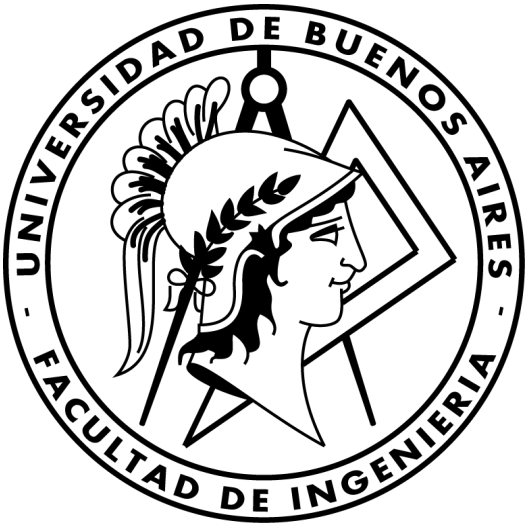
FACULTAD DE INGENIERÍA,
UNIVERSIDAD DE BUENOS AIRES

2024



SISTEMAS DISTRIBUIDOS

Informe TP2 - Tolerancia a fallas

Padrón	Alumno	Correo electrónico
108000	Juan Pablo Aschieri	jaschieri@fi.uba.ar
107870	Ugarte, Ricardo Martín	mugarte@fi.uba.ar

Cátedra: PABLO ROCA

Corrector: NICOLAS EZEQUIEL ZULAICA RIVERA

Índice

Introducción

El objetivo del siguiente documento es presentar un diseño de arquitectura para llevar a cabo un sistema distribuido. El mismo analiza reseñas de libros del sitio de Amazon y resuelve ciertas consultas necesarias para las campañas de marketing.

Para ello, se utilizarán diferentes vistas, cada una de las cuales ilustra un aspecto en particular del software desarrollado. El sistema estará optimizado para entornos multicomputadoras, así como soportará el incremento de los elementos de cómputo para escalar los volúmenes de información a procesar.

Además, se permitirá el procesamiento de múltiples clientes al mismo tiempo, mostrando alta disponibilidad, y se considerarán tolerancia a fallas. Es decir, se podrán soportar caídas de nodos del sistema en todo momento a través de un mecanismo de levantamiento de procesos y persistencia de la información.

Arquitectura de software

El cliente simulará ser un web scraper que va tomando libros y reseñas en tiempo real de la página de Amazon. Para ello irá leyendo un dataset de a batches y se los irá comunicando a los workers. Cada worker es responsable de realizar una operación a los datos, ya sea de filtrado, transformación o distribución. Luego envía sus resultados, ya sea a otro worker, para que continúe con más procesamiento, o al cliente para que éste lo almacene.

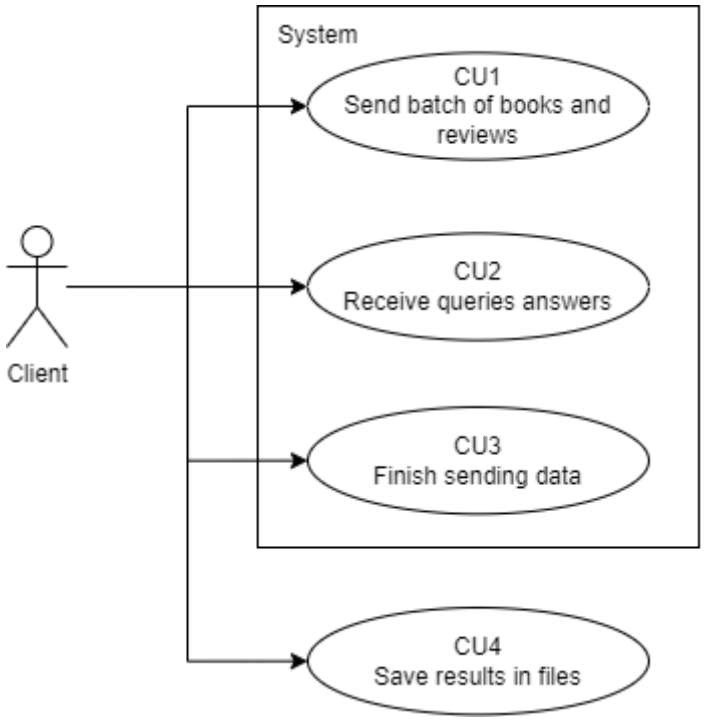
Todas las comunicaciones dentro del sistema realizadas mediante un middleware. Esto abstrae la necesidad de comunicarse mediante sockets y simplifica ambos extremos de la comunicación, permitiéndonos utilizar patrones como publisher-subscriber o producer-consumer de manera sencilla. Cabe aclarar, que el cliente y los workers no enviaran libros o reseñas de manera individual, si no que enviaran de a batches de libros y reseñas. Esto es dado que de lo contrario se estarían mandando muy pocos datos por envío, haciendo que los programas sean más ineficientes por la necesidad de pasar de kernel space a user space más veces.

Se proponen las siguientes vistas

- **Vista de escenarios:** se listan casos de uso que representen las funcionalidades centrales del sistema.

- **Vista lógica**: describe las partes significativas del modelo y qué estructuras se encuentran en él, haciendo uso de un diagrama de clases
- **Vista de procesos**: describe la descomposición del sistema en threads y procesos y cómo estos se comunican o interactúan entre sí, utilizando un diagrama de actividades.
- **Vista de desarrollo**: describe el mapeo de los paquetes en los diferentes componentes que los implementan a través de un diagrama de paquetes
- **Vista física:** describe la distribución física del sistema y cómo están relacionados sus nodos con la ayuda de un diagrama de robustez.

Vista de Escenarios



Este es un diagrama de casos de uso, identificamos al cliente como único actor capaz de realizar distintas acciones con el objetivo de resolver las siguientes consultas

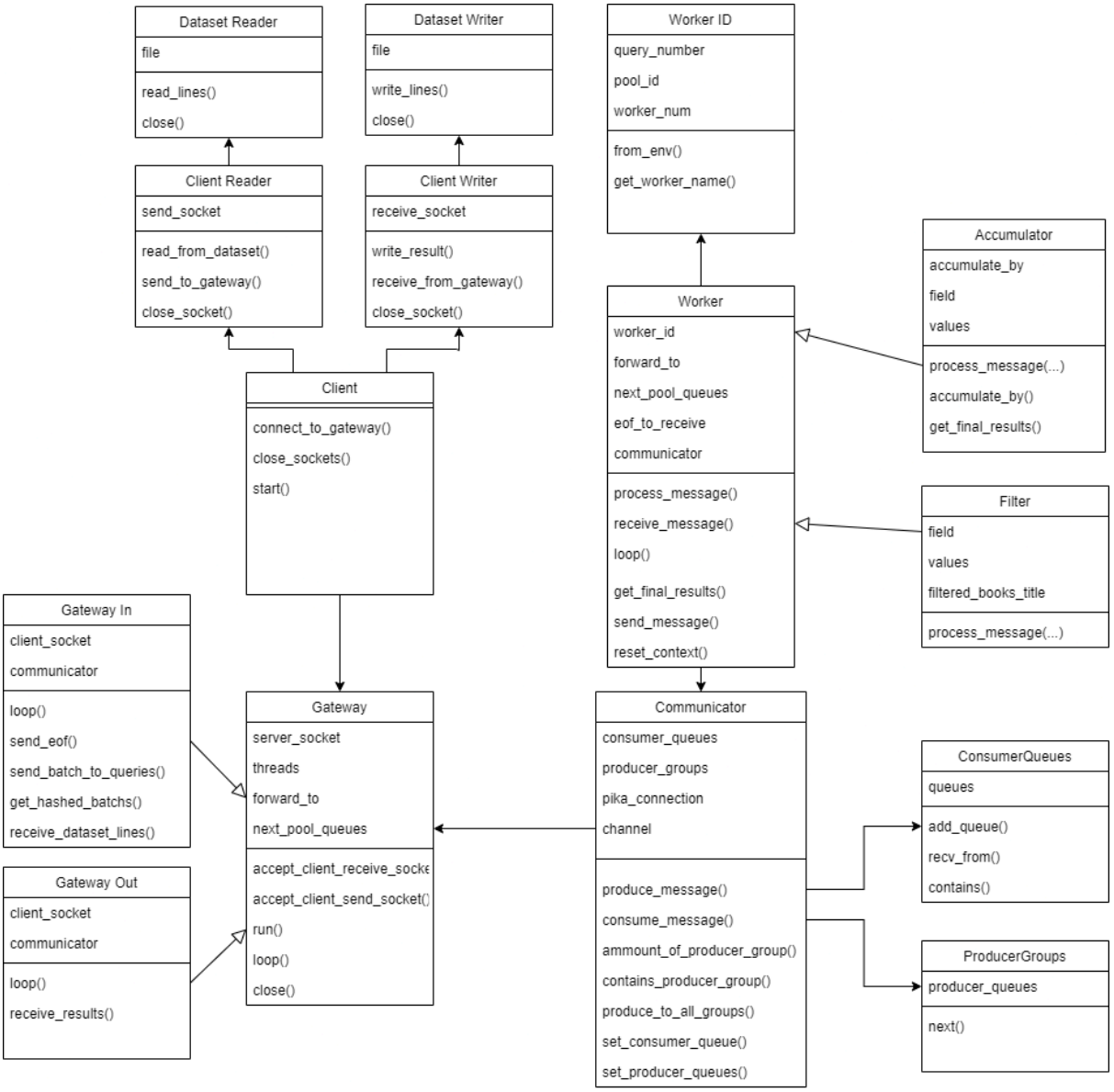
1. Título, autores y editoriales de los libros de categoría “Computers” entre 2000 y 2023 que contengan ‘distributed en su título
2. Autores con títulos publicados en al menos 10 décadas distintas
3. Títulos y autores de libros publicados en los 90' con al menos 500 reseñas.
4. 10 libros con mejor rating promedio entre aquellos publicados en los 90’ con al menos 500 reseñas.

5. Títulos en categoría "Fiction" cuyo sentimiento de reseña promedio esté en el percentil 90 más alto.

Por un lado el cliente realiza actividades independientes del sistema, descargar el dataset y almacenar los resultados en su memoria interna. Por otro lado, el cliente utilizará el sistema para realizar 3 acciones:

- Enviar los datos de libros y reseñas de Amazon para el posterior procesamiento de las queries.
- Recibir los resultados que el sistema va calculando.
- Comunicar al sistema que no hay más datos a enviar, para que este calcule los resultados finales.

