FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

Sistemas distribuidos I (75.74)
Trabajo Práctico 1: Escalabilidad
- Middleware y Coordinación de Procesos -

Integrantes	Padrón	Email
Godoy Dupont, Mateo	105561	mgodoy@fi.uba.ar
Harriet, Eliana	107205	eharriet@fi.uba.ar

Docentes:

Pablo D. Roca - Gabriel Robles - Franco Barreneche

Tomás Nocetti - Nicolás Zulaica

Amazon Books Analyzer

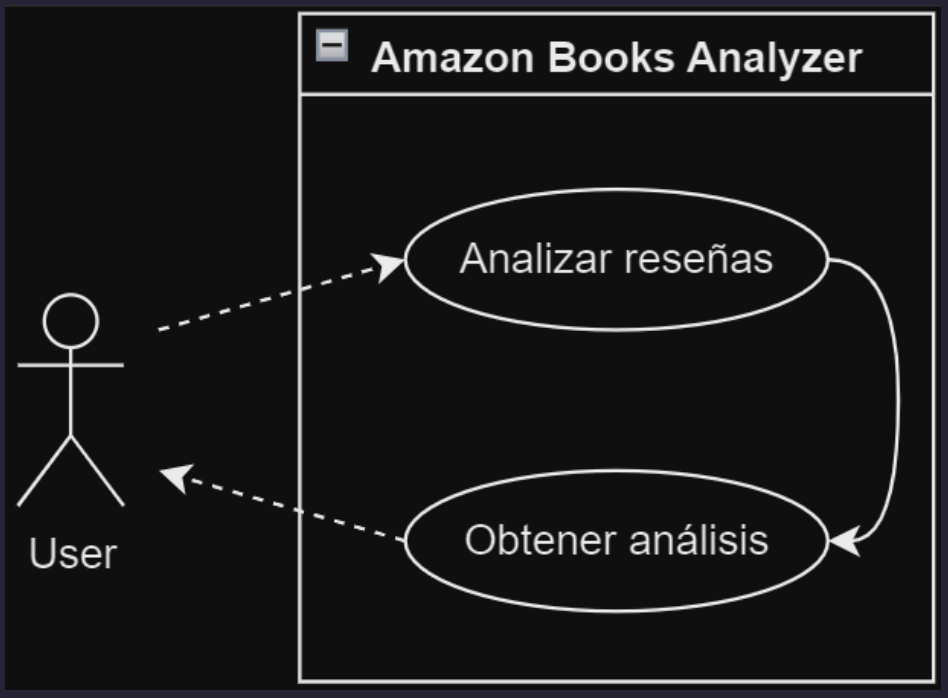
Se provee de un sistema distribuido que analiza las reseñas de los libros en el sitio de Amazon para proponer campañas de marketing. Las reseñas poseen título del libro, texto del comentario y rating. Por cada título de libro, se conoce categoría, fecha de publicación y autores.

El sistema cuenta con la capacidad de resolver las siguientes consultas:

- Título, autores y editoriales de los libros de categoría "Computers" entre 2000 y 2023 que contengan 'distributed' en su título.
- Autores con títulos publicados en al menos 10 décadas distintas
- Títulos y autores de libros publicados en los 90' con al menos 500 reseñas.
- 10 libros con mejor rating promedio entre aquellos publicados en los 90' con al menos 500 reseñas.
- Títulos en categoría "Fiction" cuyo sentimiento de reseña promedio esté en el percentil 90 más alto.

Estas consultas pueden correrse todas al mismo tiempo o seleccionando las de mayor interés. Además, si en un futuro fuera necesario modificar alguna de las consultas, o agregar una nueva, se busca que pueda llevarse a cabo reutilizando partes del sistema proveído. De esta forma se reduciría la complejidad de extender la utilidad del mismo.

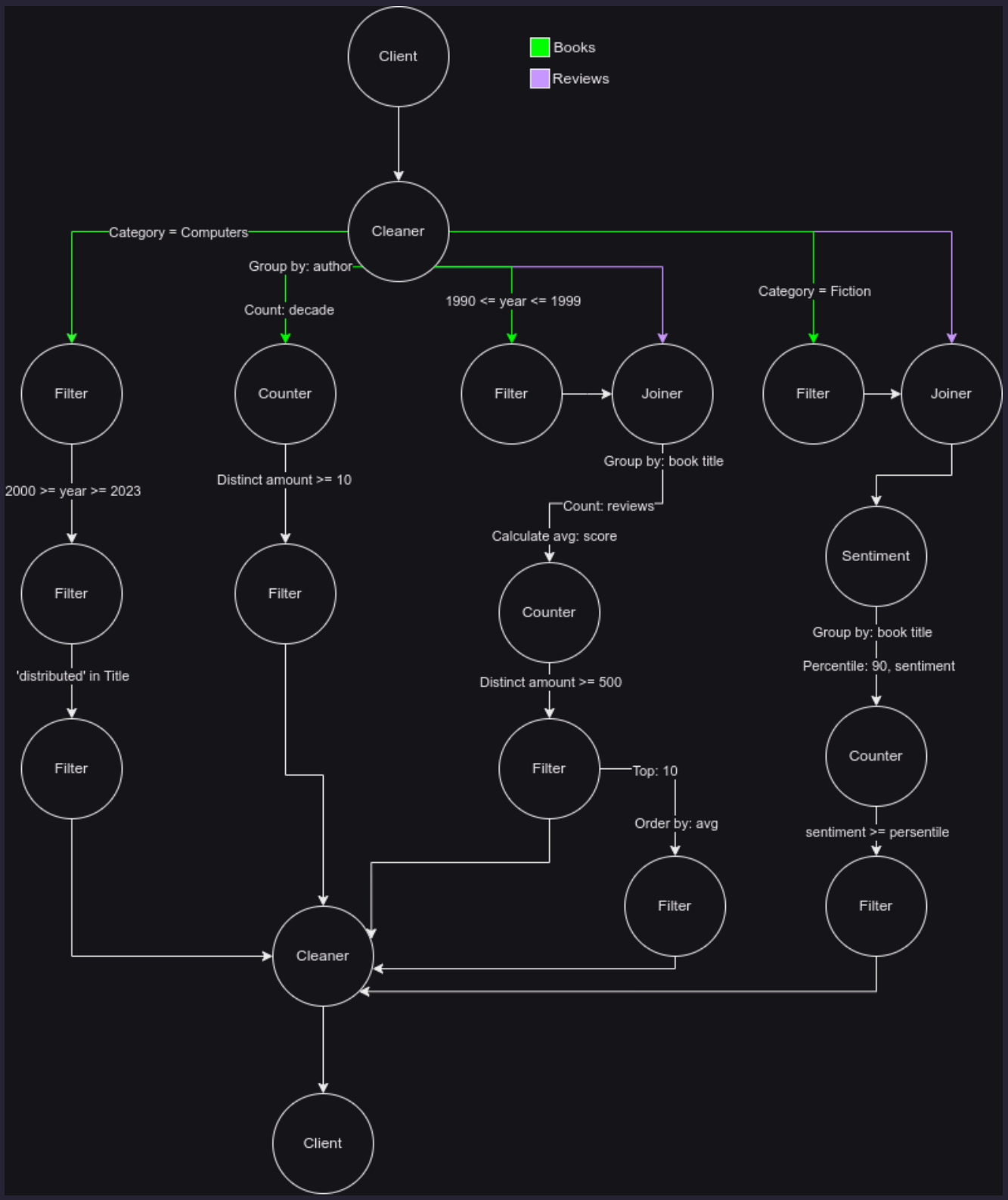
El objetivo principal del sistema es manejar un gran volúmen de datos, por esto mismo es necesario que sus componentes sean escalables.



