.UBAfiuba
FACULTAD DE INGENIERÍA

Amazon Books Analyzer

Escalabilidad, Middleware, Coordinación de Procesos y Tolerancia a Fallos

Primer cuatrimestre 2024

Apellido y Nombre	Padrón	Email
Buzzone, Mauricio	103783	mbuzzone@fi.uba.ar
De Angelis Riva, Lukas Nahuel	103784	ldeangelis@fi.uba.ar
Adris, Mario Santiago	106870	madris@fi.uba.ar

Índice

1. Alcance

Se pide crear un sistema distribuido que analice las reseñas a libros en el sitio de Amazon y que permita obtener resultado de las siguientes 5 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.

1.1. Cálculos auxiliares

Mostramos cálculos auxiliares que ayudarán a tomar decisiones respecto a los elementos a guardar en memoria.

- Hay cerca de 210.000 libros y 3.000.000 de reseñas a trabajar.
- En promedio los títulos de los libros tienen 50 caracteres.
- En promedio hay 1,30 autores por libro y el nombre de los autores mide en promedio 15 caracteres.
- El 20 % de los libros son de los 90' y el 16 % son de categoría ficción.

2. Vista de Escenarios

2.1. Diagrama de casos de uso

El sistema contempla tres casos de uso principales, marcados en el siguiente diagrama.

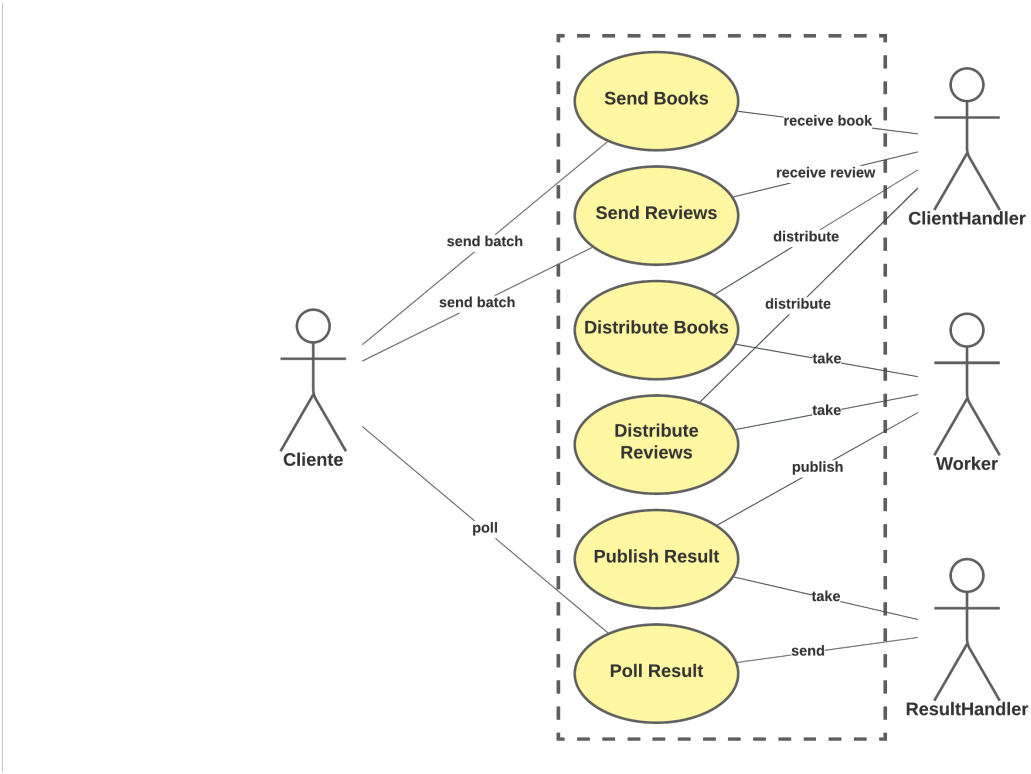


Figura 1: Diagrama de casos de uso

El *cliente* enviará *batches* de libros y reseñas al sistema. Estos serán tomados y distribuidos por el *ClientHandler*. Los workers del sistema tomarán los datos, los procesarán y publicaran los resultados correspondientes.

Dichos resultados serán guardados por *ResultHandler* que se encargará de responderle al cliente cuando este realice un pedido de Poll.

3. Vista Lógica

3.1. DAG

Se presenta el siguiente diagrama DAG mostrando las relaciones entre las distintas actividades. Se puede observar como el procesamiento de las consultas están divididas en etapas con tareas simples e individualmente escalables.

Figura 2: DAG- Amazon Books Analyzer

3.2. Diagrama de Clases

Mas cercano a la implementación se muestran los diagramas de clases de las partes más importantes del sistema.

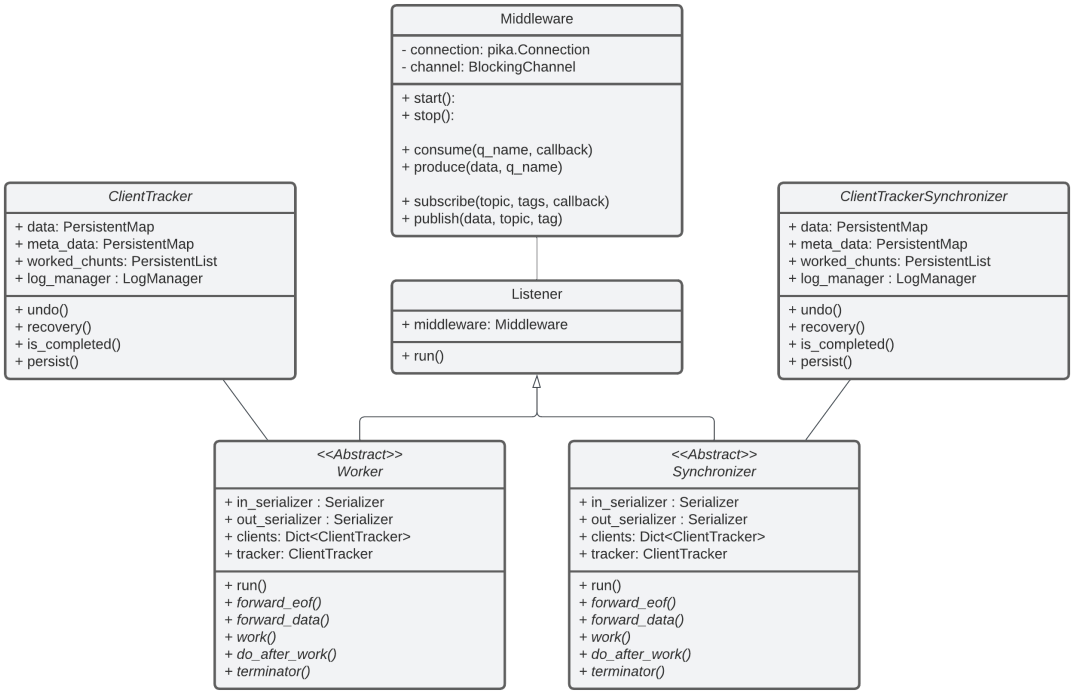


Figura 3: Diagrama de clases Middleware

El middleware encapsulará la comuncación vía rabbit utilizando funciones con un sentido amplio de protocolo de comunicación, como son *produce/consume* y *publish/subscribe*.