Vista Lógica

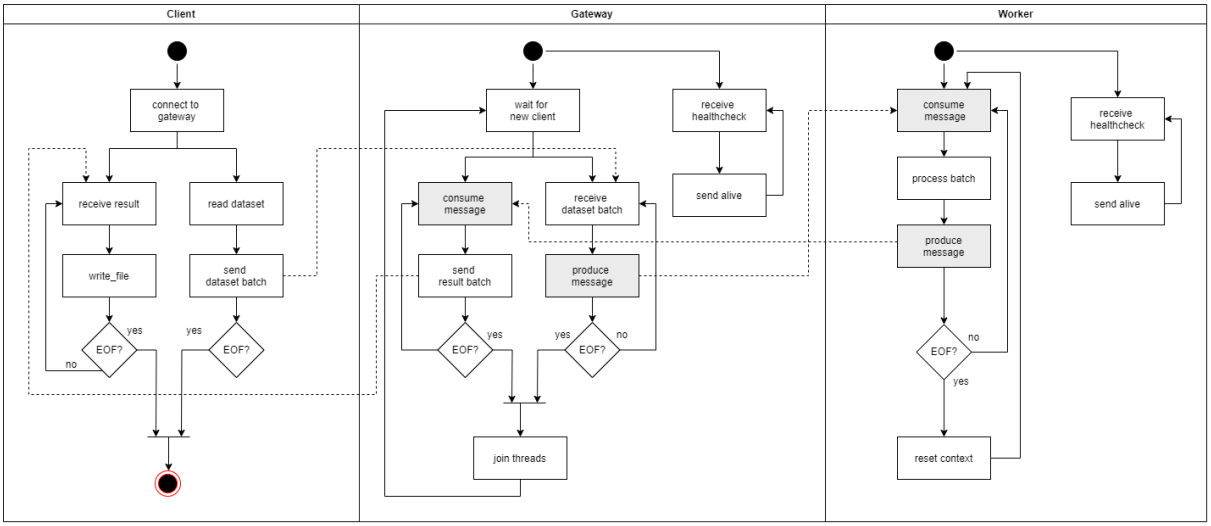


Este es el diagrama de clases y muestra las entidades que participan en el sistema. Entre las importantes se encuentran:

- **Client**: será quien actúe como el web scraper e irá transmitiendo libros y reseñas provenientes del dataset descargado del servidor de Kaggle. El cliente, usando dos threads distintos, le irá transmitiendo estos objetos al gateway y recibiendo los resultados de las consultas mediante sockets
- **Gateway:** Solo habrá uno en el sistema, y funcionará como punto de entrada y salida del sistema. En un thread, el gateway recibirá del cliente los libros y reseñas, y luego los enviará a las cadenas de workers para su procesamiento. También es capaz mediante otro thread de recibir los resultados de las consultas para enviarselos a través de sockets al cliente.
- **Communicator:** Es el middleware para llevar a cabo la comunicación entre procesos del sistema. Utilizando las interfaces que se proveen, dos procesos que se quieran comunicar podrán hacerlo de manera mucho más simple sin tener que recurrir a manejar sockets de manera directa. El middleware permite la comunicación utilizando una versión de productor-consumidor.
    - Los productores al producir indican, además del mensaje a enviar, el id del grupo al que quieren producir. Opcionalmente se puede indicar a qué miembro puntual de ese grupo se quiere enviar. Si no se indica el worker específico, se hará round robin para determinar el próximo worker a recibir el mensaje dentro del pool. Si se indica, entonces se le enviará el mensaje a ese worker particular, lo cual es útil para cuando se quiera acumular valores y que todas las claves vayan a parar al mismo worker, ya que se pueden hashear los campos por los que se quiere acumular, y así obtener en múltiples procesos distintos los mismos workers.
    - Los consumidores para consumir simplemente indican el nombre de la cola de la cual quieren recibir mensajes y posteriormente el middleware hace un ACK del mismo.
- **Worker**: Cada uno de los workers tiene su propia cola y es responsable del procesamiento de los datos recibidos por la misma, y el posterior envío del resultado, ya sea devuelta al gateway o a otros worker que tomarán su resultado como datos. Un worker siempre sabrá, por variables de entorno, a quien o quienes tiene que forwardear el mensaje y también cuántos workers tiene la próxima pool . Una cadena de workers será capaz de resolver las queries. Hay 2 tipos de workers:

- **Filter:** Se encarga de filtrar los datos recibidos en función de una condición en particular.
- **Accumulator:** Recibe datos, les aplica una función y los almacena. Una vez se cumple una condición, como puede ser cantidad de datos específicos almacenados, se envían los datos. Por ejemplo, un contador sería un acumulador simple. Adicionalmente tras recibir un eof, se pueden devolver los resultados que requieren del procesamiento de todos los datos.

Vista de Procesos



Este es el diagrama de actividades que ilustra el flujo de cada proceso corriendo en el sistema.

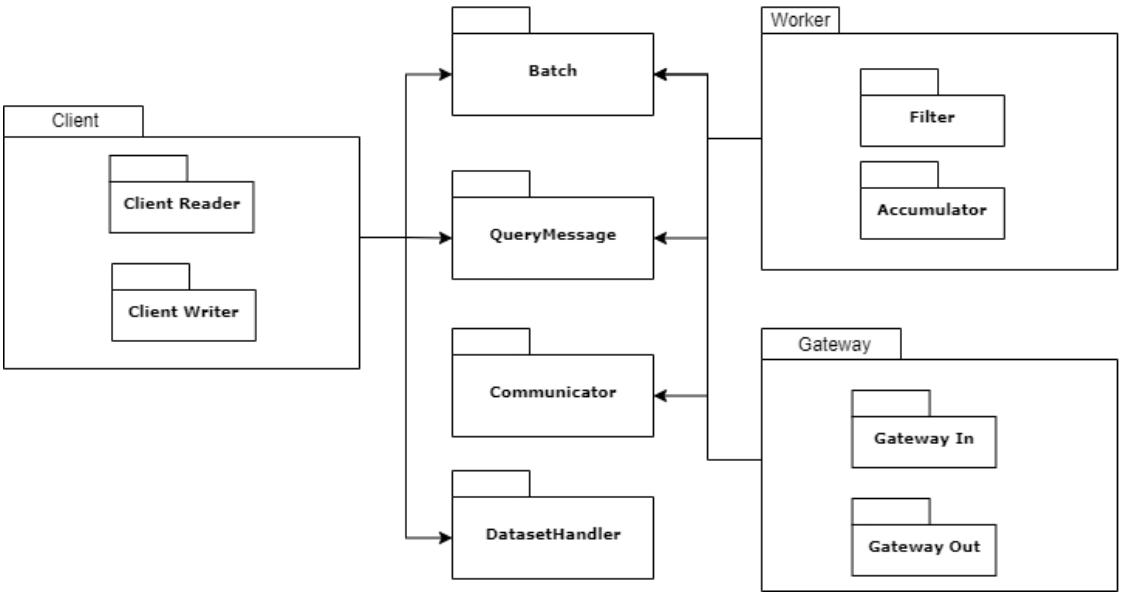
Por un lado, el cliente tendrá dos hilos de ejecución, donde uno se encarga de extraer información del dataset mientras que no se llegue al final del archivo, y luego enviar las filas de a batches de tamaño fijo al gateway. El otro thread lo que hará es quedarse escuchando nuevos resultados y según el número de query adjuntado, escribirlo en uno de los cinco archivos existentes.

El gateway tendrá por una parte un thread para recibir healthchecks y responderlos, y por otra parte como se explicó anteriormente, espera la conexión entrante de un nuevo cliente. Una vez establecido creará un threads para escuchar lo que el cliente le comunique con un socket y otro para que mediante el middleware, interactúe con los workers de la primera pool de cada query para enviarles los libros o reseñas y tras ser procesados recibir una

respuesta. Cuando se detectan EOFs, se joinean los threads y se vuelve a comenzar del principio esperando un nuevo cliente.

Luego, aparecen los workers donde se tiene un thread con el mismo uso que se le da en el gateway para el manejo de healthchecks, y usa la clase madre descrita en la vista lógica para indicar un flujo genérico de los mismos. Cada vez que reciben un batch, lo procesan y luego envían el resultado al siguiente worker mediante el middleware, o al gateway en caso de tratarse de un worker de la última pool de una query. Hay que tener en cuenta que el end of file del dataset se va propagando en los diferentes procesos hasta llegar aquí, siendo útil para terminar y que el worker reinicie sus valores y limpie su contexto para seguir escuchando para llevar a cabo futuros procedimientos. Para que un worker de por finalizado el procesamiento de un cliente, el worker debe recibir un EOF por cada worker en las pools de las que recibe mensajes. De lo contrario, si diera por finalizada la ejecución tras recibir un único EOF, podrían haber datos que no se procesen, ya que otros workers de la pool anterior podrían tener más datos a procesar.

Vista de Desarrollo



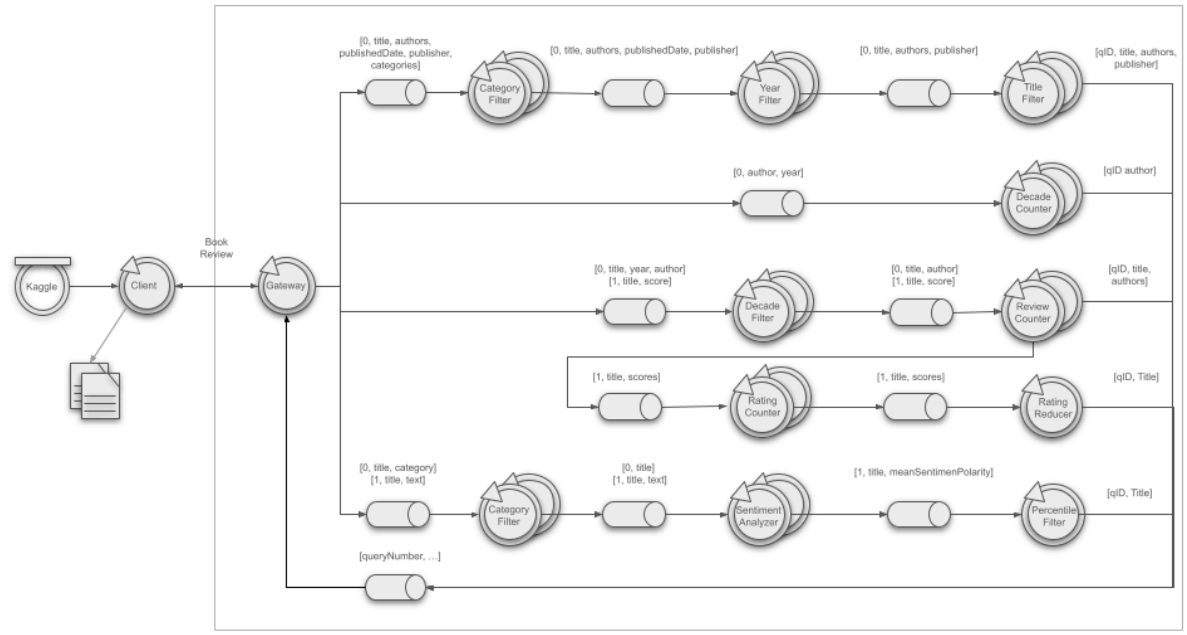
Este es el diagrama de paquetes y muestra cómo el cliente hará uso de las funcionalidades brindadas por el DatasetHandler para la lectura del dataset, así como también del paquete mensajes para llevar a cabo la serialización y deserialización de los mensajes transmitidos por socket con el gateway.

