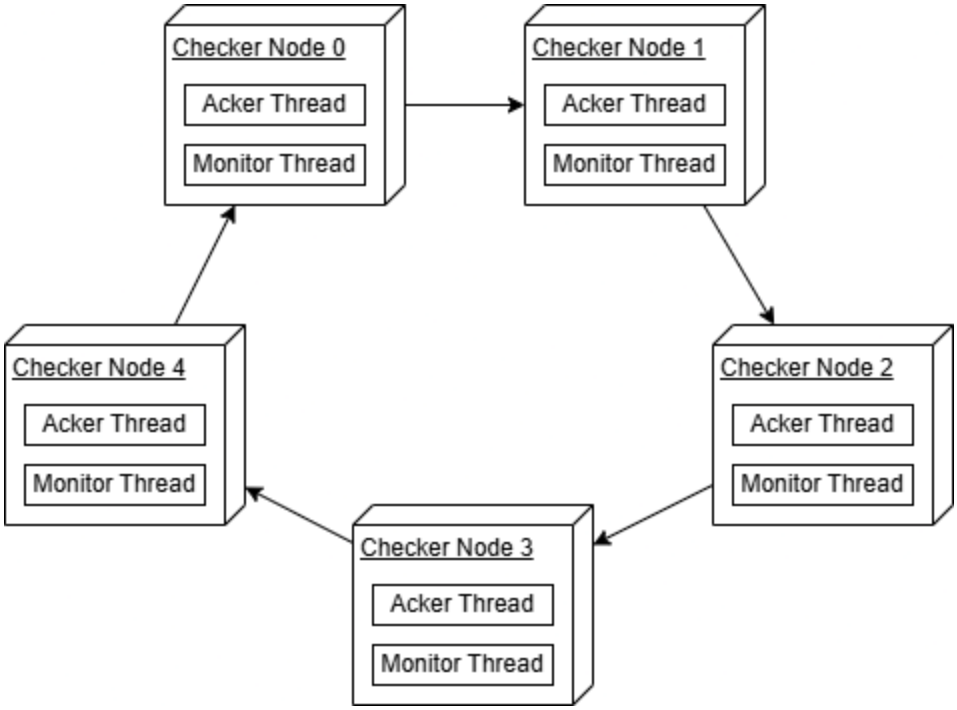
Despliegue del sistema

En este diagrama se puede observar la distribución de nodos y procesos dentro de ellos. Decidimos dividir los nodos por funcionalidad, donde hay un encargado por cada tipo de actividad y dentro del él existen procesos que resuelven alguna instancia particular del sistema, por ejemplo dentro de nodo de filtros existe un proceso encargado de filtrar por año en fechas mayor o iguales a 2000.

Todos los nodos realizan su comunicación mediante el nodo central de RabbitMQ que proporciona las colas de mensajes. Para la comunicación entre el cliente y el gateway creímos más cómodo tener una conexión directa.

En paralelo se despliegan los Checkers que se encargan de controlar la tolerancia a fallos de estos nodos. Estos son los únicos dentro del sistema que no usan colas de RabbitMQ, en cambio, se comunican mediante UDP para una comunicación más rápida.


Sanitize Node
Sanitize Process
Gateway Node
Gateway Process
GroupBy Node
GropuBy Process
Filter Node
Filter Process
Top Node
Top Process
RabbitMQ Broker Node
Explode Node
Explode Process
Sink Node
Sink Process
Join Node
Explode Join
Sentiment Node
Sentiment Process
MinMax Node
MinMax Process
Divider Node
Divider Process