Luego, todos los trabajadores tiene la misma lógica principal donde, es por ello que se propone una clase abstracta ***Worker*** que defina el protocolo básico a utilizar en el trabajo, las clases que hereden deberán implementar las funciones de trabajo *work/do after work* y funciones de envío *forward*. Por último será necesario que se defina la función *terminate* para dar un cierre al proceso de cierto cliente. Una idea similar se realizó con los ***Synchronizers***.

Además se agrergaron ***ClientTracker(Sync)*** para el seguimiento de los datos de cada cliente en particular. Este mantiene en memoria y persistida la información relevante para el correcto funcionamiento del protocolo. Mantendrá metadata sobre cuántos paquetes se esperan recibir, si se terminó el procesamiento, etc. Además de los paquetes trabajados y un log manager. Se explicará en detalle más adelante.

4. Vista de Procesos

A continuación se presentan distintos escenarios y como se maneja el flujo de mensajes a través del sistema.

4.1. Flujo de la consulta 1

En el siguiente diagrama se presenta como un batch de libros es enviado para las consulta 1. Se puede observar cómo todos los libros seleccionados por tener las características solicitadas son enviados al **ResultHandler**.

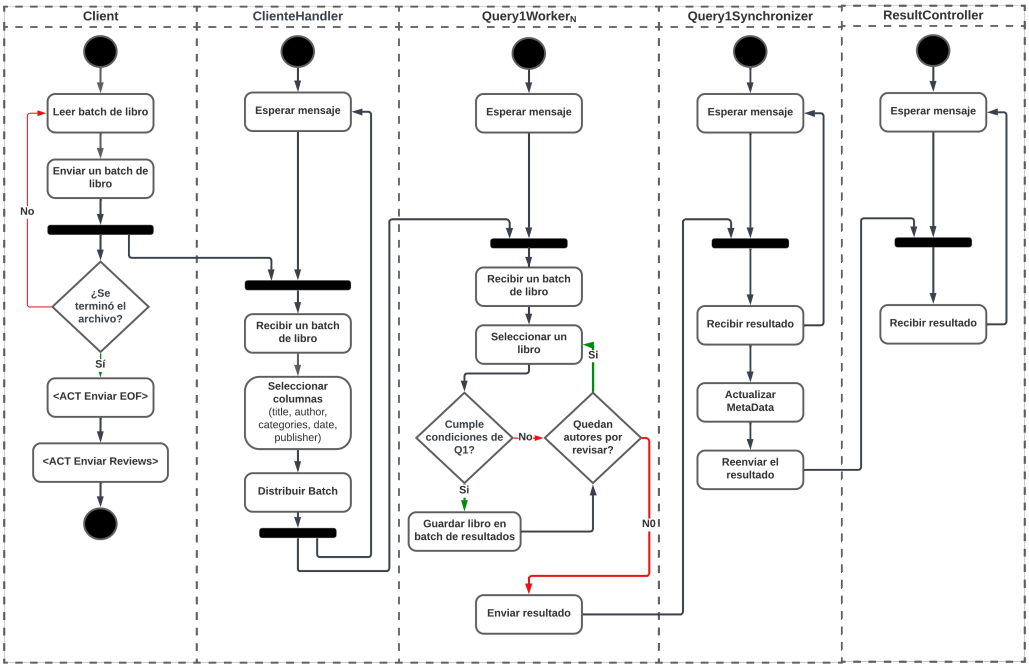


Figura 4: Flujo de mensajes para la Consulta 1

4.2. Flujo para la Consulta 2

En el siguiente diagrama se ve como como funciona el flujo de mensajes para la consulta 2.