Asimismo, el worker tendrá una implementación genérica pero además traerá consigo 2 tipos de worker como se ha descrito anteriormente, pudiendo ser un filtro o un acumulador. Ambos serán capaces de distribuir por título llamando a una función de hash.

Como nuevo agregado, elaboramos la clase KeyValueStorage …. hablar de eso

Vista Física

Diagrama de robustez



Este es el diagrama de robustez que expone el sistema distribuido entero. Todas las comunicaciones que se realizan entre los distintos componentes del sistema será realizada a través del middleware, excepto por el cliente y el gateway que usan sockets. Si se trata de un mensaje que contiene un libro, el header lo indicará con un “0”. Si es una reseña con un “1”, y si es un resultado lo indicará con el número de query correspondiente, para que al final del circuito el cliente sepa en qué archivo guardarlo

El gateway utilizará el middleware para mandar los libros y las reseñas a cada uno de los subsistemas distribuidos que se encargan de procesar las queries, donde cada worker de entrada tendrá una cola con el nombre que lo representa (cabe destacar que en el gráfico

aparece una cola por cada comunicación en todo momento, pero en realidad son múltiples colas, una para cada worker).

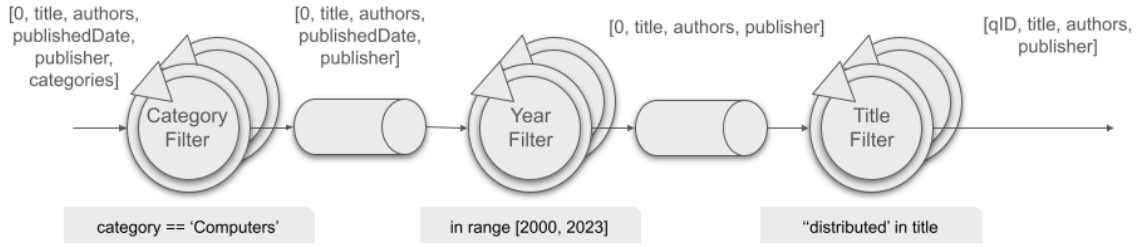
El gateway, enviará a cada query, un mensaje QueryMessage, que estará formado por los mínimos campos necesarios que necesite cada query. En el sistema hay algunos pools de workers, que necesitan acumular todos los mensajes que tenga un valor en específico en un campo, como es el caso de los contadores de reseñas por libro. Para lograr esto se enviará a el worker cuyo id corresponda al resultado de hashear el valor.
Por ejemplo, si el gateway recibe un libro con el título *“Harry Potter: La Cámara Secreta”*, iterará por cada query y aplicará la función de hash para saber a que worker de la pool correspondiente enviarselo. Para la query 1, en caso de devolver 0, querrá decir que deberá enviarle al worker “1.0.0”: el primero de la primer pool de la query 1, que está representada en el primer dígito.

Cabe destacar que a pesar de que no todos los pools necesitan esta distribución, donde tienen todos los mensajes de un campo, el sistema utiliza este para todas las comunicaciones. Esto se debe a que en un gran volumen de datos, si asumimos que el resultado de hashear algo como un título o el nombre de un autor (con SHA256), resulta en un worker aleatorio, podremos decir que se distribuyen equitativamente los datos a procesar en el próximo pool de workers.

Cada query subsystem irá procesando los libros y reseñas y de ser posible irá devolviendo resultados periódicamente por el middleware, indicando a qué query pertenece el resultado. De no poder dar resultados parciales, los subsistema de queries enviaran el resultado una vez este esté disponible, ya sea tras acumular suficientes datos o una vez el cliente envíe una señal de fin del archivo, EOF, indicando que no se enviaran más datos a procesar.

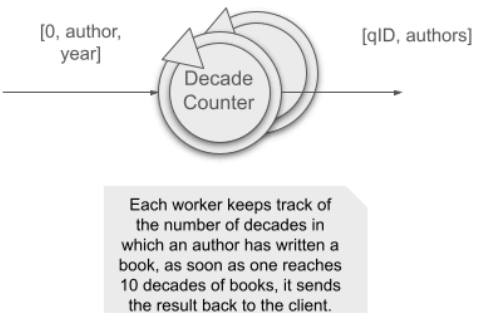
A continuación se mostraran los subsistemas de cada query

Query 1 subsystem



El sistema de la query 1, está compuesto por 3 pasos, primero un filtro por categoría, luego un filtro por año, y finalmente un filtro que chequea si 'distributed' está en el título del libro. Cabe destacar que todos estos filtros pueden ser paralelizados, y en vez de hablar de un único worker filtro, hablar de un pool de workers filtros. Esto es por la naturaleza de la query, ya que da lo mismo que libro sea agarrado por cualquier worker y no hay necesidad de coordinación, por lo que se puede paralelizar el trabajo sin ningún problema.

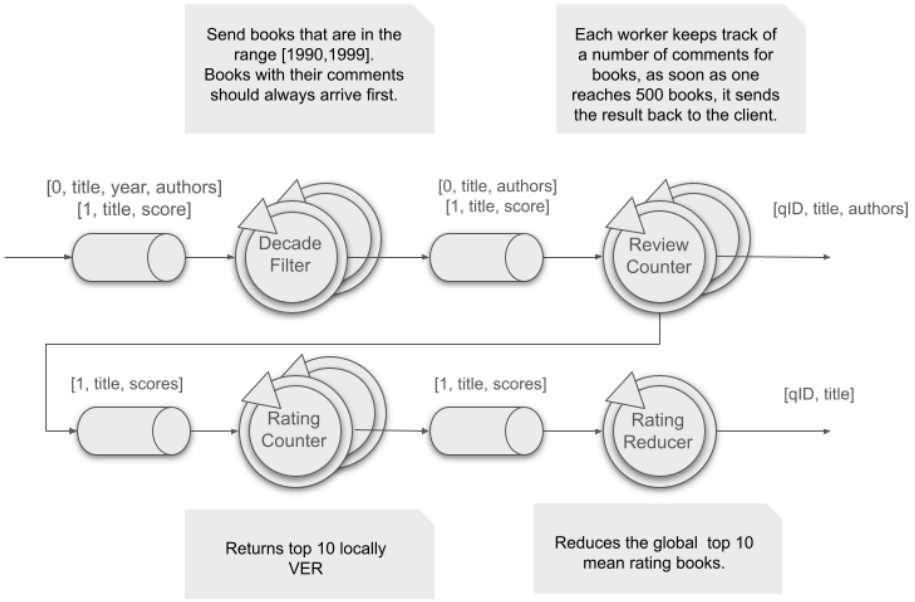
Query 2 subsystem



El sistema de queries 2 consta de contadores de décadas. Es necesario que se envíen los libros de manera distribuida por autor, es decir que a un worker, siempre le lleguen todos los libros del mismo autor. Los contadores entonces sabrán que tienen todos los libros de sus autores. Una vez reciban un libro, verificarán de qué década es, y una vez consigan 10 libros que hayan sido publicados en 10 décadas distintas enviaran el resultado al cliente.

Sin recibir los datos distribuidos por autor no sería posible paralelizar las tareas de los Decade Counters ya que no se podría garantizar que un worker tenga todos los libros de un autor, y por ende no podría verificar correctamente la condición.

Query 3 y 4 subsystem

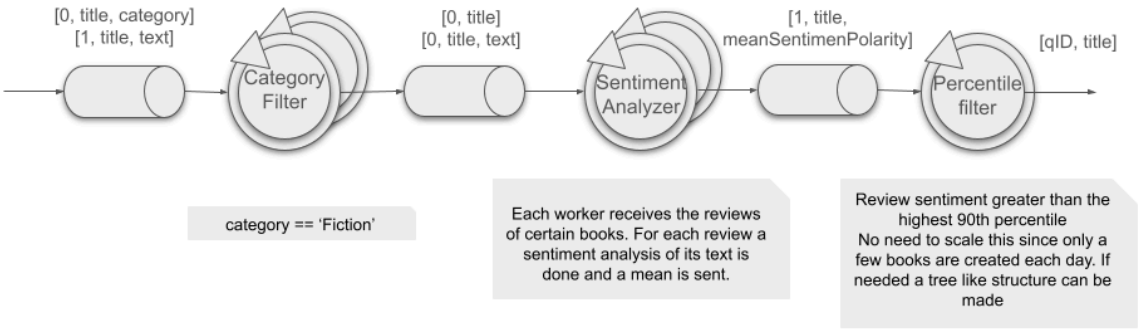


Similar al caso anterior, en el sistema de query 3 se deben recibir los libros distribuidos por título. La información luego pasará a un filtro de décadas para quedarse con los libros publicados en los años 90', que luego pasarán por un contador de reseñas que validará aquellos que tengan al menos 500 de ellas para mandarlas como resultado.