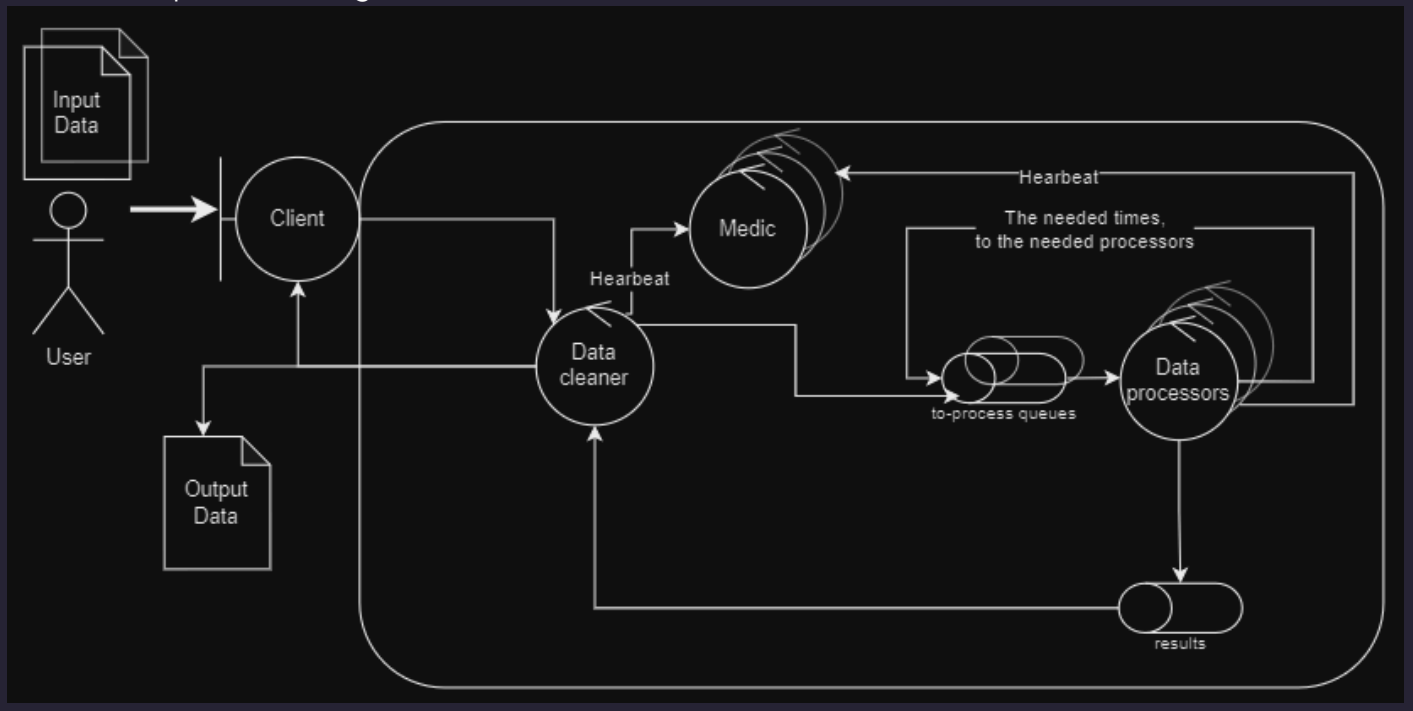
En primer lugar, el usuario puede iniciar el sistema para que comience a hacer el análisis de los datos. Una vez que el sistema comienza a correr, el usuario puede ir constatando el archivo de resultados que se va generando en su computadora. (El sistema no se cerrará hasta completar el análisis).

Esquema de resolución de las consultas

Se provee del siguiente diagrama DAG, en donde se muestra el camino que recorrerá la información para cada una de las consultas.



Respecto a la interacción entre componentes se tiene el siguiente diagrama de robustez, en donde se ve el comportamiento general del sistema:

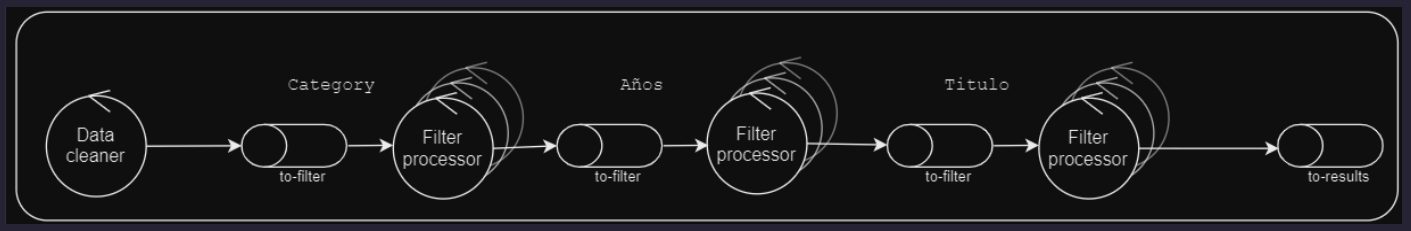


Se puede ver que se tiene una aplicación cliente, a la cual se le pasan los archivos a procesar y a continuación la información proveniente de estos archivos pasa por un nodo de limpieza. Así es como el sistema se queda con la información relevante y descarta datos que no sean íntegros. Los pasos siguientes corresponden a pasar por nodos procesadores de información tantas veces como cada consulta lo requiera y finalmente la información vuelve a limpiarse para enviar al cliente los resultados.

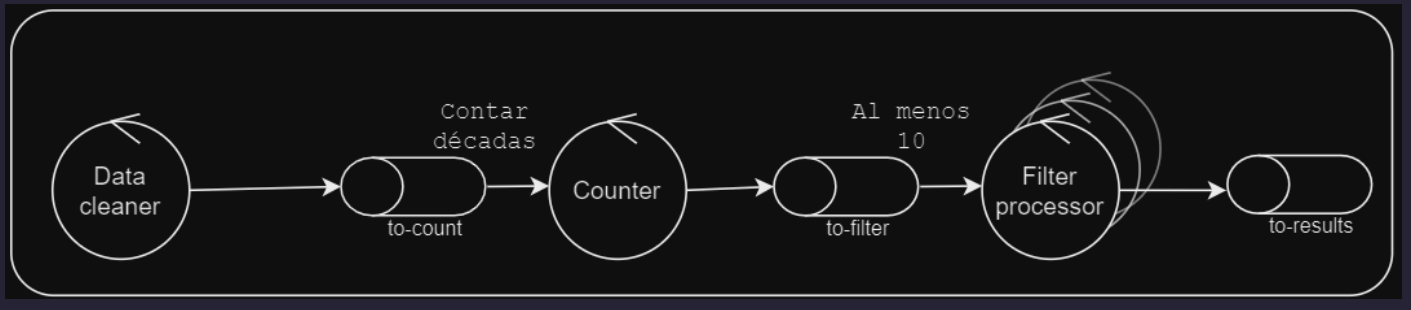
Para cada una de las consultas el paso a paso es el siguiente:

Nota: Cada nodo de tipo X (siendo X un counter, un filter, joiner o sentiment analyzer) corresponde a la misma entidad, pero no son necesariamente nodos diferentes. Se graficó de esta forma para evitar graficar ciclos que puedan confundir.