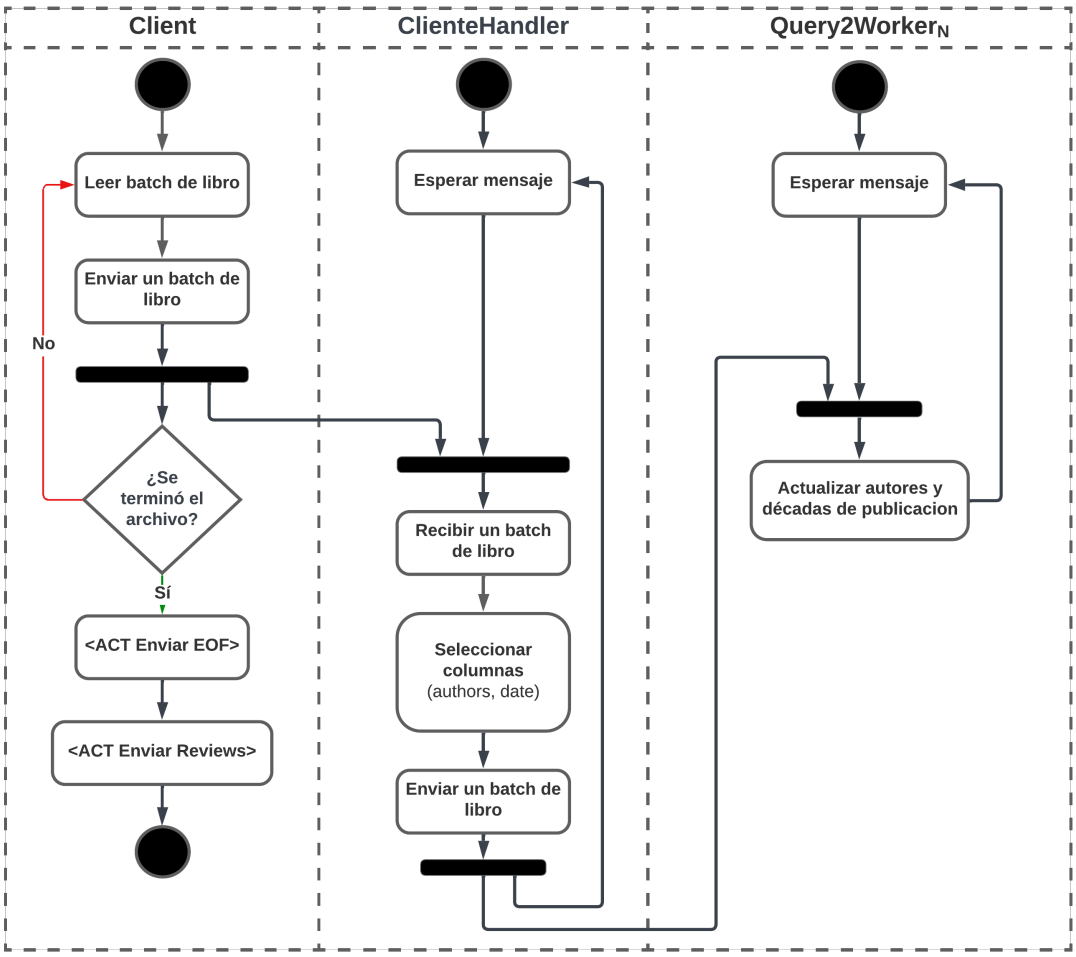


Figura 5: Flujo de mensajes con libros para la Consulta 2

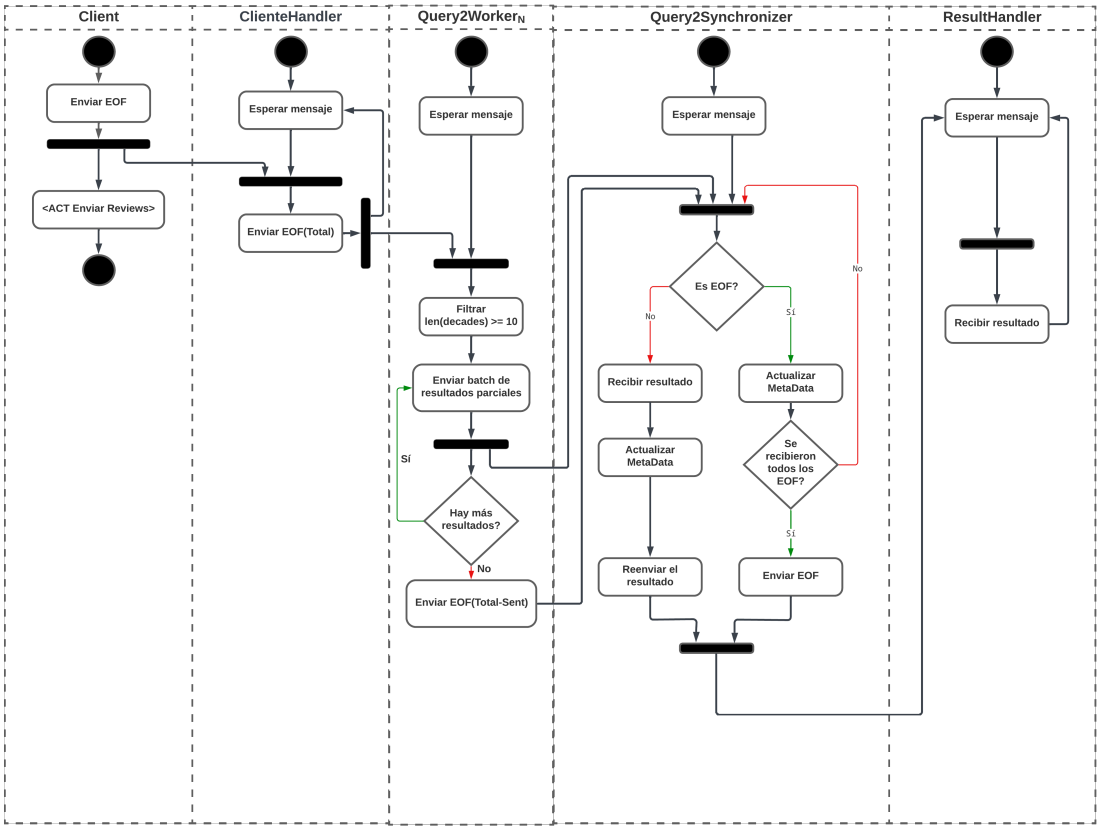


Figura 6: Flujo de mensaje EOF para la Consulta 2

Nota: Las consultas 3, 4 y 5 también hacen uso de N Workers y 1 Synchronizer y por lo tanto los flujos para los mensajes de review serán muy similares al realizado para la consulta 2

5. Vista de Desarrollo

5.1. Diagrama de paquetes

Inicialmente se presenta un diagrama de paquetes para observar la estructura del repositorio.

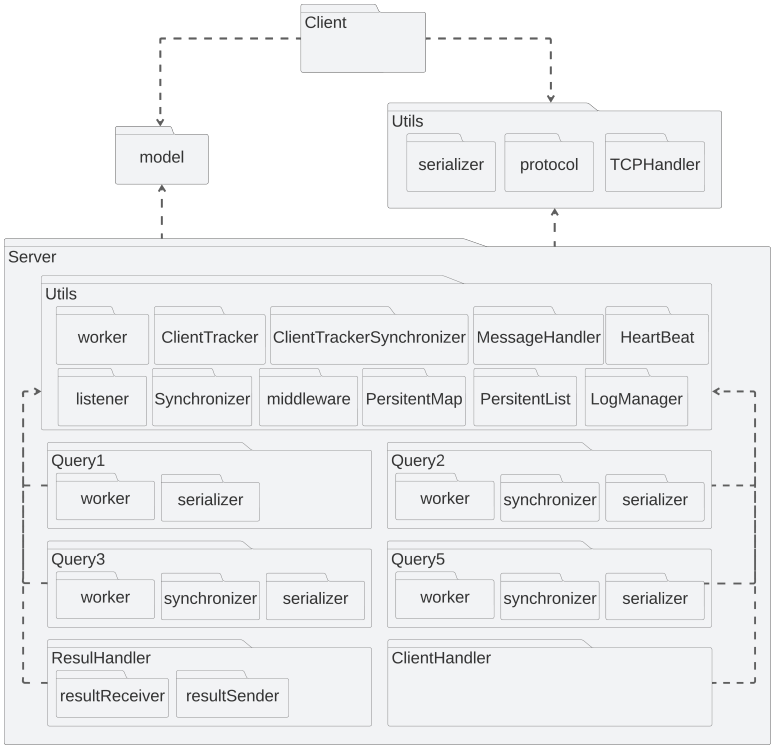


Figura 7: Diagrama de paquetes

5.2. Decisiones de diseño

5.2.1. Protocolo de comunicación

Tanto para la comunicación entre el sistema y el cliente, como para la comunicación dentro del sistema. Desarrollamos nuestro propio protocolo siguiendo una estructura TLV para estructurar los mensajes.

5.2.2. Patrones de diseño

Para implementar todas las consultas, aplicamos el mismo patrón. Donde los libros o reseñas llegan a un único punto de entrada, y este se encargar de distribuir la carga entre los distintos trabajadores. Luego todos estos resultados parciales se juntan en un único punto, que sera el encargado de enviar los resultados a la parte del sistema que se encarga de guardarlos.