La query 4 al solicitar lo mismo que la query 3, solo que adicionalmente desea quedarse con el top 10 de ellos, reutiliza la query 3 y se alimenta con la salida del mismo, recibiendo el título y su respectivo atributo meanrating. Al reutilizar la salida de la query 3 nos ahorramos tener que realizar los mismos procesamientos múltiples veces.
Inicialmente la query3 solo guardaba la cantidad de reviews para un libro y una vez esta llegaba a 500, enviaba automáticamente el resultado. Para poder reutilizar la salida de la

query3, para la query 4, la query 3 también almacena la suma del rating de cada review, y luego envía el promedio de los rating. Para hacer esto la query 3 deberá esperar hasta haber recibido todos los datos para poder enviar los resultados.

Query 5 subsystem



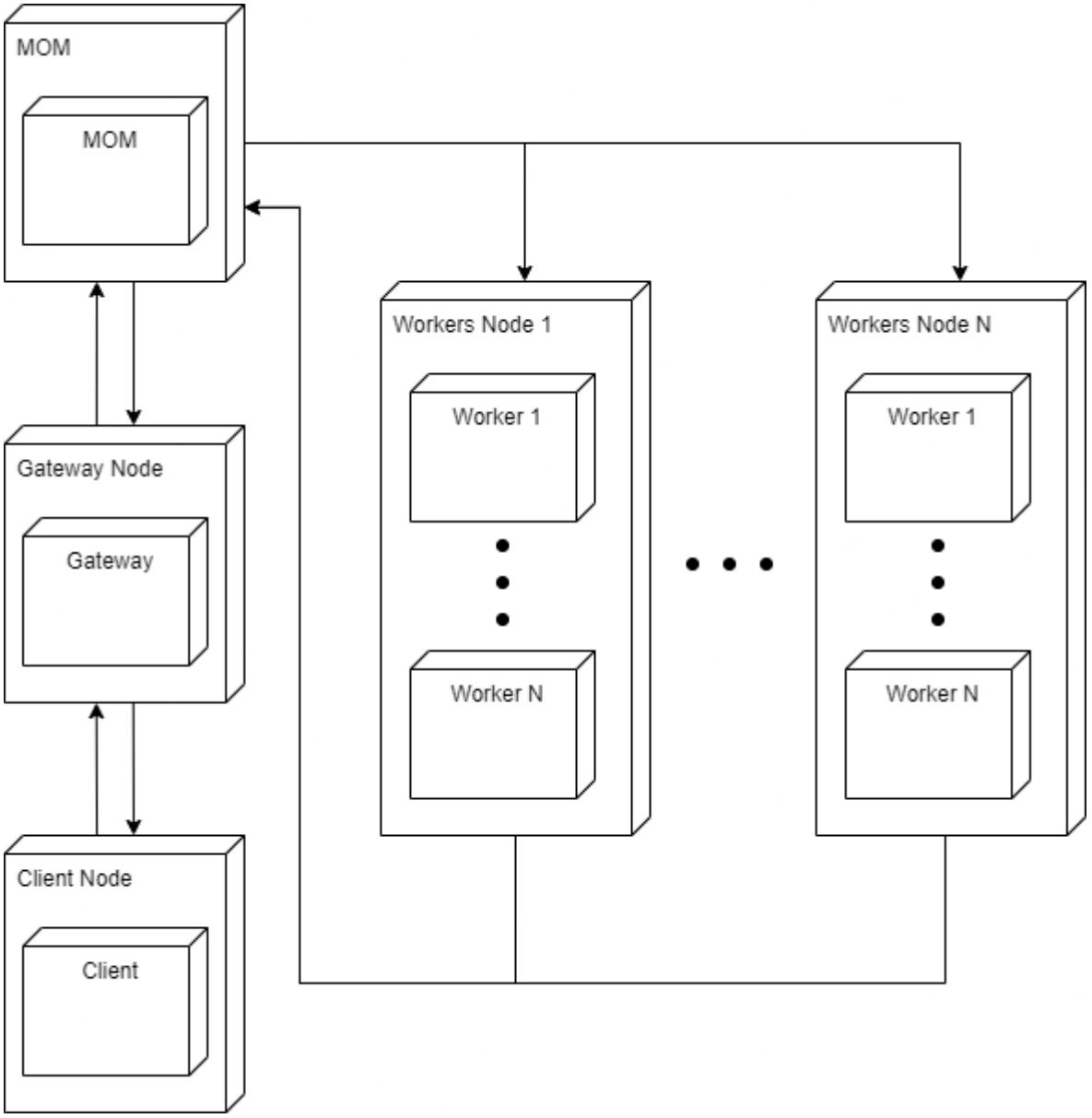
El sistema de queries 5 al igual que los anteriores, deberá recibir las queries distribuidas por título de libro. Una vez recibido los mensajes se aplica un filtro de categoría para quedarse con sólo aquellos que sean de ficción. Luego aparece un worker acumulador dedicado al análisis de sentimientos, el cual luego de realizar el análisis de las reseñas, larga como salida el título del libro y su polaridad de sentimiento promedio, en base a las reseñas asociadas al mismo. Finalmente el Percentile filter irá acumulando los promedios. Una vez recibidos todos los promedios de análisis de sentimiento de cada libro, se quedará con el 10% más alto y los enviará al cliente.

Esta query trae consigo algo que no pasaba en las anteriores queries. El último nodo es un único worker en vez de un pool de workers. Esto se debe a que si pusiéramos un pool de workers para quedarse con el 10% más alto, no conocerán los valores de los demás workers obligándolos a coordinarse entre sí para obtener el resultado final. Para solucionar la coordinación, la solución que se nos había ocurrido fue armar una estructura como árbol binario, donde dos workers se comunican entre si, juntan sus datos y mandan al próximo nivel los datos que podrían estar en el 10% más alto de todos los datos, y descartan el resto. En esta comparación de datos, si hay más de 20 workers no se podrá descartar nada, ya que la suma de los datos que ambos workers será menor igual al 10%, ya que cada uno

tiene el 5% de los datos o menos (asumiendo que fueron distribuidos equitativamente). Una vez queden menos de 20 workers se empezarán a descartar valores hasta llegar a la última comparación y obtener el 10% más alto.
Sin embargo, todo este sistema complejo no nos reduce mucho la cantidad de datos que va a tener el último worker. En vez de tener un O(n) donde n es la cantidad de datos que llegan al último nodo, tendremos O(n/10), pero en múltiples workers. Por lo que si la cantidad de datos fuera escalar mucho, el sistema no escalaría con ninguna de las dos soluciones, ni la simple con un solo worker, ni la compleja con un árbol de workers.

Ahora bien, ¿hace falta escalar esta parte del sistema?. La conclusión es que no. Esto es dado a la naturaleza de los datos. Al último nodo de procesamiento de la query 5 le llegará solo un valor por cada libro de ciencia ficción: si asumimos el peor caso donde todos los libros del dataset son de ciencia ficción, tendremos 300 MB de data a procesar, lo que no es mucho, por lo que un único worker lo podría procesar. Aún más, si asumimos que el sistema está andando en la vida real y recibe libros y reseñas a medida que estos son publicados, el último nodo no recibirá más de algunos libros por día. En definitiva, el sistema debe escalar principalmente para aquellos procedimientos que respecta a las queries que manejan reseñas ya que su volumen de datos es órdenes de magnitud más grande que los libros. Es por esto que podemos ahorrarnos la complejidad extra de armar un árbol de workers que se comuniquen entre sí para obtener el percentil 90 y simplemente procesar todo en uno.

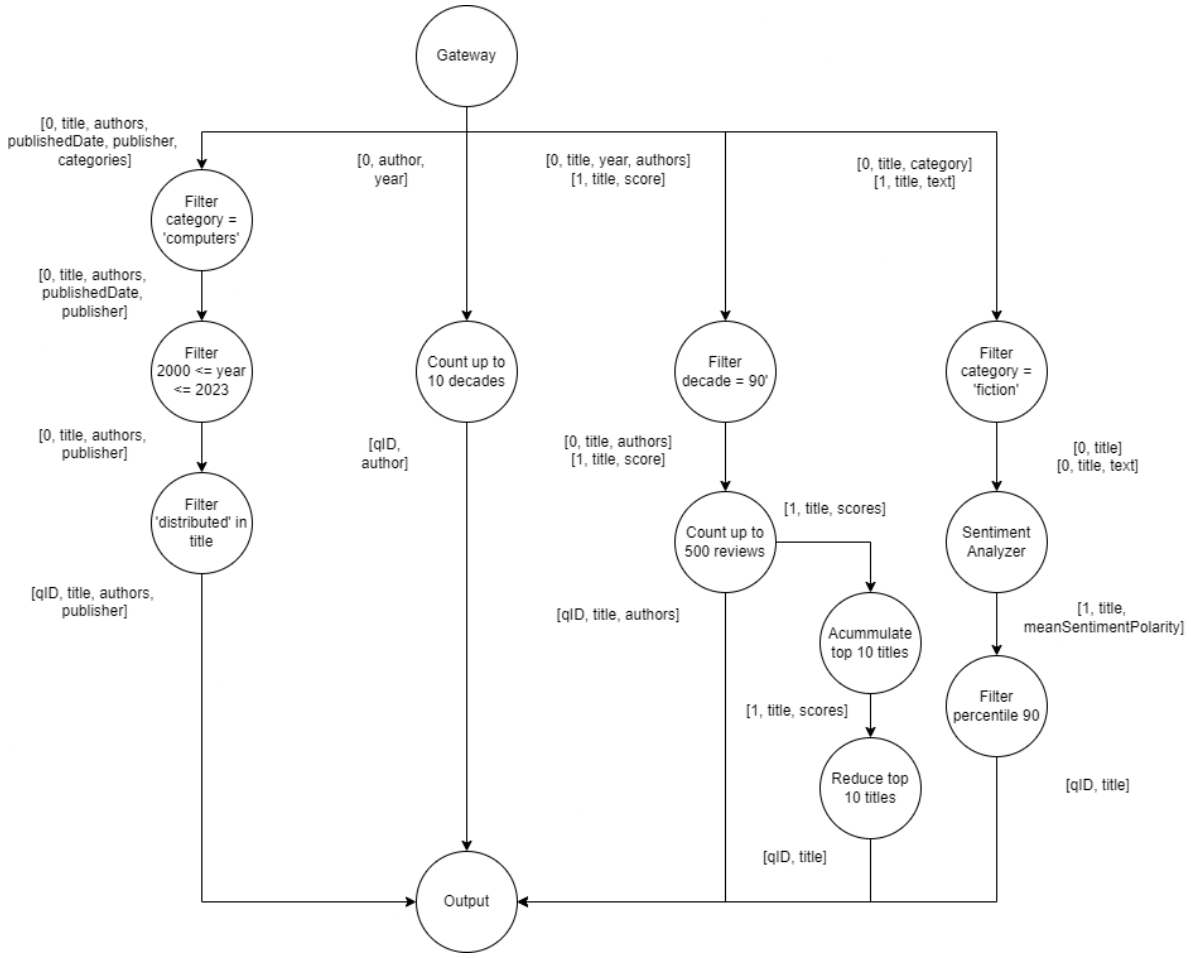
Diagrama de despliegue



Este es un diagrama de despliegue, nos muestra cómo se distribuyen los procesos del sistema en distintos componentes físicos. En este caso, tendremos 4 tipos de nodos de cómputo. Por un lado un único nodo donde se ejecutará el código del cliente, quien será quien envíe activamente los datos del sitio web. Luego tendremos un nodo que tenga al gateway, que será el que funcione como entrada al sistema y recibirá las peticiones del cliente. Como núcleo central de las comunicaciones del sistema habrá un MOM por el cual pasarán todas las comunicaciones (nosotros utilizaremos RabbitMQ). Finalmente, debido a la escalabilidad del sistema, se podrán agregar tantos nodos de workers cómo uno quiera. En cada uno de ellos, podrá ejecutar tambien n instancias de los mismos, es decir que en un mismo nodo de cómputo podríamos tener múltiples filtros y acumuladores. Cabe