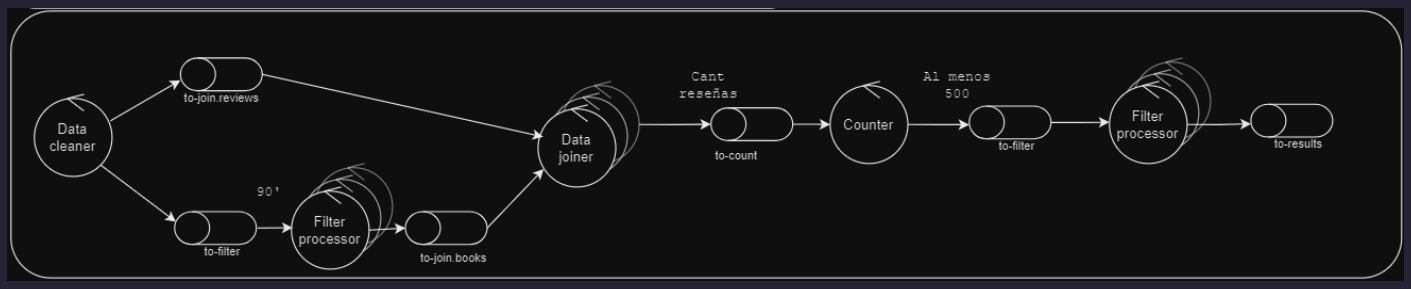
Consulta 1



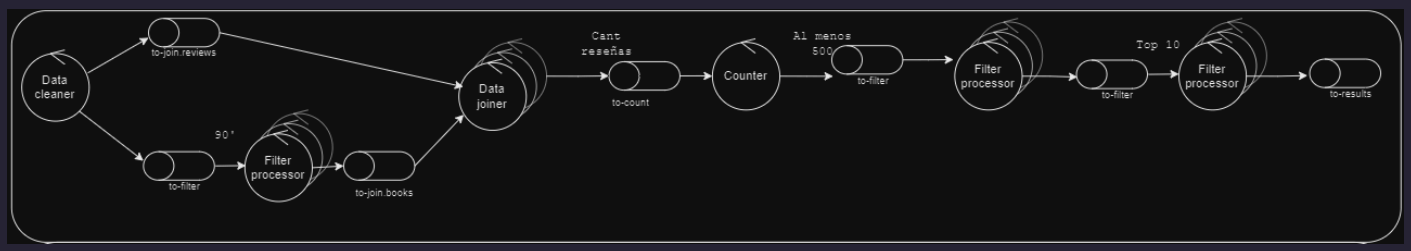
Consulta 2



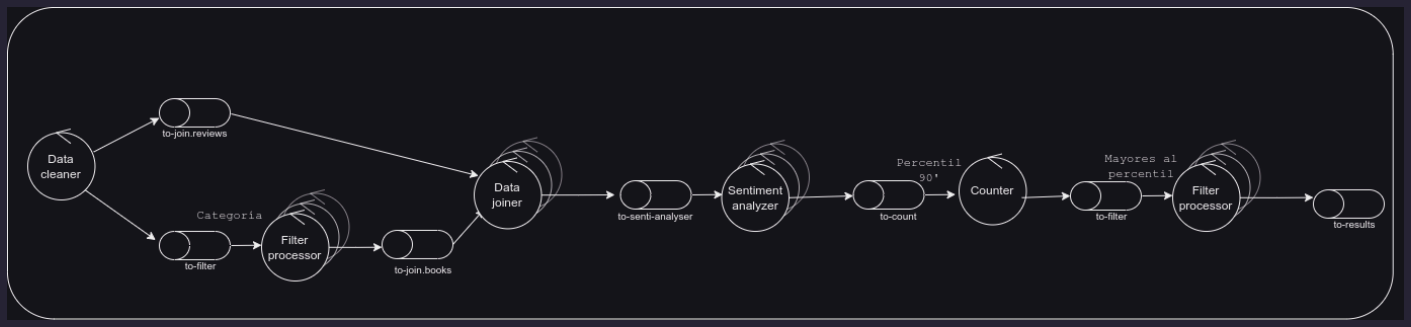
Consulta 3



Consulta 4



Consulta 5



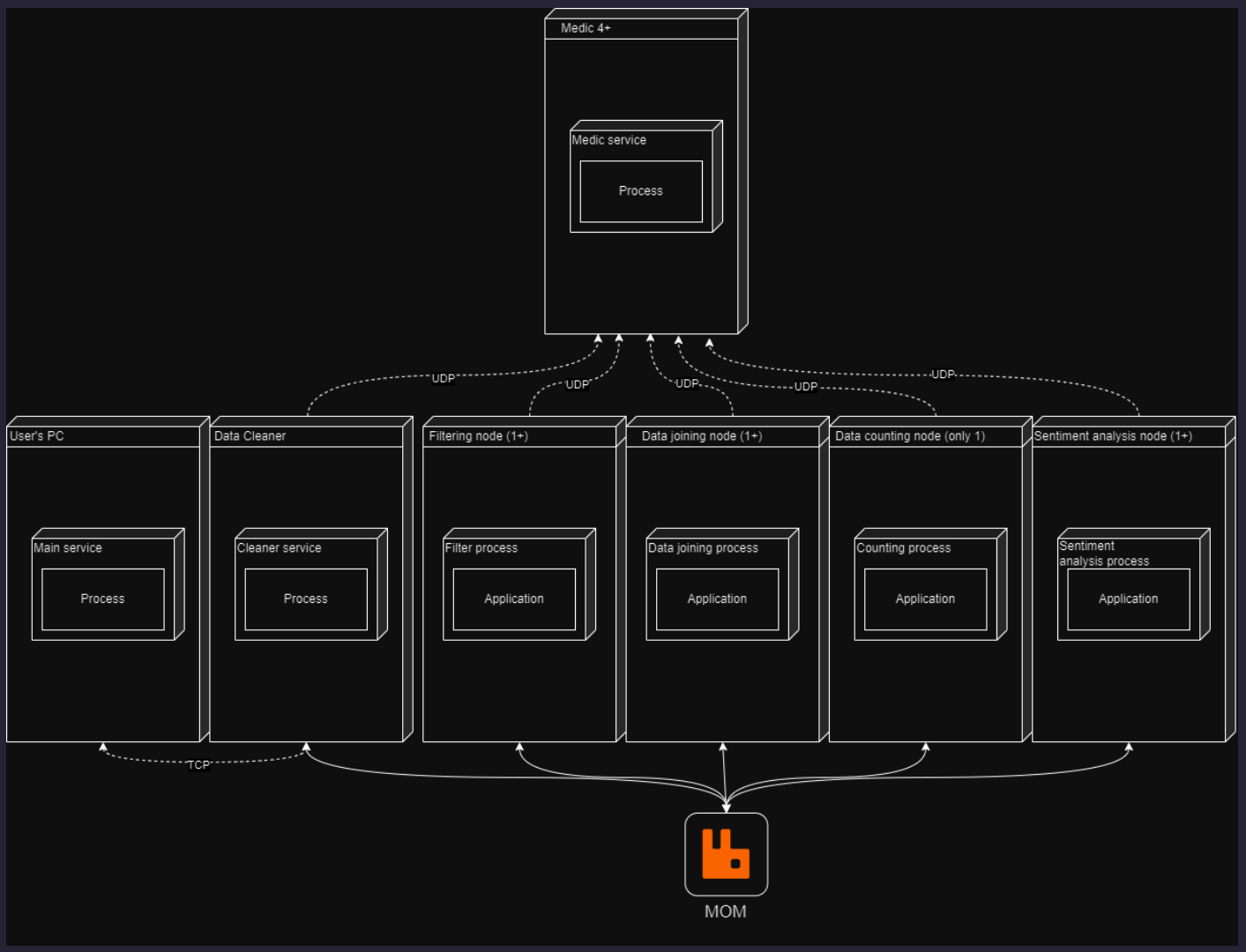
Despliegue del sistema

Como ya se mencionó, el sistema se compone de nodos. Los cuales son los siguientes:

- Client: Este nodo está fuera del servidor, se encuentra en la PC del usuario. Se comunica vía TCP con el servidor, siendo sus interacciones con el DataCleaner.
- Data Cleaner: Es el encargado de limpiar información, dejando pasar lo indispensable.
  En la ida es el encargado de recibir y limpiar la información que envía el cliente, descarta la información que no es correcta (ej.: campos de fecha mal completados) o que no cumple con un mínimo (ej.: libros sin título).
  En la vuelta es el encargado de recibir los resultados, quedarse con las columnas relevantes, y enviarlas al cliente.
  Sumado a sus tareas de limpieza de datos, este nodo es el encargado de otorgarles el formato correcto a los datos tanto desde el cliente al resto del sistema como en el camino inverso.
- Filter: Es el encargado de hacer los filtros correspondientes a las consultas. Es un único filtro que tiene la capacidad para responder en cualquier step correspondiente a filtrado de cualquier consulta. Además, puede instanciarse tantas veces como se lo requiera.
- Joiner: Es el encargado de recibir libros y reviews para luego devolver fragmentos de información con el resultado de la junta de los mismos. Es un único joiner que participa en todas las consultas que lo necesiten y también puede instanciarse la cantidad de veces que se lo requiera.
- Counter: Es el encargado de hacer los conteos correspondientes a las consultas. Es un único contador que tiene la capacidad para responder en cualquier step correspondiente a contar/agrupar de cualquier consulta.