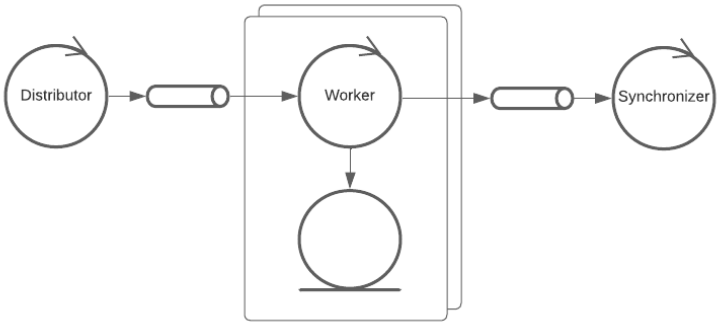


Figura 8: Patrón Distributor-Worker-Synchronizer

En el caso de nuestro trabajo práctico el *Distributor* forma parte del *ClientHandler* y los *Synchronizers* son un nodo aparte.

6. Vista de Física

6.1. Diagrama de Robustez

A continuación se muestra el diagrama de robustez divido para los distintos flujos dentro del sistema.

Consulta 1

En primer lugar se presenta la sección del sistema que incumbe a las consulta 1 (Título, autores y editoriales de los libros de categoría "Computers" entre 2000 y 2023 que contengan "distributed" en su título). Como se puede observar, un **Query1Worker** se encarga de recibir libros que envía el **clientHandler** y envía a un Synchronizador que simplemente acumula los EOFs para sincronizar el protocolo y enviar correctamente los resultados hacia el ***ResultHandler***

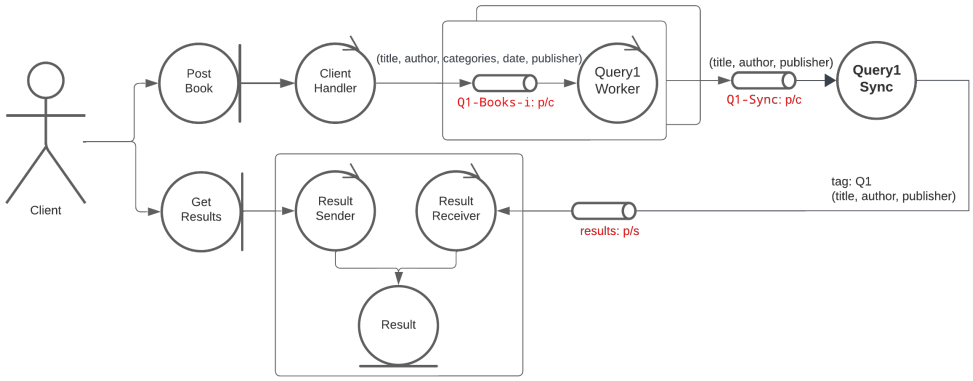


Figura 9: Diagrama de Robustez Q1.

Consulta 2

El segundo caso que presentamos es el de la consulta 2 (Autores con títulos publicados en al menos 10 décadas distintas). En esta ocasión, el procesamiento se realiza en dos partes. Inicialmente

se distribuye el trabajo *shardeado* para calcular aquellos que enviaron en más de 10 décadas diferentes. Tal cual la *Query1*, el *Synchronizer* simplemente coordina a los *Workers* para enviar todos los resultados hacia el *ResultHandler*

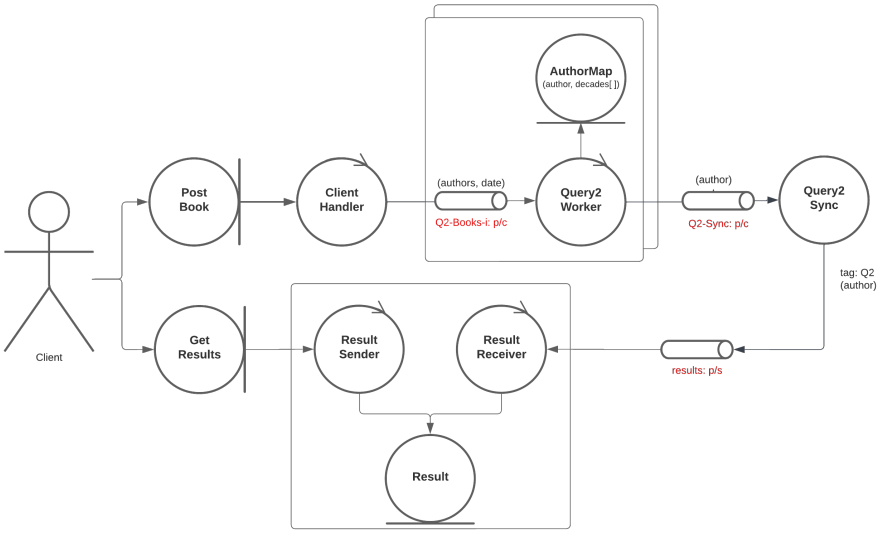


Figura 10: Diagrama de Robustez Q2.

Nota: Se decidió guardar el mapa ***AuthorMap*** en memoria ya que, asumiendo que los autores escriben durante 5 décadas diferentes habrá un total de 273.000 entradas ($210000 \cdot 1{,}30$) de en promedio 35 bytes cada una ($15b + 4b \cdot 5$). Por lo que la cota será de 10Mb

Consultas 3 y 4

A continuación se presenta las consultas 3 (Títulos y autores de libros publicados en los 90' con al menos 500 reseñas) y 4 (Libros con mejor rating promedio entre aquellos publicados en los 90' con al menos 500 reseñas). Para ambos consultas es necesario utilizar tanto la tabla de reseñas, como la de libros. Inicialmente se envían los libros que correspondan a cada trbajador a los **Query3Handler**. Aquellos publicados en los 90' son almacenados y el resto son descartados. Una vez se enviaron todos los libros, se comienza a enviar todas las reseñas de los libros que cada Worker tiene, y ellos se encargarán de actualizar la cantidad de reseñas y el rating para cada libro. Por último, se filtrarán aquellos que posean más de 500 reseñas y se enviarán hacia el Synchronizer, que enviará hacia adelante todos aquellos chunks recibidos (por la Query3) y además almacenará el Top10 para que una vez que obtenga todos EOFS enviarlos hacia el ResultHandler.