destacar, que hay acumuladores que realizan análisis de sentimiento, un procesamiento pesado. Por lo que se querría que no hubiera más de un worker de análisis de sentimiento por, si no que estos estén distribuidos en múltiples nodos para aprovechar mejor el uso de CPU.

Gráfico Acíclico Dirigido



Por último, presentamos el DAG, donde se ve a partir del gateway el recorrido que realizan los datos a medida que pasan por los diferentes workers. Cada nodo dentro del gráfico, representa una operación a realizar sobre los datos, llevada a cabo por un pool de workers. De esta forma, vemos que hay 5 salidas, una por query, sin embargo hay solo 4 entradas. Esto se debe a que algunas queries deben realizar el mismo procesamiento a los datos, por lo que para evitar repeticiones, decidimos juntar este flujo en un único pool de workers.

Cada nodo únicamente enviará los datos que necesite el próximo proceso, y descartará aquellos que no sean necesarios. Si bien todos los datos en el gráfico llegan al output, estos

no llegarán al mismo tiempo, ya que algunas queries podrán dar el resultado inmediatamente, mientras que otras deberán acumular una cantidad de datos antes de poder continuar con el procesamiento y dar la respuesta.

Por último, trabajamos bajo la suposición de que siempre llegarán los mensajes de libros antes que sus reseñas. Esto es para evitar guardar temporalmente reseñas de libros que aún no han llegado para luego asignarlos.

Tolerancia a fallos

Introducción:

A continuación se explicará cómo se transformó el sistema previamente explicado en uno capaz de tolerar fallas como la caída o desconexión de uno o más nodos del mismo. Esto se logró persistiendo tanto información propia del sistema, como números de secuencia o ids, que de ahora en más llamaremos metadata, así como también información de los datos de los clientes que de ahora en más llamaremos contexto.

El sistema en su ejecución irá guardando esta información y al reingresar un nodo la usará para continuar la ejecución de tal forma que no se pierda ni altere la información del usuario, ni se generen posibles conflictos entre la información de los usuarios. Por supuesto que para que el sistema sea tolerante a fallos este debe ser capaz de darse cuenta que falló. Para esto se diseñó también un sistema de health-checks para reconocer y revivir procesos fallidos.

Hipótesis

Se tomarán las siguientes hipótesis

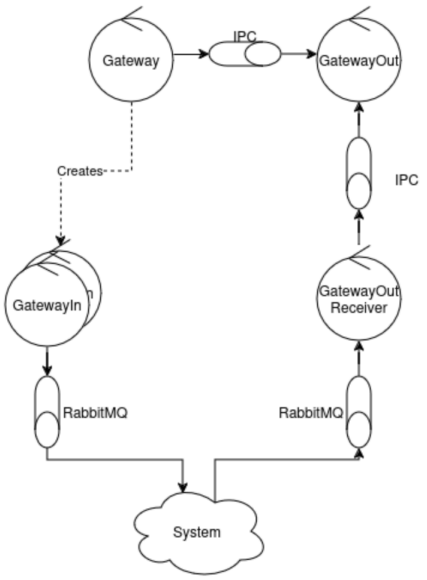
- No se corromperá el contenido del disco
- No habrá replicación
- El sistema no se responsabiliza por la caída de los clientes

Múltiples clientes

El sistema soporta múltiples clientes en paralelo. Para lograr esto al momento de conectarse con el gateway se utilizaran ids de clientes de 4 bytes. Al iniciar la conexión el

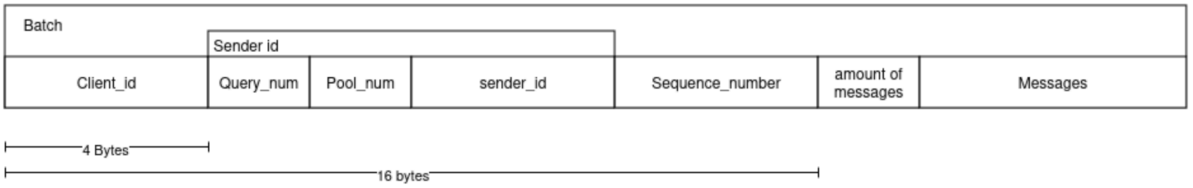
cliente envía un mensaje con id NO_CLIENT_ID=2**32-1, entonces el gateway asigna un id único al cliente y se lo envía como respuesta. Cada worker almacenará un contexto particular para cada cliente. Por ejemplo, si un nodo acumulador recibe mensajes de múltiples clientes, tendrá entonces para cada cliente un contexto donde irá acumulando lo que sea que necesite la query, como puede ser un contador de reseñas por libro. Para un fácil acceso en memoria se almacena como una tabla de hash donde la clave es el número de cliente y el valor son los valores acumulados.

El nodo de gateway fue modificado para manejar múltiples clientes en simultáneo.



Cuenta con un proceso Gateway que recibe las conexiones entrantes de los clientes. Cuando este recibe las dos conexiones TCP necesarias, el gateway envía una de las dos por un Pipe a otro proceso, GatewayOut, junto con el id del cliente. El Gateway Out es quien recibe los resultados del sistema y los reenvía al cliente correspondiente. Cuando el GatewayOut recibe un nuevo cliente y su socket, le confirma la recepción al Gateway, quien inicia un nuevo proceso GatewayIn quién será el responsable de recibir los mensajes de un cliente en particular e introducirlos al sistema para su procesamiento. Esta confirmación del Gateway Out se hace para garantizar que no se introduzcan mensajes al sistema hasta que el gateway out tenga el socket del cliente, y por ende sea capaz de procesar los mensajes salientes. El Gateway Out no se comunica directamente con el sistema, si no que hay un proceso más que actúa de intermediario quien es el que realmente realiza las comunicaciones con el sistema utilizando el mom. Entonces el GatewayOut cuando quiere recibir mensajes del sistema o realizar un ack se comunica con este proceso. Esto se hace para que el GatewayOut pueda recibir mensajes del sistema sin bloquearse, lo que será útil más adelante.

Antes de continuar es importante explicar el sistema de ids utilizado en el sistema, los cuales serán enviados en cada batch.



Se cuentan con 3 partes para el Id de un batch:

- **Client_id**: Es utilizado para identificar a qué cliente pertenece un batch. Todos los mensajes de un batch son de un mismo cliente.
- **Sender id**: Es una generalización de lo que antes llamábamos Worker Id. Consta de 3 partes, un número que indica la query, un número que indica de qué pool es, y un senderId que es el número de nodo dentro de la pool. Para el Gateway se utiliza query_id = 0
- **Sequence Number**: Es un número de batch para cada nodo del sistema que irá aumentando conforme el nodo envía mensajes, permitiendo identificar junto con el sender id unívocamente a cada batch dentro del sistema.

ACKS

El caso más básico a tener en cuenta, es que pasa si tenemos un nodo del sistema que deja de funcionar cuando tenía un batch de mensajes, pero aun no los había terminado de procesar. En ese caso, en el sistema original, el batch de mensajes se perdía ya que al usar el autoack de rabbit, el emisor del batch no tenía garantía que el receptor hubiera finalizado el procesamiento. En el nuevo sistema el nodo solo ackeara el batch de mensajes una vez sepa que este no tendrá riesgos de ser perdido. Al mismo tiempo se implementó un sistema de acks entre el cliente y el gateway ya que, a pesar de que tenemos la entrega de mensajes garantizada debido a que la comunicación se realiza mediante TCP, no hay garantía de que el mensaje llegue a ser procesado. El mecanismo de ack implementado es un simple mecanismo de stop and wait utilizando el sequence number de los batch. Un último ack es necesario en el sistema, los confirm de Rabbit. Esto se debe a que RabbitMq no proporciona RDT(reliable data transfer), por lo que utilizamos los publisher confirms, para obtener una garantía de que nuestro mensaje efectivamente se pudo encolar en el destino. Si en algún momento se llega a no recibir alguno de los distintos acks, simplemente se reenvía el mensaje con el mismo sequence number

Duplicados

El sistema de acks si bien soluciona el problema mencionado anteriormente introduce un nuevo problema. Qué pasa si un mensaje es enviado, recibido y procesado, pero el nodo que debe realizar el ack del batch se muere antes de poder hacerlo. En ese caso cuando se reintente enviar el mensaje este se duplicará. Esto puede generar que el sistema de resultados incorrectos debido a que la información del usuario puede llegar a ser procesada múltiples veces. Es aquí donde utilizamos el sequence number junto con el sender id para filtrar estos mensajes duplicados en el nodo receptor. Esencialmente si un batch tiene el mismo SenderID y mismo sequence number que otro recibido previamente, significa que ese mensaje es duplicado.