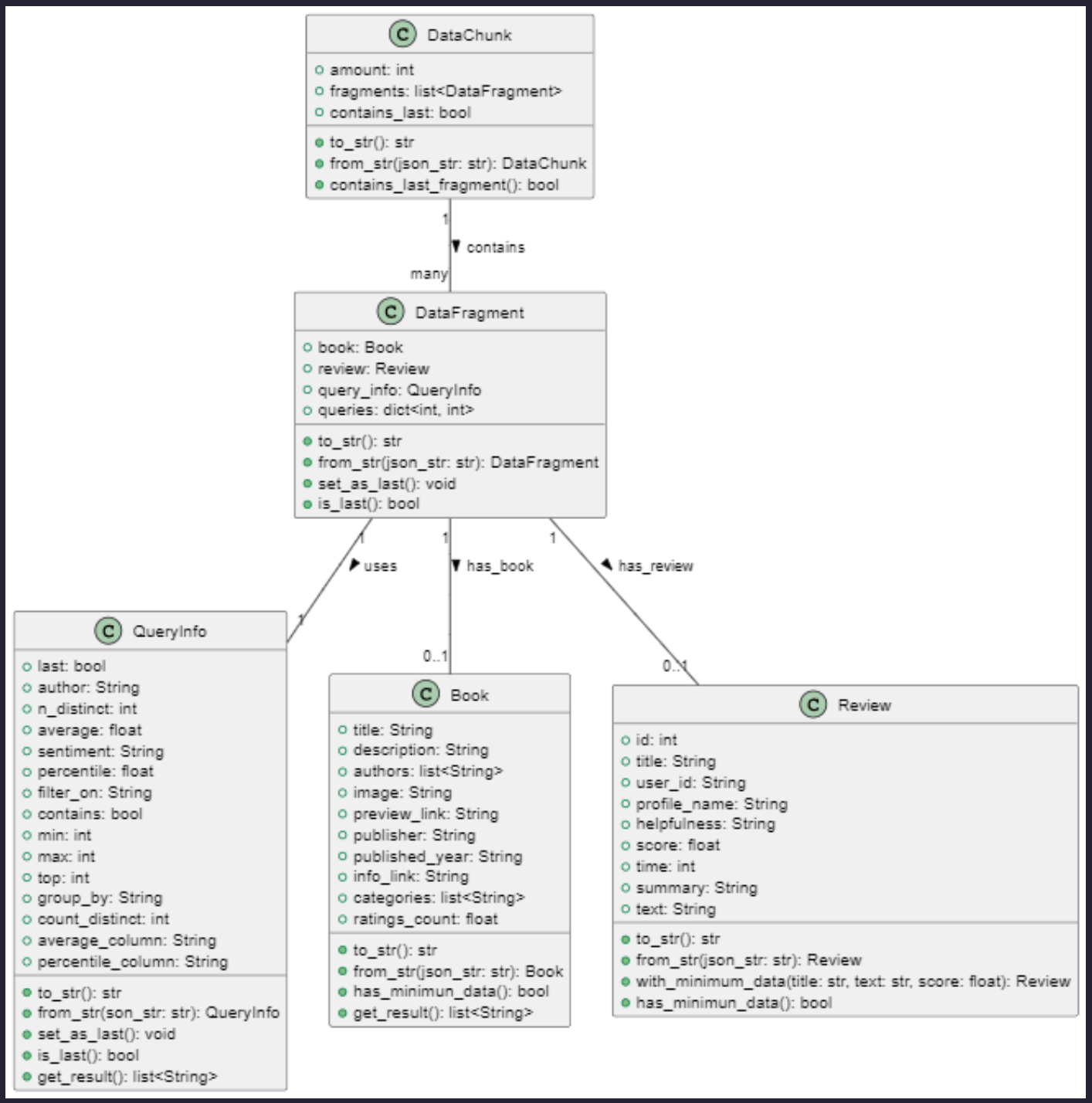
Es un nodo especial, que por limitaciones de diseño no puede ser escalado a más de una instancia. Sin embargo no se vieron consecuencias en tiempo muy severas por esto mismo.

- Sentiment Analyzer: Es el encargado de recibir reviews, tomar su texto principal y hacer un análisis de sentimiento. Puede instanciarse tantas veces como se lo requiera.
- Rabbitmq: Es el nodo en el que se instancia la herramienta utilizada para comunicar a los distintos nodos del servidor.



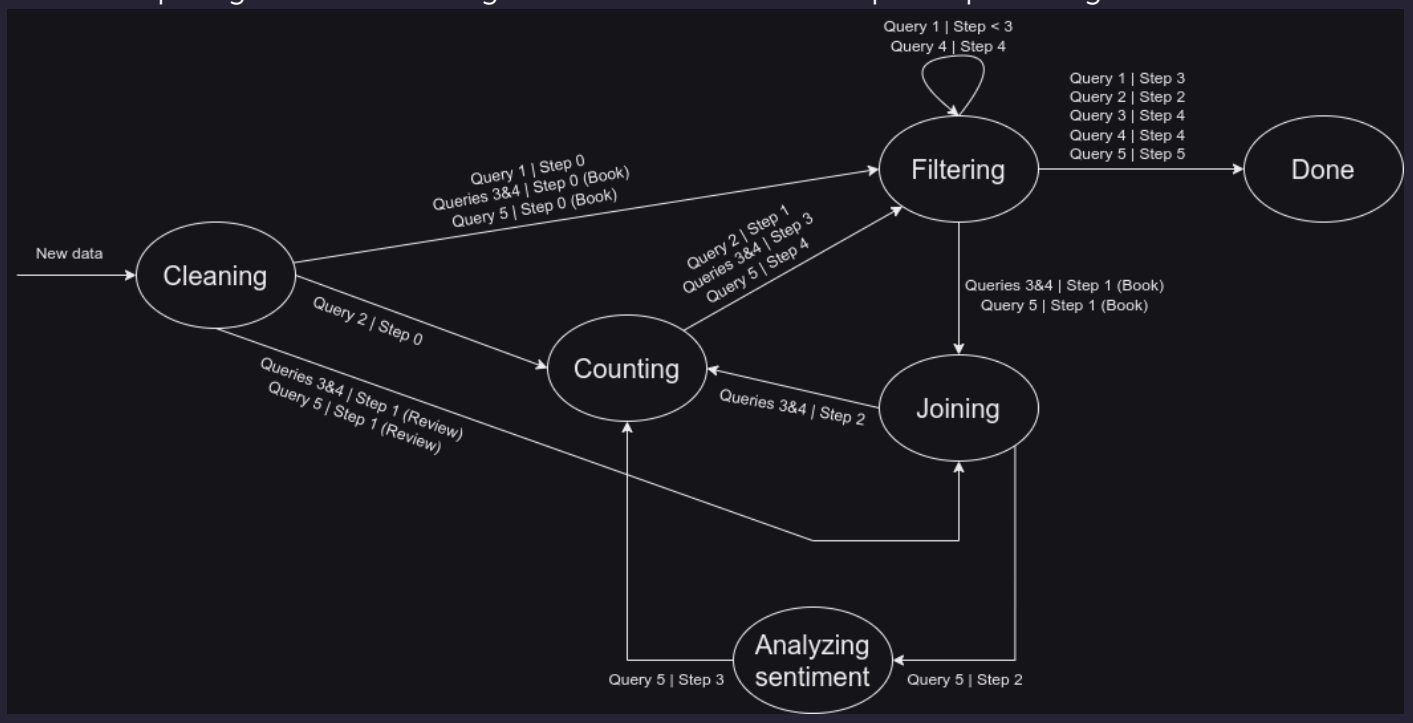
Manejo de los datos

Para trabajar de forma práctica, se crearon distintas estructuras para manejar la información. Se tiene el siguiente diagrama de clases en donde las mostramos:



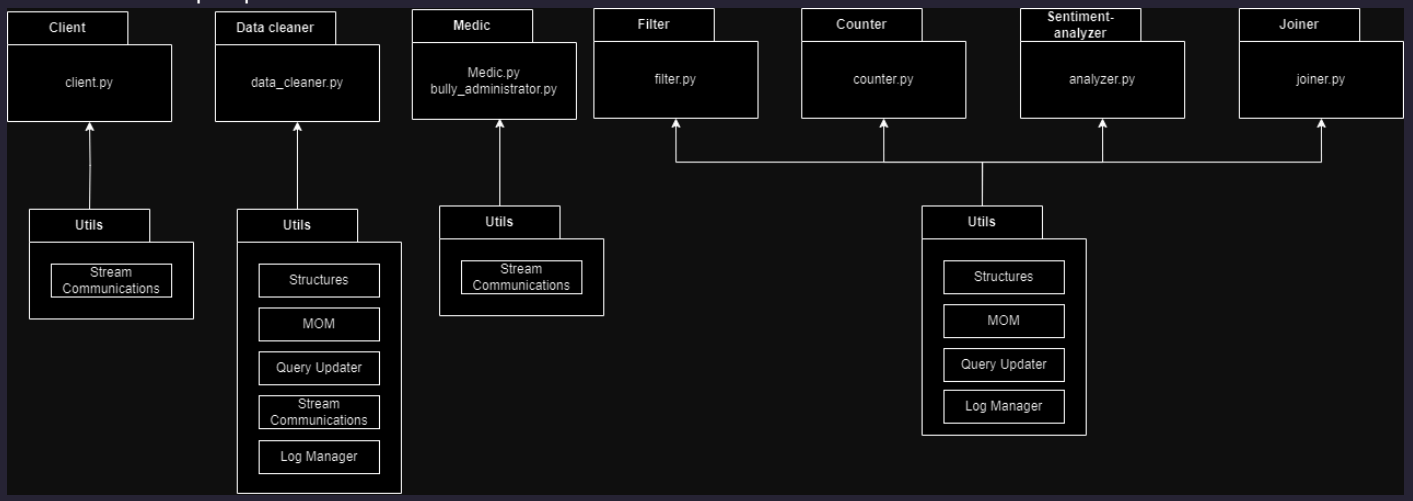
La unidad mínima de información es el DataFragment, en donde se irán haciendo las distintas actualizaciones correspondientes a cada fragmento de datos. El mismo fragmento es el que tiene la información necesaria para definir el siguiente paso en la consulta y almacenará los parámetros necesarios para que el nodo correspondiente pueda procesarlo.

El camino que sigue cada DataFragment al ser actualizado step a step es el siguiente:

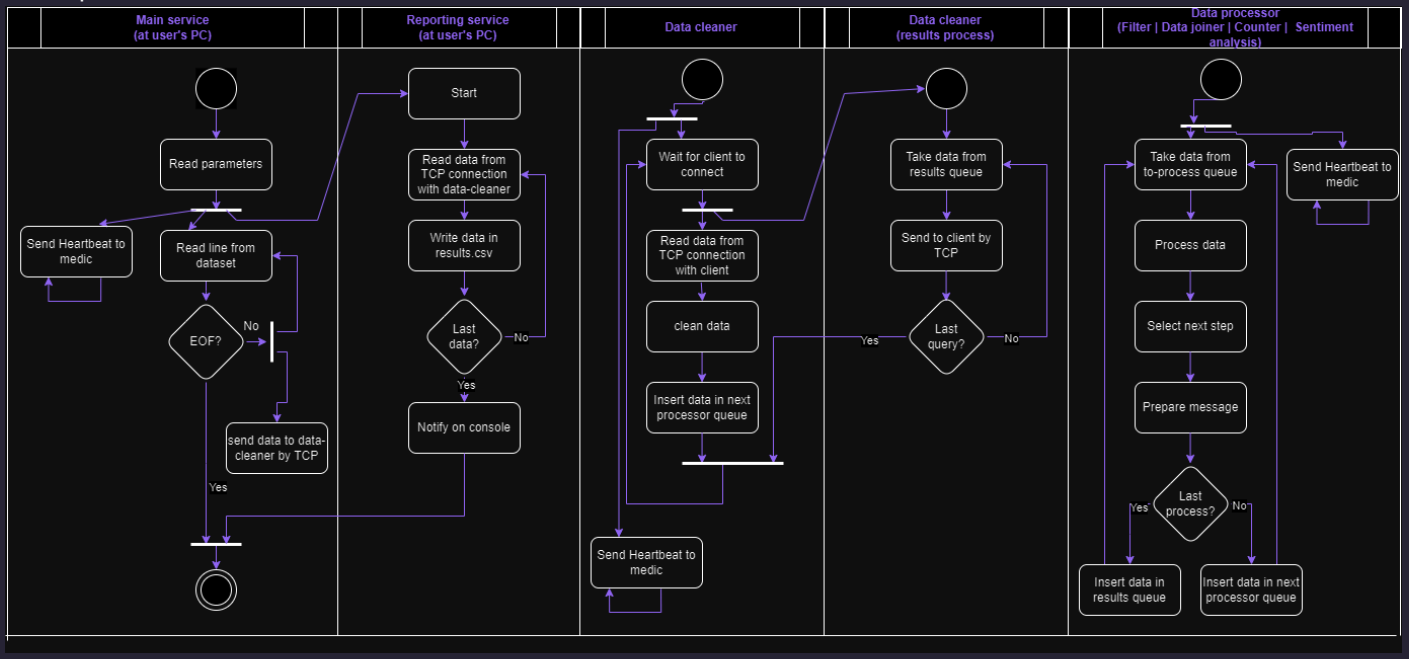


Desarrollo del sistema

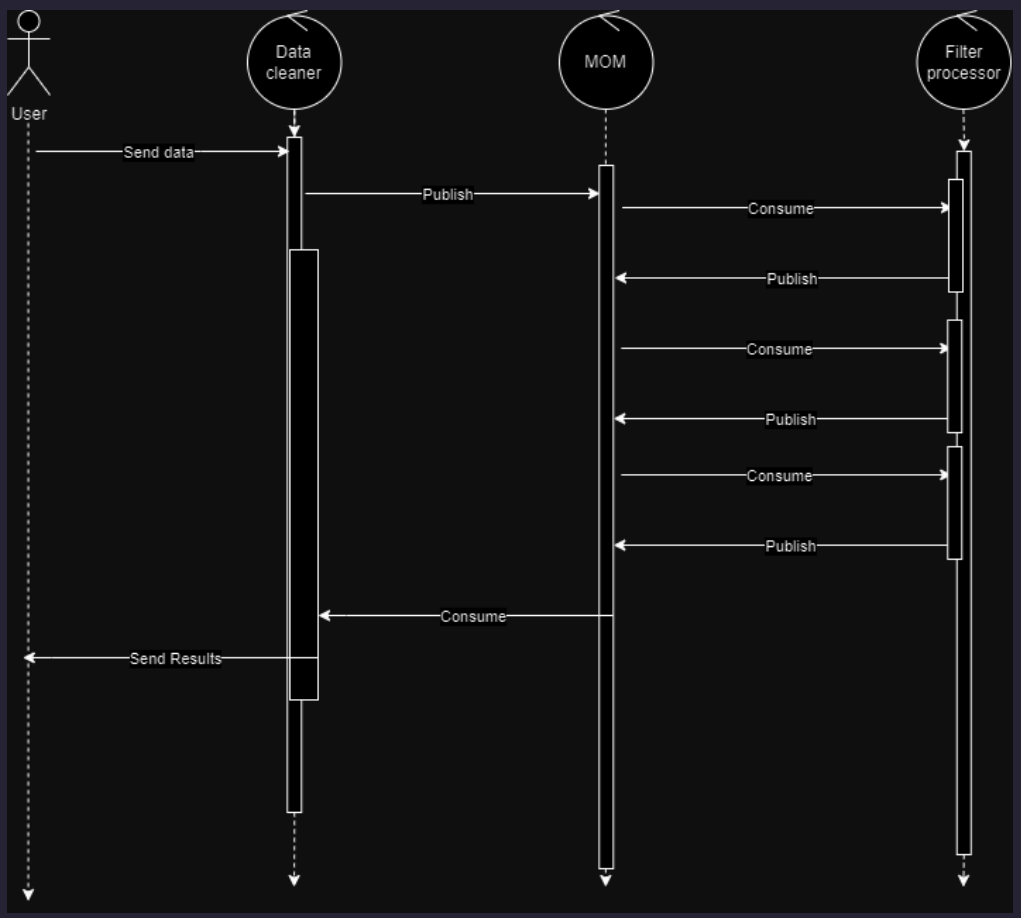
Ya habiendo mencionado los distintos componentes del sistema, en este diagrama se muestra la estructura de paquetes utilizada.



Respecto a los nodos, en este diagrama se muestran el paso a paso del procesamiento llevado a cabo por cada uno.



En el siguiente diagrama de secuencia se muestra un ejemplo de ejecución para la primera consulta (*Título, autores y editoriales de los libros de categoría "Computers" entre 2000 y 2023 que contengan 'distributed' en su título*). En este ejemplo hay un único nodo filter, pero podrían intervenir N nodos de tipo filter.