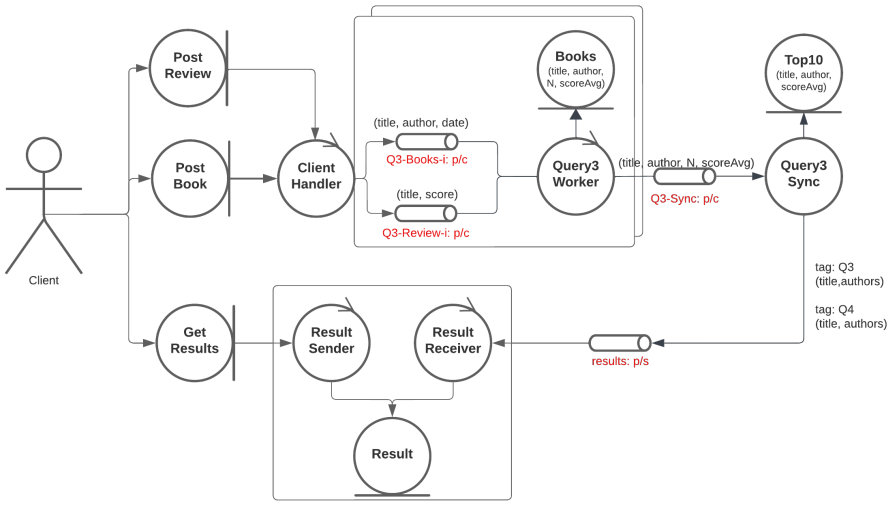


Figura 11: Diagrama de Robustez Q3 y Q4.

Nota: Se decidió guardar el mapa ***Books*** en memoria ya que tendrá un total de 210000 entradas de en promedio 77,5 bytes cada una ($50b + 1{,}3 \cdot 15b + 4b + 4b$). Por lo que la cota será de 15,5Mb

Consulta 5

Por ultimo, se presenta la consulta 5 (Títulos en categoría "Fiction" cuyo sentimiento de reseña promedio esté en el percentil 90 más alto). Al igual que en las consultas anteriores el procesamiento debe realizarse en dos partes. Inicialmente los **Query5Handler** reciben los libros y almacenan solo los que son del genero ficción. Luego para cada reseña de alguno de los libros de ficción, se calcula el sentimiento promedio. Por ultimo se envían estos resultados a un sincronizador que se encargará de juntar la información y obtener el 90 percentíl más alto, para luego enviar los resultados.

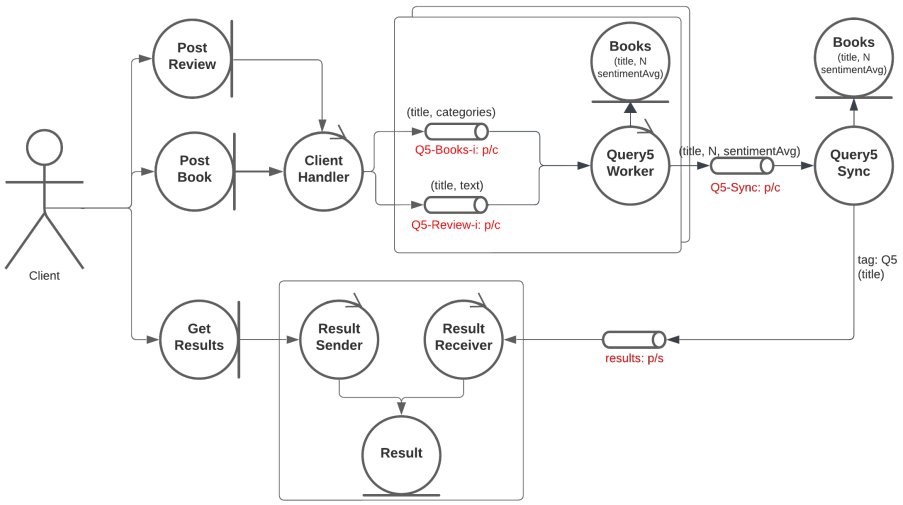


Figura 12: Diagrama de Robustez Q5.

Nota: Se decidió guardar el mapa ***Books*** en memoria ya que tendrá un total de 210000 entradas de en promedio 58 bytes cada una ($50b+4b+4b$). Por lo que la cota será de 11,6Mb

6.2. Diagrama de Despliegue

A continuación se muestran las aplicaciones que se despliegan en el sistema. Como podemos observar, todas ellas se comunican a través del middlerware. Esto quiere decir que el middlerware es un punto único de falla, pero que los distintos componentes no depende entre si.

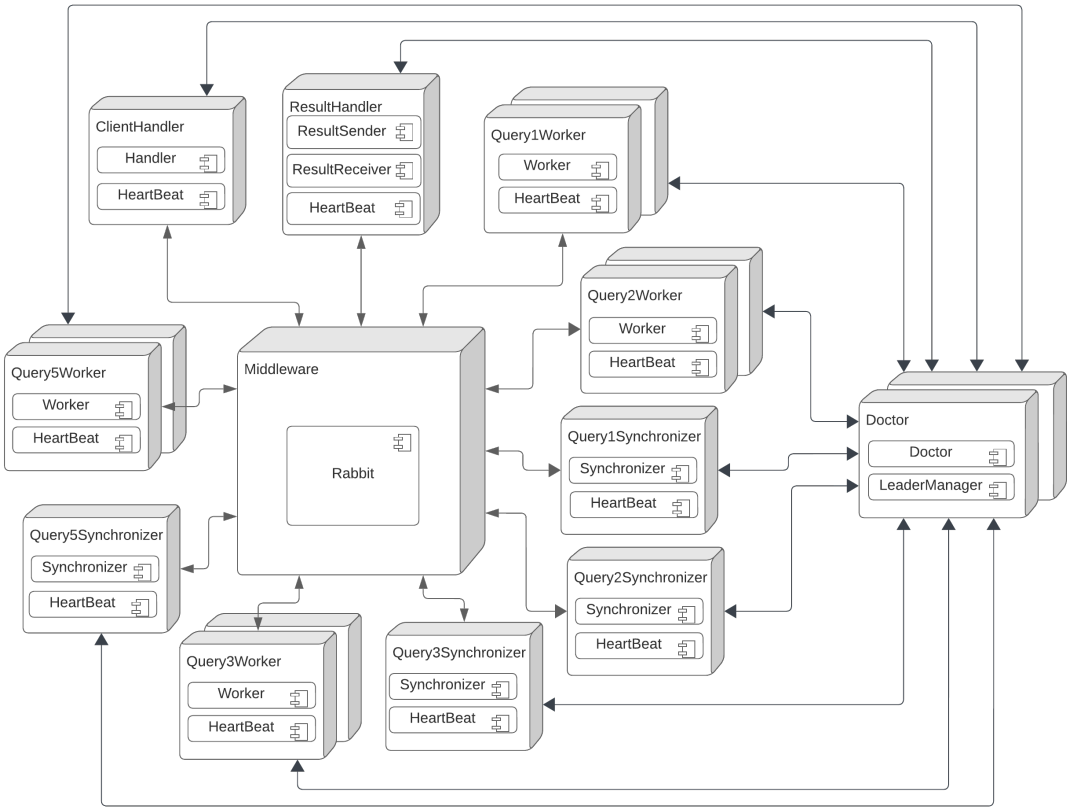


Figura 13: Diagrama de despliegue

7. Tolerancia a Fallos

En esta sección se describe los cambios realizado al sistema para que sea tolerante a fallos. Los requerimientos solicitados son:

- El sistema debe permitir la ejecución de múltiples análisis de clientes en paralelo si ser reiniciado.
- Los clientes deben enviar el conjunto completo de datos a analizar, permitiendo futuras extensiones en las consultas implementadas por el sistema.
- El sistema debe mostrar alta disponibilidad hacia los clientes.
- El sistema debe ser tolerante a fallos por caídas de procesos.
- En caso de usar un algoritmo de consenso, el mismo tiene que ser implementado por los alumnos.
- Se puede utilizar docker-in-docker para levantar procesos caídos.
- No está permitido utilizar la API de docker para verificar si un nodo está disponible.

7.1. Cambios del modelo original

Al considerar que los distintos procesos se pueden caer, algunos algoritmos que se utilizaban anteriormente ya no son validos. De modo que se debieron realizar varias modificaciones.

7.1.1. Shardeo

Una de las modificación que se hicieron al sistema fue que ahora tantos los libros como las reseñas se shardan entre los multiples workers. De esta forma se simplifica el costo de juntar resultados parciales.

- **Query1**: Los mensaje se distribuyen según el titulo del libro.
- **Query2**: Los mensaje se distribuyen según el autor del libro. En esta ocasión como un libro puede tener varios autores, se debe crear una copia del libro para cada uno. Ya que dicha consulta busca saber cuando autores publicaron en mas de 10 décadas.
- **Query3**, **Query4** y **Query5**: Tanto los libros, como las reseñas se distribuyen por libro.titulo. De esta forma nos aseguramos que todas las reseñas de un libro sean enviadas a un único worker, el cual contiene la información de ese libro. De esta forma los resultados que obtenga dicho worker ya serán definitivos y no hará falta juntarlos con los resultados procesados por otros workers.

7.1.2. Manejo de EOF

En primer lugar al ahora shardear la información en los distintos workers ya no consumen mensajes de la misma cola, sino que cada uno tiene una cola propia. De modo que ahora el clientHandler debe multiplicar el EOF de los libros y reseñas a todos los workers que consumieran alguna cola.

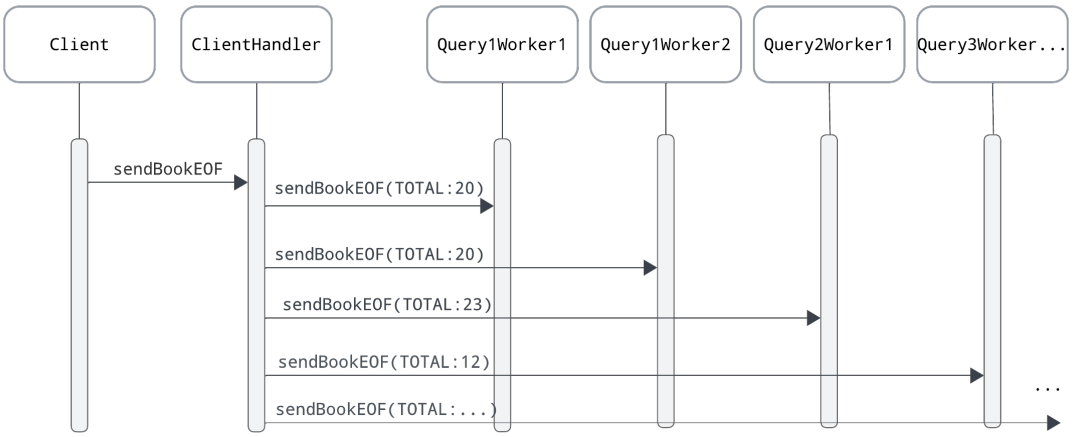
Además ahora como los distintos procesos se pueden caer, no se puede asegurar que el EOF llegara si solo si ya fueron enviados todos los libros o reseñas. Para apelar esta complicación se debe persistir información de cuantos libros o reseñas fueron recibidos.

En el caso del clientHandler debemos llevar la cuenta de cuantos libros envió el Cliente y cuantos se enviaron a cada worker de cada consulta para ese Cliente. Para ello se creo una clase

QueryManager que es la encargada de persistir dicha información y encapsular toda la lógica de como distribuir los libros y reseñas a todos los workers.

Cuando el clientHandler recibe un EOF del cliente envía hacia adelante el mensaje de EOF y agrega como parámetro la cantidad de elementos que debió haber recibido.

Para ilustrar como funciona se presenta un ejemplo donde el cliente envió 40 libros y luego envió el EOF.



En este diagrama se puede ver como el clientHandler envió la mitad de los libros al query1Worker1 y la otra mitad al query1Worker2. Por otro lado vemos que envió un poco mas de la mitad de los libros al query2Worker1, esto debido a que algunos de esos libros tenían varios autores, de modo que se duplicaron para enviar una copia con cada autor. De la misma manera para los Workers de la Query3 y 5.

Para que este algoritmo funcione los workers deben registrar cuantos elementos fueron procesando para luego compararlo con el total que el envía el clientHandler en el mensaje de EOF. Si al llegar dicho EOF, compara y la cantidad de elementos procesados es menor a el total enviado por el clientHandler, es necesario esperar ya que descubre que que le queden mensajes por leer, y también sabe cuańtos. Por otro lado si al comparar la cantidad de elementos procesados es igual a el total enviado por el clientHandler, el Worker tiene la certeza de que ya no quedan mensajes por leer.

De la misma forma que el clientHandler persiste cuanto elementos envió a cada worker, cada uno de ellos también debe persistir cuantos envió hacia los Synchronizer para que este también puede asegurarse de que recibieron todos los mensajes.

Los Synchronizer deben asegurarse de que todos los workers le envíen todos los mensajes antes de enviar el EOF final hacia el resultHandler. Para ello mantiene la cantidad de elementos enviados por cada worker y el total esperado enviado en el EOF por cada worker.