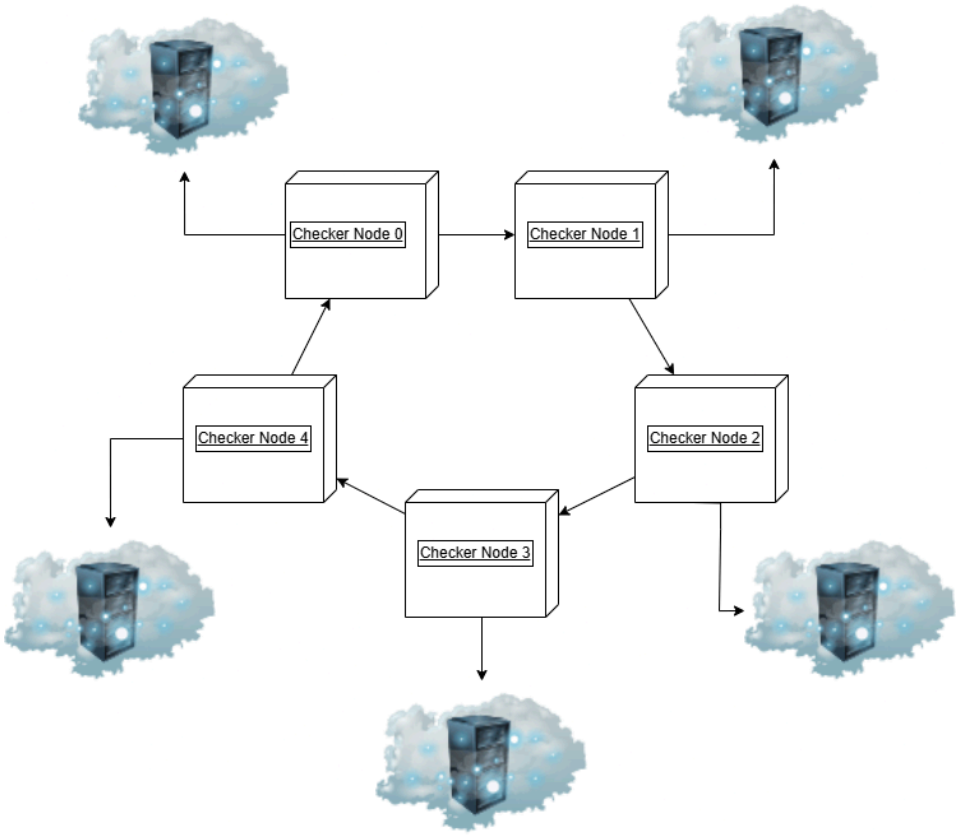


Tolerancia a fallos

Checkers

Con el objetivo de mantener un control constante sobre los nodos del sistema y detectar posibles caídas, se implementa un mecanismo denominado Checker. Cada Checker es responsable de monitorear un subconjunto **único** de nodos y, al mismo tiempo, debe ser supervisado por otro Checker, formando una topología de anillo. Esta estructura permite detectar fallos tanto en los nodos como en los propios Checkers.

Un nodo o Checker se considera “muerto” si no responde durante un período configurable de tiempo tras múltiples intentos (también configurables) de verificación. En ese caso, se procede a "rematarlo" para posteriormente ser reiniciado, permitiendo que retome sus tareas normalmente. También se considera como “muerto” si no se puede resolver la ip del nodo y se revive automáticamente sin mandar ningún check



Persistencia

Dado que los nodos pueden fallar, se implementa un mecanismo de persistencia para garantizar la integridad de los mensajes procesados. Antes de confirmar la recepción de un mensaje (ACK) a través de la cola de RabbitMQ, la información procesada se guarda en disco. Esto permite enfrentar dos escenarios críticos en caso de caída:

- Caída durante el procesamiento (antes de la persistencia):
    - Este es el caso más sencillo. Si el nodo se cae mientras procesa el mensaje, al reiniciarse volverá a recibir el mismo mensaje y podrá procesarlo nuevamente sin problemas.

- Caída después de la persistencia pero antes del ACK:
    - En este escenario, como el mensaje ya fue persistido junto con su número de secuencia y réplica, el sistema detectará que ya fue procesado cuando vuelva a recibirlo, y enviará el ACK automáticamente. Es importante destacar que el sistema puede generar mensajes duplicados, pero siempre cuenta con un mecanismo de de-duplicación para evitar re-procesamientos innecesarios.

Seguimiento de un Mensaje

Receive

La recepción de los mensajes es manejada por la entidad de Receiver, es quien ordena los mensajes recibidos y hace conteo de algunos tipos de mensajes para no reaccionar ciertos mecanismos, por ejemplo cuenta la cantidad de mensajes de EOF para garantizar que a cada nodo siempre le llegue una sola vez en vez de n veces, siendo n la cantidad de réplicas superiores.

Gracias al ordenamiento de mensajes pudimos dejar seteada la opción de Qos de Rabbit a un número mayor a 1, cosa que acelera el procesamiento global del sistema. Si no está seteada la opción, rabbit por defecto envía todos los mensajes posibles a los nodos y en caso de que el nodo caiga no garantiza que el reenvío de estos mensajes tenga el orden original, por lo que decidimos que exista un intermediario entre el Worker y la cola de rabbit que ordene estos mensajes.

En caso de tener más de un receiver, siendo el único caso el Join, éste está preparado de forma que solo se procese un solo mensaje y que no exista la posibilidad de que se modifiquen datos por ninguno de los receivers mientras haya un mensaje siendo procesado sea de la cola que sea. Esto es vital para la parte de la escritura a disco de los datos del receiver, que es quien detecta mayormente los duplicados y desecha esos mensajes.