Ejecución del trabajo:

En lo que respecta a la ejecución del trabajo, se deben cumplir ciertas pre-condiciones. Las mismas consisten en tener ambos archivos de datos en la carpeta "data", es decir, los archivos de "books_data.csv" y "Books_rating.csv". Adicionalmente se debe configurar que queries se desea ejecutar, por defecto el cliente ejecutará las queries 1,2,3,4 y 5 sin embargo esto puede ser personalizado en el archivo de Dockerfile del cliente.
Una vez cumplidas las pre-condiciones, se deben levantar todos los nodos necesarios. Para ello es necesario estar parado en la carpeta "src" y en caso de ejecutarlos individualmente, se deberá ejecutar los comandos respectivos a cada uno de los siete nodos:

- Rabbitmq
- Cleaner
- Counter
- Filter
- Joiner
- Sentiment
- Client

Aquí vale la pena aclarar que los nodos que admiten múltiples instancias deberán ser ejecutados junto al índice de la instancia, es decir, para ejecutar tres instancias del nodo joiner, se deberían ejecutar los comandos:

- Ejemplo:
  - ‘Docker compose up joiner1’
  - ‘Docker compose up joiner2’
  - ‘Docker compose up joiner3’

Visualización del resultado:

Tras la ejecución, se podrán encontrar los archivos de resultados (uno para cada query) en la misma carpeta data donde anteriormente pusimos los csv con los datos.

Baja de nodos

Para bajar nodos aleatoriamente de forma práctica se proveen los siguientes alias:

Para bajar cualquiera:
alias kill_any='sudo docker ps | grep -v "rabbitmq" | awk "NR>1 {print \$1}" | shuf | head -n 1 | xargs sudo docker kill'

Para bajar nodos procesadores de data:
alias kill_processors='sudo docker ps | grep -v "rabbitmq\|client\|cleaner\|medic" | awk "NR>1 {print \$1}" | shuf | head -n 1 | xargs sudo docker kill'

Pueden ser aplicados al agregarse al archivo ~/.bashrc (luego correr el comando `source ~/.bashrc`)

En caso de querer “matar” un nodo a elección durante la ejecución, se puede utilizar el comando ‘docker kill ID_CONTAINER’.

Alias extra

Por si fueran de utilidad, se provee de los siguientes alias:

alias dc_prune='sudo docker system prune -af'
alias delete_all='rm_all && dc_prune && volume_rm && rm_logs && rm_data'
alias restart_all='stop_all && delete_all && clear && git pull && start_all'
alias rm_all='sudo docker rm $(sudo docker ps -aq)'

```
alias rm_data='rm -f ../data/Result*'
alias rm_logs='rm -f ../logs/*.log'
alias start_all='sudo docker compose up -d --force-recreate --build'
alias status_all='sudo docker ps -a'
alias stop_all='sudo docker stop $(sudo docker ps -aq)'
alias volume_rm='sudo docker volume rm amazon-books-analyzer_rabbitmq'
```

Recomendamos start_all para levantar el sistema, stop_all para frenarlo y delete_all para hacer la limpieza necesaria. Muchos de estos alias necesitan ser corridos desde la carpeta src.

Actualizaciones respectivas a la segunda entrega

Recepción de múltiples clientes

Con el fin de identificar a los diferentes clientes, se utilizan UUIDs. De esta forma cada dato puede ser diferenciado por consulta y cliente. Esto está hecho bajo la premisa que para que dos UUIDs coincidan, se debe generar una cantidad muy grande de los mismos, siendo así muy baja la probabilidad de colisión de los mismos.

Tolerancia a fallas ante caída de nodos:

Con el fin de que la ejecución del sistema pueda continuar aún si uno o múltiples de sus nodos se caen, se debieron implementar distintas medidas, entre ellas se encuentra la creación de un nuevo nodo llamado “Medic” y un protocolo de logueo. El rol del nuevo nodo es verificar continuamente el estado de los demás nodos del sistema y en caso de detectar una caída, levantar nuevamente el contenedor.

Modus operandi ante distintas caídas:

- **Cliente:** Ante el caso de la caída de la aplicación cliente se da por cerrada su sesión, el sistema al detectar esto enviará un mensaje señalando esto para que cada nodo pueda hacer la limpieza de datos correspondiente.
- **Cleaner:** Ante el caso de la caída del nodo que centraliza las requests de los clientes se cuenta con el logueo de información relevante. De esta forma, al momento de ser re-levantado por el medic se cuenta con los ids de los clientes que previamente estaban conectados. Ante el corte de la conexión cliente-servidor, debido a la caída de este nodo, se asumen cerradas todas las sesiones por lo que se enviará una señal a cada nodo del sistema para que puedan hacer las limpiezas que correspondan.
- **Nodos procesadores (counter, joiner, sentiment-analyzer, filter):** Ante la caída de un nodo de este tipo se cuenta con el logueo de la información relevante, de forma que al ser re-levantado por el medic no haya percepción de la caída (más que alguna demora temporal).
- **Medic:** Será explicado más adelante.

Eliminación de duplicados

Al momento en que se cae un nodo, el batch que estaba procesando puede ser procesado por otro nodo en su lugar. Es importante que los nodos no dupliquen información, es por esto que se cuenta con ids.
Cada DataFragment cuenta con un identificador único que, en conjunto con la consulta a la que corresponde y el cliente, sabemos que no puede estar dos veces ya que sino se estaría en presencia de un dato duplicado. De esta forma, cada nodo guarda todos los identificadores que procesó para que al momento de recibir una réplica poder omitirla.

Detección de caídas por parte del Medic:

El método elegido para realizar la verificación del estado de cada uno de los nodos del sistema es un “Heartbeat”. Se optó por este método por encima de otras opciones como un “HealthCheck” dadas las características del sistema, especialmente el hecho de poder tener una cantidad variable de nodos de cada tipo. Al implementar un sistema de Heartbeat, los nodos del sistema envían a los

nodos Medic un mensaje con formato “TIPO_DE_NODO.ID_DEL_NODO$” siendo ambos valores numéricos enteros que le permiten al medic determinar el nodo del cual está recibiendo el mensaje. Para el envío del heartbeat, se creó un proceso independiente al proceso principal del nodo el cual envía la información al medic con una frecuencia determinada únicamente si el proceso principal está ejecutándose correctamente.
Estos mensajes son enviados utilizando el protocolo de UDP y simplificando de esta forma el envío de los datos. Sin embargo, dado que este protocolo no es RDT se deben tener ciertas consideraciones respecto a la posibilidad de que el mensaje no llegue a destino o que llegue corrupto y para esto se creó una relación entre la frecuencia de envío del mensaje y el “timeout” o el tiempo en el que el nodo medic decide que un nodo está caído. Así, se define que el timeout deberá ser 4 veces la frecuencia de envío del mensaje ya que la posibilidad de que se pierdan los mensajes de un mismo nodo cuatro veces seguidas se considera lo suficientemente baja.
En lo que respecta de características de RDT como por ejemplo orden de los mensajes, control de flujo, mensajes duplicados, entre otros, no son necesarias y por eso consideramos correcto utilizar UDP.

Accionar una vez detectado una caída:

Cuando un nodo medic detecta la caída de un nodo mediante su respectivo timeout, deberá ejecutar un script el cual recibe como parámetro el nombre del nodo a levantar y utiliza las funcionalidades provistas por docker-in-docker para cumplir con su objetivo. Sin embargo, al tener múltiples instancias del nodo medic (más adelante explicaremos porque es necesario tener más de un nodo medic) se debe llegar a un acuerdo de cuál de ellos deberá levantar el nodo caído. Dicho acuerdo se obtiene al designar a uno de ellos como líder del resto mediante el algoritmo de “bully”.