7.1.3. Delegación

Se decidió que en algunas cuestiones, el cliente tenga ciertas responsabilidades, por un lado, simplifican el diseño en algunos aspectos, pero mas importante, aseguran integridad de los datos. Veamos brevemente que responsabilidades tendrá el cliente:

- **Generación de ID de cliente:** Dado que el id de cliente solamente lo utiliza el sistema en un principio se creyó que lo mejor mejor era que el sistema lo generase, pero esto produce ciertas discrepancias, por ejemplo, el siguiente escenario daría un resultado incorrecto:

  ```
  1. Cliente envia datasets
  2. Cliente solicita los resultados
  3. Cliente se desconecta
  4. Cliente solicita los resultados
  ```

  Por otro lado, es lógico que el cliente haga peticiones con sus credenciales, de esta manera el sistema lo puede identificar.

- **Fallo en la comunicación** Ante un fallo en el canal de comunicación (TCP), el cliente es responsable de intentar nuevamente. Ya sea crear un nuevo socket y conectarse al servidor, o reenviar una petición (esto ultimo esta encapsulado en el protocolo)

- **Paginación de resultados:** Relacionado con el punto anterior, al solicitar resultados, el cliente puede pedir que pagina solicitar. Por lo tanto, cuando se caiga el servidor, el cliente reintentara la solicitud de la ultima pagina (que no recibió `ACK`). Una alternativa a utilizar paginación, es `poll`. Es decir, el cliente pide resultados, y el servidor devuelve mensajes del tipo: `RESULT`, `WAIT` (que indica que debe seguir `polleando`), o `EOF`. Por lo que el servidor tiene que implementar lógica para persistir un cursor, que indique cual es el próximo batch de resultados a enviar.

Listing 1: `client.py`

```
i = 0
while True:
    res = get_result(page=i)
    if res == CONN_ERROR:
        # retry on server disconnection or ack not received
        retry(page=i)
    if result.type == DATA:
        save_result(res)
    if result.type == EOF:
        return
    i += 1
```

Listing 2: `server.py`

```
i = 0
while client_alive:
    page = get_result_request()
    res = get_persisted_resukt(page=page)
    send_result(res)
    # if server dies before here, client will retry
```

7.2. Multicliente

El sistema debe poder procesar la información de múltiples clientes al mismo tiempo. Para ello, se desarrollo un ClientTracker encargado de almacenar toda la información respectiva a un cliente en particular.

Un Worker o Synchronizer tiene un diccionario de ClientTrackers y ClientTrackerSynchronizer respectivamente y dependiendo de quien sea el mensaje que deben procesar utilizaran uno u otro para trabajar con el.

Los ClientTrackers y ClientTrackerSynchronizer almacenaran y persistirá la siguiente información:

- worked_chunks: Una lista con los IDs de los chunks procesados.
- meta_data: Un mapa con información como las elementos procesados, los elementos totales esperados, los elementos enviados, etc...
- data: Un mapa con la información que provee el cliente, libros o reseñas dependiendo de la consulta.

7.3. Persistencia y recuperación

Para que un proceso pueda recuperar su estado previo luego de caerse debe persistir información en disco. Para ello se implemento dos objetos **PersistenMap** Y **PersistenList**. Estas dos objetos funciona con los respectivos tdas pero además persiste su información en disco. Además de ello se implemento un sistema de Logs para persistir el ultimo valor antes de una modificación de los datos en caso de que un proceso se caiga en medio de el procesamiento de un mensaje.

7.3.1. Persistencia en el ClientHandler

El **clientHandler** debe almacenar, para cada uno de los clientes, los libros y reseñas enviados a cada worker de cada query y el ID del ultimo chunk recibido. De modo que persistirá las siguiente información **para cada cliente**.

```
total_books = {
    "Q1": {"1": 0,...,"nQ1": 0}
    "Q2": {"1": 0,...,"nQ2": 0}
    "Q3": {"1": 0,...,"nQ3": 0}
    "Q4": {"1": 0,...,"nQ4": 0}
    "LAST_CHUNK": "20b2884a..."
}
```

```
total_reviews = {
   "Q3": {"1": 0,...,"nQ3": 0}
   "Q5": {"1": 0,...,"nQ5": 0}
   "LAST_CHUNK": "34c2f4e1..."
}
```

Cuando el **clientHandler** debe enviar un nuevo mensaje de por ejemplo libros a los workers, sigue los siguientes pasos:

```
a. Para cada consulta consulta Qx:
    1. Shardea la libros entre todos los workes.
    2. Para cada worker de la consulta Qx
        1'. Actualiza el nuevo total de libros enviados por el cli.
        2'. Envia los nuevos libros.
b. Actualizo LAST_CHUNK
c. Flushea total_books a disco.
d. Envio ACK al cliente.
```

Si el proceso se cae antes de hacer el flush a disco, no enviara el ACK al cliente. De modo que este volverá a enviarle el mismo mensaje y el **clientHandler** lo replicara a los workes. Si bien se envían mensaje duplicados luego son filtrados por los workers.

En caso de que el **clientHandler** se caiga luego de hacer el flush a disco, no enviara el ACK al cliente. De modo que este volverá a enviarle el mismo mensaje y el al recibirlo podrá compararlo con el LAST_CHUNK y automáticamente hacerle ACK sin reenviar el mensaje a los workers.

7.3.2. Persistencia en el Workers

Los workers contara con un sistema de Logs que le permita volver al estado anterior si se cae en medio del procesamiento de un mensaje. Utilizando un **LogManager** que se encargara de generar un archivo de log con los valores anteriores a manejar el mensaje. Se guarda los valores viejos ya que se utilizara el método de recuperación **UNDO**.

```
BEGIN;20b2884a-732e-47e8-be2d-4d5c55b374c8
META;WORKED;200
META;EXPECTED;-1
META;SENT;0
WRITE;tittle1,old_data1
WRITE;tittle2,old_data2
COMMIT;20b2884a-732e-47e8-be2d-4d5c55b374c8
```

En caso de que el Worker se caiga durante el procesamiento del mensaje somos capaces de restaurar los valores anterior y manteniendo la consistencia. Haciendo que la operacion que quedo inconclusa no tenga efecto.

Para ello el algoritmo para persistir la información es el siguiente:

```
Persistir Info()
1  Logear BEGIN chunk_ID:
2. Si se guardan datos
        Flushear data a disco.
3. Flushear meta_data a disco
4. Flushear chunkID a disco.
4. Logear COMMIT chunk_ID
```

Si al levantar devuelta el Worker la ultima linea del log no es el COMMIT, debemos restaurar todos los valores por los indicados en el log. De este modo, se deshace todo lo que se modificado por el mensaje que se proceso a medias.

- **Query1**:

  Los trabajadores de la consulta 1 deben almacenar los IDs de los chunks ya procesados (filtrado de repetidos) y un mapa con los metadatos **para cada cliente**.

```
worked_chunks = [
    20b2884a-732e-47e8-be2d-4d5c55b374c8
    ef6c146a-7b94-4447-b0a2-24c4f8ae4be4
    82d776fd-0d49-4cd4-ba75-628b249bfea7
    2f6b08fe-4421-4f0d-9933-368dcbc9b19
]
```

```
meta_data = {
    "WORKED": 200
    "EXPECTED": -1
    "SENT": 0
}
```

Inicialmente los valores de metada se setean en 0 para WORKED y SENT y -1 para EXPECTED.