Process

Dependiendo del tipo de nodo y tipo de mensaje puede o no modificar estado interno y/o ser enviado. Los nodos stateful acumulan estado y llegado el EOF envían el resultado de todos esos baches de datos acumulados a los nodos siguientes.

Esta etapa de procesamiento es muy particular a cada tipo de nodo, es de hecho lo que diferencia a cada uno. La forma de guardar y recuperar datos en nodos stateful es mediante el uso de Persistors.

Decidimos que los nodos stateful hagan lecturas y escrituras a disco en esta etapa. Toda información que deba ser persistida a nivel de Worker se hace acá y no luego del envío. Para esto tuvimos que idear un sistema de guardado idempotente cosa que si quisiéramos guardar algo podamos detectar datos duplicados.

Este sistema es el que implementan los Persistors. Van a guardar de forma atómica datos en disco dados el nombre del archivo, los datos y el id del paquete. Dentro del id del paquete se encuentran datos como, de que cliente son los datos, el id de réplica superior que los envió y el número de secuencia de esa réplica para ese cliente.

Cada archivo lleva consigo un header de información donde están los últimos números de secuencia de cada réplica superior que modificó el archivo, por ejemplo, si un paquete del cliente 0 que envió la réplica 1 con número de secuencia 1337 pide escribir un cierto archivo, el Persistor va a chequear que en su header el número de secuencia correspondiente a la réplica 1 sea menor a 1337. Si encontrara un número igual o mayor, significa que el mensaje está duplicado, por lo que no modificaría el archivo y avisa mediante un valor de retorno.

Send

Esta etapa quedó dentro del Process, porque dependiendo del tipo de mensaje podía o no enviar datos al siguiente y nos pareció que lo más fácil era incluirla dentro del procesamiento y que cada tipo de nodo supiera si hay o no que enviar un paquete a los nodos siguientes.

De esta etapa se encarga la entidad de Sender, existen 2 tipos de sender, estos son el Robin Sender y el Shard Sender. Uno se encarga de enviar a las réplicas siguientes de a uno por vez, siguiendo una metodología de Round-Robin, mientras que la otra va a desarmar cada bache, a shardear sus contenidos y luego potencialmente enviar más de un mensaje a varios nodos en caso de que el nodo de más abajo esté replicado.

Generalmente se utiliza el Robin, el Shard es particularmente usado por nodos que anteceden a los nodos Join y GroupBy.

Dump

Esta etapa se encarga de bajar a disco los datos del middleware, esto incluye a los receivers y a los senders. La entidad que los engloba se llama Mailer, expone esta funcionalidad vía un método que dado un cliente escribe en disco en un mismo archivo el estado actual de ambas entidades. Esto es para que los números de secuencia entrantes no diverjan de los salientes, esto podría ocasionar que un mensaje se pierda o un número de secuencia nunca llegue a un nodo de más abajo.

Ack

Nosotros utilizamos el Ack no solo para avisar a Rabbit que no envíe este mensaje otra vez, sino también para avisar a los receivers que es hora de entregar el próximo mensaje recibido. Cada mensaje que llega es envuelto en una estructura que lleva consigo un mutex compartido entre todos los receivers, cosa que siempre se procesen de a uno a la vez. El Ack libera el mutex y deja que los receivers modifiquen su estado interno pasando consigo el próximo mensaje al worker.

Posibles mejoras

Si bien el sistema funciona y cumple con lo pedido, el mismo podría ser optimizado al menos de las siguientes formas.

- Dumpear y Ackear cada n mensajes
  - En vez de hacer el dump y el ACK cada mensaje que llega, se podría hacerlo de a batches lo que ocasiona un mayor flujo de procesamiento sin guardado en disco y constante espera del mutex
- Hacer que los mensajes de los workers estén acotados por tamaño
  - En vez de que los mensajes sean dinámicos y tener que pasar el largo de cada mensaje para saber cuánto leer, se podría fijar a un tamaño único para saber siempre cuánta data esperar.
- Roulette
  - Es muy difícil encontrar una probabilidad lo suficientemente grande para que tire algunos nodos y lo suficientemente chica para que se procesen los datos en un tiempo razonable.

Distribución de tareas

Tarea	Responsable
Gateway	Todos
Protocolo	Todos
Cliente	Todos
Filter	Ghosn
Explode	Bianchi
Join	Todos
Group By	Bianchi
Top	Gentilini

Sink	Ghosn
Divider	Gentilini
MinMax	Gentilini
Sentiment	Bianchi
Multicliente	Todos
Checkers	Todos