Elección de un medic líder:

El algoritmo implementado toma como base al algoritmo de Bully y le agrega ciertas características con el fin de obtener algunas garantías o funcionalidades extra. En cuanto al protocolo de comunicación entre los nodos, se optó por TCP dado que la cantidad de conexiones no representa un costo en cuanto a recursos y se obtiene la ventaja de un protocolo RDT necesario para asegurarse que los mensajes lleguen a destino, lleguen en orden, sin modificaciones en su contenido.
El sistema tiene seis tipos de mensajes y a continuación se especificará la función de cada uno y el accionar del nodo al recibir dicho mensaje:

- ELECTION: Este mensaje es utilizado para iniciar una nueva elección, en caso de que un nodo reciba este mensaje deberá responder el mismo con un ANSWER e iniciar una elección enviando el mensaje a sus nodos con ID superiores. Así mismo deberá registrar el momento del inicio de la elección para que en caso de no recibir respuesta en un determinado timeout se proclamará el líder.
- COORDINATOR: Este mensaje se utiliza para avisar que el nodo que lo envía se declarara como líder. Al recibir este mensaje el nodo deberá registrar al nuevo líder, iniciar su ciclo de envío de mensajes ALIVE al líder, marcarse a sí mismo como un nodo “no líder” y enviar un mensaje de ACK.
- ANSWER: Este tipo de mensajes es utilizado para notificar a un nodo inferior que se recibió su mensaje de ELECTION. Al recibir este mensaje, el nodo deberá borrar su timeout de su proceso de elección.
- ALIVE: El propósito de este mensaje es que los nodos medic conozcan el estado de otros nodos. Siendo más específico, los nodos “no líder” conocerán el estado del líder y el nodo líder conocerá el estado de los nodos “no líder”. Al recibir este mensaje no es necesario realizar ningún accionar ya que el propio protocolo de TCP nos permite saber si el mensaje se entregó correctamente.
- ACK: Estos mensajes tiene la finalidad de informarle al nuevo nodo líder que el nodo al cual envió el mensaje COORDINATE se encuentra listo para que se transforme en el nuevo líder. La importancia de este mensaje radica en que si el nuevo líder se proclama como tal justo

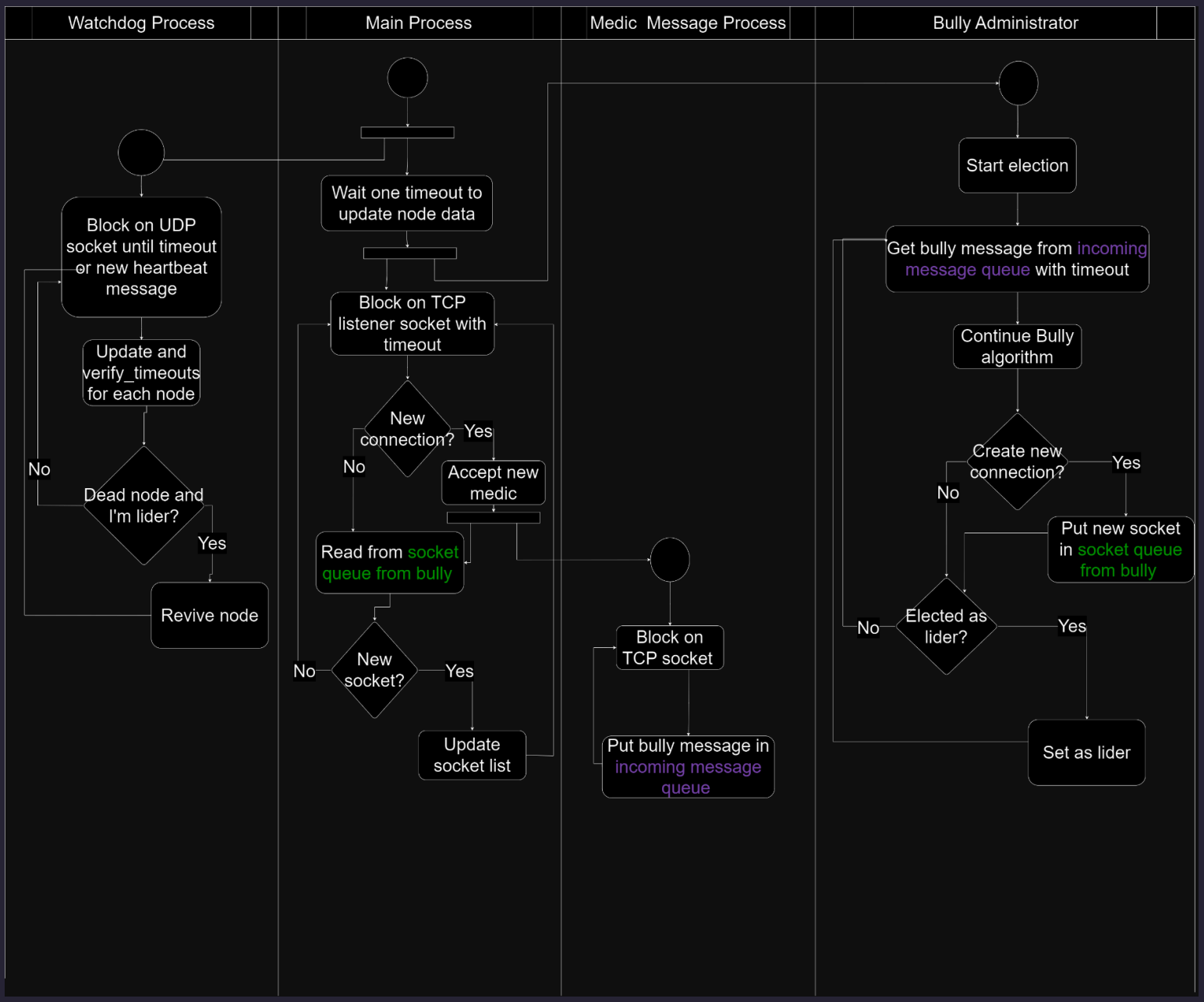
después de enviar el COORDINATE, es posible que el viejo líder tenga una cola de mensajes por procesar y exista una demora hasta que detecte el COORDINATE, bajo esas condiciones existirían dos nodos líder lo cual no es aceptable. El nodo que recibe este mensaje, deberá descontar uno a la cantidad de ACK faltantes antes de proclamarse como líder (dicha cantidad será la cantidad de COORDINATE enviados).
En este punto es importante aclarar que en caso de que un nodo se caiga luego de recibir el COORDINATE pero antes de enviar el ACK, el nuevo líder cuenta con un timeout que le permite declararse líder pese a no tener la cantidad de ACK necesaria.

- DEAD: Este es un mensaje interno de cada uno de los nodos que se utiliza para la comunicación entre procesos. Este mensaje contiene la información de un nodo medic al cual se le cortó la conexión TCP para que todos los procesos tengan los sockets actualizados.

Tolerancia a fallas ante caída de un medic:

Hasta este punto los medic pueden detectar caídas, levantar nodos del sistema que se encuentren caídos y seleccionar a uno de ellos como el líder del resto, sin embargo nace el interrogante de ¿Qué sucede si se cae un nodo medic?.
Esta posibilidad de que un nodo medic pueda fallar es la que nos lleva a tener más de un nodo medic, todos ellos con la misma información respecto a los heartbeat de los nodos del sistema (ya que los nodos del sistema enviar por UDP a todos los medic) para que al momento que el líder falle se seleccione un nuevo líder y se continúe sin problemas desde el mismo punto sin necesidad de logs.
Cuando un nuevo nodo es seleccionado como líder, se deberá encargar de levantar a todos los nodos medic con ID superior al suyo (en caso de haberlos) y luego por el algoritmo de Bully el liderazgo se transmitirá nuevamente al proceso con mayor ID de todos. Este proceso es el único capaz de levantar a nodos con ID inferior al propio y lo hace mediante un seguimiento de los mensajes de ALIVE que recibe. Esto se realiza con una verificación cada cierto tiempo de los ID de los mensajes recibidos para ese periodo.
Por lo tanto, el mensaje de ALIVE que envían los nodos más pequeños al líder no solo sirve para verificar que el líder sigue activo sino que el propio líder lo utiliza como heartbeat de dichos nodos. Esta solución presenta muchas ventajas frente a añadirlos al sistema de Heartbeat del resto del sistema siendo la principal un ahorro de mensajes y el aprovechamiento de la conexión TCP existente entre los nodos en lugar de enviarlo mediante UDP (con todas las consideraciones que se deberían tomar).
Por último, se debe tomar una consideración especial al momento de levantar un nodo ya que si el mismo comienza enviando un mensaje al inicio de su ejecución, entraría al “sistema de medics” pero con una información distinta al resto ya que es posible que todavía no haya recibido todos los mensajes de Heartbeat de los diferentes nodos del sistema. Para prevenir este caso, simplemente se debe esperar un ciclo de timeout de los nodos del sistema antes de integrarse nuevamente con el resto de medics. Sin embargo, esto implica una relación entre el timeout con el que el los medic reconocen la caída de los nodos del sistema y el timeout de los nodos medic siendo este último mayor al primero ya que de ser de otra forma el modic líder intentará levantar nuevamente un nodo que ya está levantado pero esperando que se cumpla este tiempo de sincronización de datos.