Ahora, se pueden duplicar todos los mensajes? No, solo el último. Esto es porque el emisor solo reenviará el último mensaje enviado y no volverá a enviar mensajes anteriores que ya sabe que fueron ackeados. Entonces, como en nuestro sistema cada nodo es el único consumidor de la cola de la que saca batches, si tuviéramos un solo nodo que emite a esa cola, podríamos decir que si dos batches contiguos tienen el mismo sequence number entonces es porque se duplicó el mensaje. Lo mismo aplica para cuando tenemos múltiples emisores, solo que en vez de decir que sean contiguos en la cola, comparamos los mensajes de cada emisor en particular. Es decir nos guardamos el último sequence number de cada emisor y cuando recibimos un batch de un emisor comparamos con el sequence number almacenado de ese emisor. Este mensaje duplicado si bien no es procesado debe ser ackeado.

Persistencia de datos

Ahora bien, si dos nodos del sistema se caen podría ocurrir que se duplique un mensaje y que el otro nodo no tenga correctamente actualizado el último mensaje recibido entonces se procesaría dos veces el mensaje. Pero aún peor, como la mayoría de nuestros nodos guardan un contexto del cliente para poder procesar futuros mensajes e ir calculando resultados, todos estos datos se perderían, nuevamente invalidando cualquier resultado que el sistema pudiera arrojar. Para que no se pierdan los datos procesados del cliente ni tampoco se pierda metadata, como el último sequence number de cada emisor, es que se persiste la información a disco.

Primero se planteó guardar la información a disco de a ráfagas. Es decir cada una cierta cantidad de batches se almacenarán los cambios en los contextos de los clientes y en la

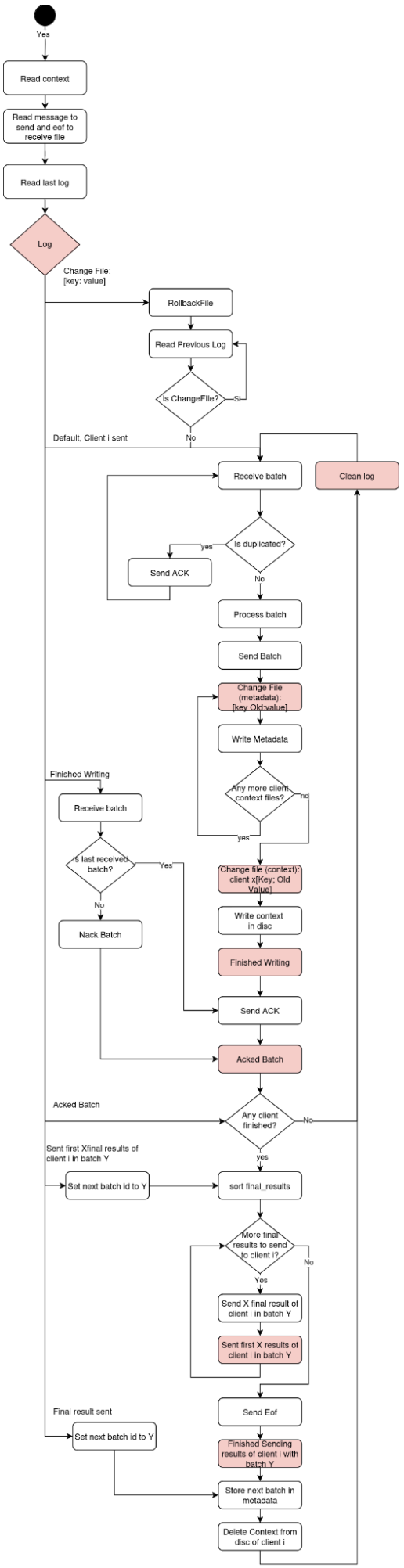
metadata. Pero se terminó optando por simplificar el sistema y bajar a disco de a un solo batch por vez. Para evitar que se realicen muchas bajadas a disco se aumentó la cantidad de mensajes que hay en un batch. Anteriormente se usaban batches de alrededor de 25 mensajes, mientras que ahora utilizamos valores como 512 o 1024.

Logging

Cuando decimos bajar a disco no es tan simple como llamar a una función write y listo. Esto se debe a que python no da ninguna garantía a la hora de hacer una escritura más que garantizar la atomicidad de modificaciones de un byte. Es decir, que nunca se escribirá un byte a medias, pero sí podría ocurrir que si se quieren escribir 5 se escriban los primeros 2 y no los últimos 3. A su vez, aun si python garantizara writes atómicos de una cantidad arbitraria de bytes no nos serviría, ya que contamos con muchos archivos que modificar y necesitamos que o se modifiquen todos o no se modifique ninguno. Para solucionar esto se utilizará un archivo extra para logear las operaciones y según el momento de la falla hacer un UNDO o un REDO, es decir deshacer los cambios a los archivos de contexto y metadata o reintentarlos. Adicionalmente al reiniciarse un nodo del sistema, este utilizará la información del log, para saber que proceso de inicialización debe realizar para incorporarse nuevamente, manteniendo válidos los datos.

Persistencia Worker

A continuación se presenta un diagrama de actividad sobre el funcionamiento del worker y como este realiza las bajadas a disco. Aquellos cuadrados de color rosado, identifican operaciones de logueo.


Yes
Read context
Read message to send and eof to receive file
Read last log
Log
Change File:
[key: value]
RollbackFile
Read Previous Log
Is ChangeFile?
Si
No
Default, Client i sent
Receive batch
Clean log
Is duplicated?
yes
Send ACK
No
Process batch
Send Batch
Change File (metadata): [key:Old:value]
Write Metadata
Any more client context files?
no
yes
Finished Writing
Receive batch
Is last received batch?
Yes
No
Nack Batch
Change file (context): client x[Key; Old Value]
Write context in disc
Finished Writing
Send ACK
Acked Batch
Acked Batch
Any client finished?
No
yes
Sent first Xfinal results of client i in batch Y
Set next batch id to Y
sort final_results
More final results to send to client i?
No
Yes
Send X final result of client i in batch Y
Sent first X results of client i in batch Y
Send Eof
Final result sent
Set next batch id to Y
Finished Sending results of client i with batch Y
Store next batch in metadata
Delete Context from disc of client i

Worker Loop

Previamente los Workers ejecutaban un loop donde recibían un batch de mensajes, lo procesaban y enviaban sus resultados parciales y si era necesario enviaban los resultados finales. Ahora a ese loop, se le agrega la persistencia de datos y loggeo de operaciones.

La información, tanto metadata como, información sobre el contexto del cliente, se persiste principalmente luego de realizar el procesamiento y envío de resultado parciales, si es que los hay, de un batch. Nuestros archivos son del tipo *KeyValueStorage* los cuales nos permiten acceder a datos a través de una clave (el KeyValueStorage será explicado en detalle más adelante). Para realizar la bajada a disco de un archivo se debe primero realizar un log del tipo *ChangeFile*, en el que se almacenará el nombre del archivo a modificar, las claves y sus viejos valores. Esto se debe a que en caso de no completar todas las bajadas a disco, se realizará una operación de UNDO haciendo un rollback de los archivos con los valores logueados.

Una vez bajado a disco todos los archivos se escribirá un log *FinishedWriting*, para indicar que todas las modificaciones de disco fueron realizadas correctamente. Esto entonces nos permite realizar el ack del batch, ya que tenemos garantía que aunque nos caigamos, no se perderá la información del batch.

Luego de ackear el batch se realizará un log *AckBatch* para indicar que el batch ya fue ackeado. Posteriormente si el cliente asociado al batch finaliza de enviar mensajes, el worker empezará a enviar sus resultados finales. Para esto el Worker primero realizará un ordenamiento de los resultados, no es importante el orden en el que estén, simplemente que este ordenamiento sea reproducible en múltiples ejecuciones. Luego empezará a enviar todos los resultados de a batches. Tras enviar un batch que contiene i resultados, se escribirá en disco un log del tipo *SentFirstFinalResults* que contiene la siguiente información: el id del cliente al que le corresponden los resultados finales, la cantidad de resultados finales enviados hasta el momento, y el número de batch del último batch enviado. Esta información se loguea para que ante una caída se pueda continuar el envío de resultados desde el mismo punto respetando los números de batch.

Tras enviar todos los resultados finales se escribe un log *FinishedSendingResults*, que contiene nuevamente el id de cliente y el número del último batch enviado. Luego el número del último batch enviado es persistido a disco, y se realiza un remove client. Que eliminará la información del cliente almacenada en el nodo tanto de RAM como de disco.

Tras finalizar el procesamiento de un batch y antes de recibir el próximo se realiza un clean del log que limpia todo el archivo de logs, ya que esos logs no nos serán más de utilidad.

Inicialización basada en log

Todo lo mencionado anteriormente es realizado para que ante la caída de un proceso este se pueda levantar teniendo toda su información válida. Cuando un proceso se levanta, comienza leyendo el contexto, la metadata y el último log registrado de su anterior corrida. En base a este log, el proceso será inicializado en cierta forma antes de entrar al código principal. A continuación se explican los distintos procesos de inicialización en función del ultimo Log:

- **ChangeFile(file_name, keys, old_values):** Si al iniciar nos encontramos con este log, significa que puede que no se haya completado el proceso de escritura en disco. Ya sea porque el archivo al que hace referencia este log no haya terminado de bajar todos sus datos o porque faltó escribir más información en otros archivos. En este caso realizará un UNDO, rollbackeando todos los cambios poniendo en los archivos los valores anteriores o eliminandolos si eran nuevas inserciones. Esto no solo se realiza para el último log, si no que se realizará también para todos los logs anteriores que sean del mismo tipo, haciendo de esta forma que todos los archivos involucrados vuelvan a su estado anterior. Como no se realizó un ack del batch todavía, el batch que estaba siendo procesado al momento de caerse el nodo, simplemente será procesado nuevamente en la próxima recepción de mensaje, ya con todos los archivos rollbackeados. Esto puede hacer que resultados parciales sean enviados múltiples veces, pero siempre lo harán con el mismo sequence number, por lo que serán filtrados en el destinatario.
- **FinishedWriting():** Si al iniciar nos encontramos con finished writing, significa que el batch ha sido procesado, sus resultados parciales enviados, y todas las modificaciones persistidas a disco. Sin embargo no podemos afirmar si el ack fue enviado o no. El nodo podría haber muerto luego de enviar el ack pero antes de loguear AckBatch, o directamente antes de enviar el ack. Para verificar en cuál de los dos casos nos encontramos, se recibe inicialmente un batch. Como el nodo es el único que recibe de su cola, y rabbit, si hay un solo consumidor, mantiene el orden de los mensajes al reencolarlos, si el mensaje no fue ackeado entonces lo recibiremos nuevamente. Podemos darnos cuenta de esto, ya que tenemos la información del sequence number persistida, y si en efecto es el mismo batch podemos ackearlo y loguear AckBatch. Si en cambio es otro batch, significa que ya lo habíamos ackeado, por lo que solo logueamos AckBatch y el batch sacado de rabbit es devuelto haciendo un Nack del mismo. Luego se continúa con la ejecución

normal, desde el punto donde se realiza el log de AckBatch. Este proceso se podría mejorar almacenando el mensaje recibido, en vez de hacer un Nack para después recibirlo nuevamente.

- **AckBatch():** Si al iniciar nos encontramos con este tipo de log, significa que tenemos que verificar si hay algún cliente a quien hay que mandarle los resultados finales. Esto se debe a que el último mensaje ackeado podría ser el último Eof de un cliente. En caso de serlo, se procede a enviar los resultados finales del mismo.
- **SentFirstFinalResults(client_id, already_sent_results, last_batch):** Si al iniciar nos encontramos con este tipo de log, significa que estábamos en el proceso de enviar resultados finales del cliente. Para esto es que ordenamos los resultados finales, para poder tener siempre un orden consistente al volver a iniciar, y poder saber que resultados finales fueron enviados sabiendo solo la cantidad. Nuevamente se puede duplicar un batch de resultados enviado si no se llega a logear luego del envío, pero no hay problema ya que se utiliza el siguiente sequence number a last batch y por ende si se duplica tendrán el mismo sequence number y será filtrado por el receptor.
- **FinishedSendingFinalResults(client_id, last_batch):** Si al iniciar nos encontramos con este log, significa que debemos eliminar tanto de Ram como de disco al cliente y setear el último batch enviado en last batch, tanto en disco como en ram. Esto se hace con un Redo, es decir que cuando se vuelve a iniciar el nodo, se reintenta todas las operaciones, por más que algunas de estas ya hayan sido realizadas.
- Si el log está vacío, significa que no hay necesidad de realizar ninguna inicialización especial, ya que toda la información quedó en un estado válido.

Cabe aclarar que si bien hay algunos nodos que no tienen que enviar resultados finales, respetan el mismo procedimiento, simplemente nunca loggean SendingFirstFinalResults.

Metadata:

Como se mencionó anteriormente la metadata es todo lo que necesita el nodo en cuanto información del sistema para poder continuar con su ejecución de manera correcta al levantarse. Esto incluye 3 tipos de datos:

- Un único dato que guarda cual es el último sequence number utilizado en el último batch enviado. De esta manera al levantarse el nodo continuará usando los sequence number desde el mismo punto y de esta manera en caso de tener que reenviar algo por segunda vez que pueda ser filtrado por el receptor.

- Un campo por cada sender que guarde el último sequence number del mismo, para que al recibir un nuevo batch se pueda comparar con el último batch recibido incluso si este fue en una ejecución anterior.
- Un campo que indique cuántos eof faltan recibir de cada cliente que esté utilizando el sistema. Esto se debe a que se debe recibir un eof por cada worker que le envie mensaje al nodo, indicando que ese worker terminó de procesar los datos del usuario. Para no perder esta información se persiste en la metadata. Una vez enviados los resultados finales del cliente como parte de remover al ciente, se elimina este campo.

Contexto:

El contexto guarda la información mínima necesaria para poder obtener los resultados y es distinto para cada tipo de acumulador. Los filtros también cuentan con contexto ya que se guardan los títulos de los libros que cumplen con la condición. De está forma, se logra filtrar también las reseñas de los libros que no se encuentran en el contexto. Esto se podría mejorar ya que hay queries que no reciben reseñas y por lo tanto estos filtros podrían ser stateless, pero esta configuración no fue implementada.

Manejadores de archivos:

Dado el gran volumen de datos a procesar y la gran cantidad de datos que vamos a tener que estar persistiendo a disco, se diseñaron dos manejadores de archivos binarios para hacer eficiente el almacenamiento de datos y atómicas algunas operaciones de escritura que lo necesitan.

KeyValueStorage:

Dada la gran cantidad de datos que maneja el sistema, reescribir todo el archivo de datos, o al menos una gran porción de él cuando se requiere modificar los datos es algo muy costoso. Para evitar esto buscamos diseñar un sistema de almacenamiento que permita únicamente modificar una pequeña cantidad de bytes cuando se quiere agregar una nueva entrada o modificar un dato. Como su nombre lo indica el KeyValueStorage, permite dada una clave guardar un conjunto de valores de distintos tipos de datos, y lo hace modificando únicamente el valor en cuestión. La forma en la que se logró esto es mediante tamaño fijo de datos.

Cuando se crea un KeyValueStorage, se le debe indicar además del archivo, el tipo de dato de la clave, la longitud que tiene la clave, los tipos de datos de los valores de cada clave y el

tamaño de cada uno de ellos. Esto permite esencialmente armar una tabla donde todas las entradas tienen el mismo tamaño. Al almacenar un dato en la tabla la clase KeyValueStroage también se guarda qué número de entrada es. Entonces a partir de este número de entrada y el tamaño fijo de las entradas, si uno quisiera modificar una entrada del medio, uno podría simplemente pisar esos bytes, sin necesidad de tocar más bytes. Por ejemplo:

str Key (8 bytes)								int (4 bytes)			
'B'	'o'	'o'	'k'	'1'	255	255	255	0	0	0	15
'B'	'o'	'o'	'k'	'2'	255	255	255	0	0	0	2
'B'	'o'	'o'	'k'	'3'	255	255	255	0	0	0	4

Si se tiene un key value storage donde la clave es un string de 8 bytes y cada key tiene asociado un entero de 4 bytes el archivo se vería algo así. Uno podría por ejemplo aumentar en 1 el valor de ‘book2’. Para obtener la posicion del numero a modificar se puede usar la siguiente fórmula:

$$índice \ * \ \#bytes\ por\ entrada \ + \ \#key\ bytes$$

Entonces el Key Value storage utilizará su diccionario interno para saber que ‘book2’ se encuentra en la posición de índice 1 y entonces sabría que el dato a modificar empieza en 1*12 + 8 es decir en el byte 20. De esta manera solo debería modificar los bytes 20, 21, 22 y 23 para modificar la entrada.

Como se ve en el ejemplo si un dato usa menos bytes del tamaño máximo se rellenará con padding. En el caso de los strings se rellena con 255 ya que no es un caracter válido dentro de UTF-8.

Fallos en la escritura:

Si el archivo ya estaba creado y tenía entradas, el KeyValueStorage las levantará, le devolverá las claves y valores al nodo para que este las tenga en RAM y además almacenará en su diccionario interno las posiciones en las que se encuentran las claves. SI falla la escritura a la mitad pueden ocurrir dos casos:

- Se estaba escribiendo un valor cuya clave ya se encontraba guardada. En este caso el valor quedará corrupto, pero cuando se haga el rollback con el log, se pisará y dejará el valor anterior.
- Si se estaba escribiendo un nuevo valor, es fácil de identificar ya que al final de todo habrá una entrada que no cuente con todos sus bytes.

Swap remove:

Además se implementó para la utilización de algunas Queries un remove. Más específicamente un swap remove, donde se toma el último elemento y se pisa el elemento a eliminar. Este proceso no se puede realizar sin precauciones, ya que si se llegara a modificar a medias una de las claves, ni siquiera usando el log se podría mantener la validez de las entradas. Esto es debido a que no se podría diferenciar la entrada inválida de una entrada que no se modificó en el último batch. Para solucionar esto se “tacha” primero la clave de la entrada a modificar con un carácter inválido, es decir se pisa con este caracter (254 no es usado en utf8), luego se pisa con la última entrada y finalmente se trunca el archivo. En cualquier punto que falle esto se puede recuperar la validez del archivo al volver a levantarlo. Por ejemplo si se quiere tachar la segunda entrada del ejemplo anterior:

- Si se empieza a tachar la clave pero no se termina [254,254,’o’,’k’,’2’,255,255,255] se podrá identificar que se estaba eliminando esta entrada al levantar debido a que contiene 254.
- Si se terminó de tachar pero no se terminó de escribir la nueva clave entonces también se podrá identificar que esa entrada se estaba eliminando debido a que contienen 254. [‘b’,’o’,’o’,’k’,’3’,255,254,254]
- Si se termina de escribir la clave, pero no los datos, nos quedará la entrada inválida, pero no es problema porque cuando se vuelva a encontrar la misma clave con otro valor al final del archivo, se tomará éste como su valor.

Distintos tamaños usando datos de longitud fija:

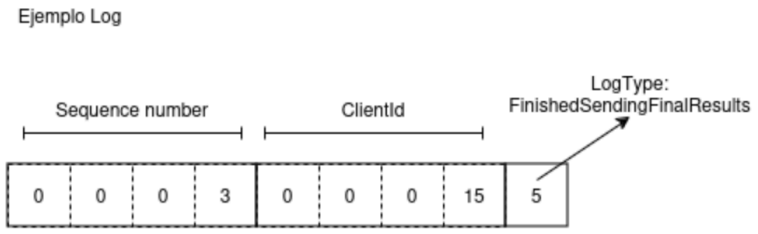
Así como el KeyValueStorage saca muchas de sus ventajas a partir de los tamaños fijos de datos, también esto trae desventajas. Esto viene en 2 formas, por un lado si el tamaño fijo es mucho más grande que el dato que se quiere almacenar, se gastará mucha memoria en padding. En el caso contrario, si el dato no entra en el tamaño fijo, este será recortado, y 2 datos distintos podrían terminar con la misma clave. En nuestro caso, estos problemas aparecen con los string, tanto en el tamaño del título como en el tamaño de los autores.

Para tratar de solucionar ambos problemas, no se utilizó un único archivo por contexto de usuario. Si no que se definieron múltiples archivos con distintas longitudes para los string. Más precisamente cuando se procesa un mensaje de un cliente, se toma el tamaño en bytes del string (este es diferente al tamaño del string debido a caracteres que ocupan más de un byte) y se obtiene un valor i que es la menor potencia de dos en la que entra ese tamaño. Entonces se almacena en un archivo que tiene todas entradas cuyos string entran 2**i pero no en 2**(i-1). Si bien esto aumenta un poco el overhead debido a la cantidad de

archivos, se gana mucho más ya que se reduce la cantidad de bytes utilizados por cada entrada y se garantiza que independientemente de las longitudes los datos entrarán en el storage sin ser recortados.