Implementación del nodo medic:



Protocolo de logueo de información

Se cuenta con un protocolo de logueo de información, en donde muchas de las líneas son compartidas entre distintos nodos, mientras que otras son particulares a comportamientos o estados especiales.

Nodos filter y sentiment-analyzer

Son los nodos más sencillos en los que a este tema respecta, ya que no guardan gran cantidad de información. De esta forma, vamos a encontrar las siguientes líneas en sus logs:

- RECEIVED_ID: Señaliza el id de un DataFragment recibido, de esta forma un nodo no procesa dos veces al mismo fragmento. Contiene los campos <client_id> <query_id> <df_id>
- RESULT: Corresponde a un resultado que será guardado en un batch de salida, esperando su envío. Contiene los campos <node> <datafragment como str>.
- QUERY_ENDED: Señaliza la terminación de una consulta, de esta forma al leer los logs pueden omitirse las líneas RECEIVED_ID que correspondan a dicha consulta. Contiene los campos <client_id> <query_id>.
- RESULT_SENT: Señaliza el envío de un batch. De esta forma pueden omitirse las líneas RESULT que hayan sido enviadas. Contiene el campo <node>.
- IGNORE: Señaliza el término de la ejecución de las consultas de un determinado cliente. Permite omitir todas las líneas de información que correspondan al mismo. Contiene el campo <client_id>.

Nodo joiner

En los logs de este nodo se pueden encontrar las líneas mencionadas anteriormente, sumadas a las siguientes:

- BOOK: Contiene la información respecto a un libro, para poder mantener la side table con la que está trabajando. Cuenta con el campo <book como str>.
- SIDE_TABLE_UPDATE: Señaliza la adición de un libro a una side table. Las side tables están divididas por cliente y consulta. Cuenta con los campos <client_id> <query_id> <book title>.
- SIDE_TABLE_ENDED: Señaliza la terminación de la recepción de libros para una side table determinada, de esta forma se sabe que está lista para ser usada y recibir las reviews que correspondan. Contiene los campos <client_id> <query_id>.

Nodo counter

En los logs de este nodo se pueden encontrar las líneas IGNORE, RECEIVED_ID, QUERY_ENDED y se agregan dos líneas nuevas:

- COUNTED_DATA: Contiene información parcial respecto a un conteo que se esté llevando a cabo. Contiene los campos <client_id> <query_id> <count info>, en donde <count info> depende del tipo de conteo (suma total, promedio, percentil, etc).
- COUNTED_DATA_SENT: Señaliza el envío de la información contada, correspondiente a una query que terminó. Permite la omisión de las líneas RECEIVED_ID, QUERY_ENDED y COUNTED_DATA que correspondan. Contiene los campos <client_id> <query_id>.

Nodo cleaner

El nodo cleaner es muy básico en cuanto a sus logueos, cuenta con las siguientes líneas:

- NEW_CLIENT: Señaliza la recepción de un cliente nuevo. Contiene el campo <client_id>.
- ENDED_CLIENT: Señaliza el término de la ejecución de las consultas de un cliente. Se loguea esta línea tanto para sesiones exitosas como para sesiones que fueron interrumpidas debido a una caída. Contiene el campo <client_id>.

    Al tener estas dos líneas en log, al momento de levantarse el nodo cleaner se enviará una señal de limpieza para todos los clientes de los que cuente con la primera línea pero no la segunda.

Implementación del protocolo de logueo

Escritura

Cada línea es logueada tras el momento exacto en el que su acción correspondiente ocurre. Tras cada logueo se agrega una línea que contiene 'END_LOG', que será utilizada para eliminar incertezas debido a posibles caídas durante la escritura. Generalmente al loguear es necesario escribir varias líneas de las anteriormente mencionadas, por lo que son escritas y al final se agrega la línea 'END_LOG' para señalar que dicho conjunto de líneas debe ser leído al completo o no debe ser leído.
Por ejemplo:
Se tiene un DataFragment el cual es procesado, debe loguearse su id y su resultado. Entonces las líneas RECEIVED_ID y RESULT deben leerse en conjunto. Es necesario que sean leídas en conjunto ya que de leer sólo el id, no volveríamos a procesar dicho fragmento pero habremos perdido el resultado. En el log las podemos ver así:

```
END_LOG (correspondiente al log anterior)
RECEIVED_ID (data)
RESULT (data)
END_LOG
```

De esta forma, los grupos de líneas deben estar encapsulados por señalizadores para saber que el log está completo. De lo contrario podríamos encontrarnos con algo así:

```
RECEIVED_ID (data)
RES
```

En donde la última línea está corrupta. De esta forma evitamos leer logs incompletos.

Lectura

Debido a que las líneas más importantes (las que nos permiten omitir otras líneas y evitar levantar información de más) son escritas posteriormente a las que queremos omitir, el archivo se lee desde la última línea hasta la primera. Las líneas no son tenidas en cuenta hasta que no aparece el primer 'END_LOG' (que temporalmente sería el último en ser escrito), de esta forma nos aseguramos no leer líneas corruptas o grupos de logs por la mitad.

Posibles mejoras para la implementación

Al momento de implementar el TP surgieron diversas ideas de mejora que no pudieron ser llevadas a cabo debido a la cercanía con la fecha de entrega o la necesidad de implementar otras cosas. Dentro de las mejoras notamos las siguientes:

Uso de callbacks

En el código actual se usa el método basic_get en conjunto de una pequeña inteligencia al momento de realizar un sleep. De esta manera nos aseguramos que los nodos que estén a la espera de información se encuentren usando un porcentaje de procesamiento tendiente al 0%.
Una mejora para esto es el uso de callbacks, el cual no pudo introducirse debido a la necesidad de acceder a diferentes queues sin importar si alguna de ellas no recibe mensajes. Una opción para poder realizar esto es dividir las tareas de cada nodo en diferentes procesos paralelos, de forma que si alguno no continúa su ejecución, debido a la falta de recepción de mensajes, el resto pueda continuar con el procedimiento de sus tareas.

Eficiencia de memoria al momento de levantar logs

Para poder hacer la lectura de forma inversa hay dos opciones:

- Cargar todas las líneas en memoria y luego invertirlas.
- Leer las líneas una a una e ir moviendo el puntero.

En esta implementación se optó por la primer opción debido a lo siguiente:

- Los logueos fueron minimizados, de forma que las líneas cuentan con la mínima cantidad de información.
- Se harán limpiezas periódicas, de esta forma la cantidad de información a cargar en memoria será equivalente a las líneas del archivo.
- Se contaba con tiempo finito y más features para realizar.

Una mejora es cambiar el algoritmo de lectura por el segundo mencionado. Optimizando así la memoria utilizada.

Eficiencia de tiempo al momento de limpiar logs

Al momento de hacer la limpieza de logs se hace en el proceso principal de cada nodo, permitiendo así que una vez leído el archivo a limpiar no se agregarán nuevas líneas. Sin embargo esta solución demanda un tiempo debido a que pausa el procesamiento de información.
Una idea de mejora es que la limpieza se haga en un proceso paralelo, el cual agregue todos los logs nuevos que se agregaron luego del comienzo de la lectura del archivo original.

Disclaimer

Issue respecto al logueo y recuperación de información correspondiente a la query 4:
Al momento de levantar información correspondiente al filtrado por TOP de la query 4 se observó un error en el cual los datafragments eran logueados como bytes en lugar de como string.
Para buscar solucionar este problema se hizo una modificación en la cual se agregaron los métodos to_str y from_str para concentrar la zona de causa/solución del problema en un único lugar.
Previo a la demo se buscó replicar el error con estos cambios, no volvió a aparecer pero no lo damos por solucionado debido a la escasa cantidad de testeos.