LogReadWriter:

Al igual que el KeyValueStorage el logger es un manejador de archivo binario, esta vez para manejar eficientemente los logs, y aún más importante que estos sean atómicos.
Inicialmente se pensó en utilizar un archivo auxiliar en donde se escribirán las entradas del log, y una vez finalizado se renombrará para convertirse en el log. Esto es atómico porque al finalizar el archivo llamado log tiene todos los nuevos bytes o ninguno. Esto no nos convenció, por la necesidad de crear archivos constantemente, por lo que optamos por otro sistema.
Antes de hablar del otro sistema necesitamos explicar un poco más cómo son nuestros logs. El archivo de log se lee de atrás para adelante y por ende los logs se estructuran de la misma forma., teniendo al final un byte que indica el tipo de Log y este indica cómo interpretar el resto de resultados.



De no tener escritura atómica, al leer el último log, si no se hubiera completado la escritura se podría interpretar un dato como si fuera un tipo de log. Para realizar la escritura atómica entonces definiremos 2 tipos de log auxiliares que no tienen ninguna información además del tipo, serán Prepare y CountPrepare. Como su nombre lo indica “preparan” el archivo para ser logueado. Si se quiere escribir un log de n bytes antes de escribir los bytes del log se escribirán primero n logs de tipo Prepare y n logs de tipo CountPrepare. Luego se pisarán los bytes de Prepare con los bytes del log y se truncaran los bytes de CountPrepare. En general la idea es reservar el espacio del log con Prepares y luego almacenar la información de la longitud del log con CountPrepare ya que cada Log CountPrepare indica que le precede un byte del log. Al escribirlo luego de los n bytes de Prepare se evita que se pierda la información de la longitud a medida que se van escribiendo los bytes del log. A continuación se explica cómo se manejan todos los casos de escrituras parciales:

- Si solo se llegaron a escribir algunos Log de Prepare al momento de leer el archivo de log estos son simplemente salteados y truncados.

- Si se escribieron todos los Prepare y solo algunos CountPrepare, se leen todos los CountPrepare y por cada uno de ellos se saltea un byte, luego si quedan aún Logs Prepare se saltean y se trunca el archivo
- Una vez escrito todos los Logs CountPrepare tenemos la información de la cantidad de bytes del log y por ende, independientemente de si se llegó a escribir todo el log o solo solo algunos bytes, los podemos saltear y nuevamente truncar.
- Si se escriben todos los bytes del Log entonces se trunca el archivo para eliminar los Logs de tipo count prepare.

Entonces, en cualquiera de los primeros 3 casos, luego de levantar el archivo será como si no se hubiera logueado nada, y en el último caso, están todos los bytes escritos, resultando entonces en una escritura atómica.

Persistencia Gateway:

De los procesos mencionados anteriormente del gateway el único que tiene persistencia de datos es el GatewayOut. Persiste de la misma manera que el resto de los nodos del sistema con el agregado de guardar el id del cliente más grande hasta el momento en la metadata. Esto se hace para que al iniciar nuevamente el sistema no se le asigne a un cliente el id de otro cliente asignado previamente.

Como última aclaración acerca de la persistencia del sistema, se utilizó la configuración de colas durables de rabbitMq para evitar que mensajes que estaban en las colas se eliminarán al no haber nadie conectado a las mismas.

Mecanismo para levantar procesos caídos

Ahora tenemos un proceso llamado Waker, haciendo referencia al proceso levantador de procesos, que puede tomar dos roles: el líder, que enviará healthchecks a todos los nodos del sistema (menos al que levanta rabbit y los clientes) para corroborar su estado, y si están muertos los revive, y los no líderes, que a parte de los workers también responderán los healthchecks del waker líder, y si se detecta su pérdida, se comenzará una nueva elección de líder usando el algoritmo de Bully

Un proceso líder lleva consigo un heap de eventos. Estos son inicialmente de dos tipos: tiempo de espera para enviar el próximo broadcast de healthchecks, y tiempo de espera

para timeoutear un nodo en particular. Entonces, al desencolar del heap se obtendrá el evento más próximo a vencerse. Si se trata del primer caso, enviará healthcheck a todos los nodos y posteriormente se volverá a encolar el mismo evento en el heap pero ahora con el tiempo actual de corrida sumado con un delay constante definido. Si es el segundo, supondrá que el proceso del evento se murió y lo revivirá, para luego encolar el nuevo evento para ese nodo con el tiempo actual de corrida sumado con una constante que representa el timeout tolerado.

Un ejemplo de eventos en el heap sería el siguiente:

- Event(HEALTHCHECK, 1),
- Event(ALIVE, filter1.1.0, 2),
- Event(ALIVE, waker2, 3.5)]

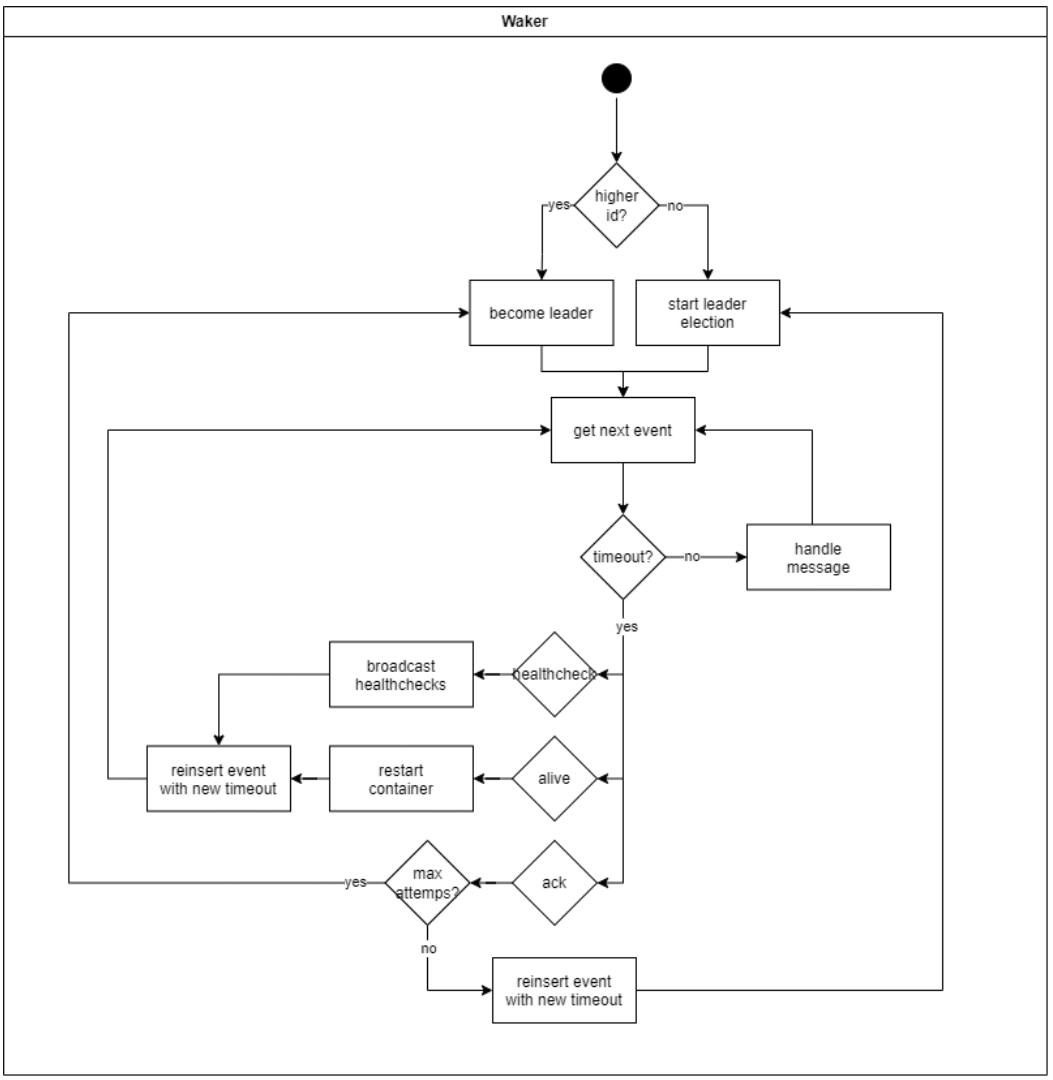
En este caso, el próximo evento a vencerse sería el healthcheck, luego la espera de un alive message por parte del *filter1.1.0* dentro de 2 segundos, y por último la espera de un alive message por parte de otro *waker* no lider, en 3.5 segundos.

Elección de líder

Se optó por Bully como algoritmo de elección de líder. Al principio de todo, un proceso levantador (Waker) se pregunta si tiene el id más alto de todos los wakers, y si lo tiene, se autoproclamará líder y enviará coordinator a todos. Si no, el proceso iniciará una elección de líder para conocer quién es el líder para luego a raíz de la llegada de healthchecks enviar confirmaciones de estado activo. Las conexiones se realizan mediante sockets UDP.

Adicionalmente se agregó un tercer tipo de evento para manejar una política de reintentos en el proceso de elección, donde si el nodo no líder desencola este evento de tipo ACK y alcanzó el máximo de intentos, luego de haber enviado los mensajes election a los líderes con id más altos que él, se autodesigna líder.

En el siguiente diagrama se puede observar el flujo de un waker



Distribución de tareas

Tarea	**Encargado**
Clase DatabaseHandler	Pablo
Clase Worker	Ambos
Subclase Filtro	Pablo
Subclase Acumulador - Contadores	Martin
Subclase Acumulador - Análisis de sentimientos	Martin
Biblioteca Interfaz de comunicación del middleware	Ambos
Subsistema Query 1	Martin
Subsistema Query 2	Pablo

Subsistema Query 3 y 4	Pablo
Subsistema Query 5	Martin
Elaboración del cliente	Pablo
Elaboración del gateway	Martin
Unión de partes	Ambos
Testing (durante todo el proceso)	Ambos
Clase KeyValueStorage	Pablo
Bajada a disco	Pablo
Sistema de ack	Pablo
Filtrado de duplicados	Pablo
Sistema de logeo	Pablo
Lectura de contexto al iniciar	Pablo
Mecanismo de levantar procesos caídos	Martin
Elección de líder	Martin
Envío de Heathchecks	Martin
Killer	Martín
Múltiples usuarios	Pablo