Cuando el Worker recibe un chunk de libros sigue los siguientes pasos:

```
a. Recibe un chunk de libros.
b. Deserializa el chunk.
c. Por cara libro.
    Si cumple los requisitos.
        Se guarda en una lista de resultados
d. Envia la lista de resultados hacia delante.
e. Actualiza los valores de WORKED y SENT.
f. Persistir Info.
g. Envio ACK a rabbitMQ
```

En este caso se presentan varios escenarios importantes:

1. Caerse antes de flushear la metadata:
   En este caso como todas las modificación fueron en memoria no habrá registro de que se trabajo el mensaje y se volverá a procesar una vez levantado. Si bien se envían mensaje duplicados luego son filtrados por los Synchronizer.
2. Caerse después de flushear la metadata pero antes de flusher el worked_id:
   En este caso el Worker vera que el Log no esta completo, de modo que comenzara el proceso de **UNDO**, dejando su estado en el inmediato anterior a recibir el mensaje y volverá a recibir mensaje como si no hubiera ocurrido nada.
3. Caerse antes de flusher el worked_id :
   Cuando el worker vuelva a levantarse y el mensaje le vuelva a llegar, automáticamente le hara ACK

- **Query2**:

Los trabajadores de la consulta 2 aplica los mismo conceptos que los workers de la consulta 1, la única diferencia es que estos trabajadores si almacenan información, particularmente las décadas de publicación para cada actor.

```
data = {
    "title1": [2000,2010,2020]
    "title2": [1950,2010]
    "title3": [1980,1990,2000]
    "title4": [1990]
}
```

Lo que diferencia a estos workers de los de la consulta 1 es que envían los resultados una vez recibidos todos los autores. Una vez esto ocurra, comenzaran a mandar al synchronicer los autores con mas de 10 décadas.

Para estos nuevos mensajes reutilizaran los ids de los chunks que fueron trabajando. Para asegurarse de que aun callándose mientras envían resultados, los IDS serán los mismo y así el synchronizer los podrá filtrar.

- **Query3** y **Query5**:

  Como estos dos tipos de trabajadores deben almacenar los libros, deben guardar metadata extra. Para poder saber si ya fueron enviados todos los libros y pueden comenzar a consumir reseñas.

```
meta_data = {
    "WORKED": 0
    "EXPECTED": -1
    "SENT": 0
    "N_BOOKS": 0
    "ALL_BOOKS_RECEIVED": false
}
```

  Luego tanto para recibir como para enviar un mensaje funcionan similares a los workers de las consultas 1 y 2.

7.3.3. Persistencia en el Synchronizer

Los Synchronizer funcionan de una manera similar a los antes mencionados. La diferencia radica en que los mensajes que recibe son enviados por múltiples workers y debe asegurarse de a ver recibido todos los mensajes de todos los workers. La metadata de los synchronizer inicialmente es la siguiente:

```
meta_data= {
    "WORKED_BY_WORKER":{
        "Q1": {"1": 0, ... ,"nQ1": 0}
        "Q2": {"1": 0, ... ,"nQ1": 0}
        "Q3": {"1": 0, ... ,"nQ1": 0}
        "Q5": {"1": 0, ... ,"nQ1": 0}
    }
    "TOTAL_BY_WORKER": {
        "Q1": {"1": -1, ... ,"nQ1": -1}
        "Q2": {"1": -1, ... ,"nQ1": -1}
        "Q3": {"1": -1, ... ,"nQ1": -1}
        "Q5": {"1": -1, ... ,"nQ1": -1}
    }
}
```

Los Synchronizer de la consulta 1 y 2 funciona de igual manera que los workers de la consulta 1, ya que no necesitan guardar información, ellos solo redirigen el mensaje hacia el resultHandler. Además de persistir la cantidad de elementos totales a trabajar y la cantidad de elementos trabajados hasta ahora, para poder enviar el EOF final de la consulta.

El Synchronizer de la consulta 5 funciona de igual manera que los workers de la consulta 2, ya que debe guardar los resultados que envian los workers para poder calcular el 90percentil.

El Synchronizer de la consulta 3 es quiezas el mas complejo de todos ya que realiza las acciones de ambos Synchronizer anteriores. Por un lado cada vez que le envían un resultado, lo redirige al resulHandler, pero además lo persistir en su PersistenMap de data para una vez recibidos todos los mensajes de todos los workers poder calcular el top10 de libros con mas reseñas.

7.3.4. Persistencia en el ResultHandler

El **ResultHandler** es el encargado de persistir los resultados finales y esperar a que el cliente los solicite. Por lo tanto lo mas importante que debe almacenar nos dichos resultados.

Los cuales los son almacenados en un sistema de paginación. Por un lado estará el archivo con resultados, el cual tendrá como nombre el uuid del cliente dueño de esos datos. Y por otro tendrá un archivo de punteros a las distintas partes del archivo.

Además tendrá que persistir en un PersistentMap la cantidad de resultados totales para cada query, los cuales fueron enviados por los distintos Synchronizer.

```
"TOTAL_BY_WORKER": {
    "Q1": -1
    "Q2": -1
    "Q3": -1
    "Q5": -1
}
```

Cuando el ResultHandler recibe un mensaje los maneja de la siguiente manera:

```
a. Deserializar los resultados.
b. Persistir en el archivo de punteros el tamaño actual del archivo para realizar
    el protocolo de Polling.
c. Agregar a la lista de resultados todos aquellos que aún no se encuentran en
    dicha lista y persistir la lista en disco.
```

De esta manera se resguarda la información, en caso de que el nodo se caiga persistiendo el archivo de punteros, al ejecutar la lógica de recuperación se analizará el tamaño del archivo y si no resulta múltiplo de 4 se trucará.

Si el programa se cae agregando resultados a la lista, se eliminará aquel resultado mal escrito (que no termine en salto de página) y se encontrán persistidos todos aquellos que lograron escribirse correctamente. En este caso, todos aquellos escritos formarán parte de la página de resultados, mientras que los que se trabajarán más adelante (los que no fueron escritos) formarán parte de otra página. De esa manera:

1. Tener que eliminar todos los resultados escritos hasta el momento.
2. Se realiza la menor cantidad de accesos a disco.

En caso de que el nodo se caiga luego de agregar todos los resultados pero antes de realizar ACK al Middleware, el chunk volverá a ser procesado y no se escribirán nuevos resultados ni se actualizará el mapa de punteros.

7.4. Elección de líder

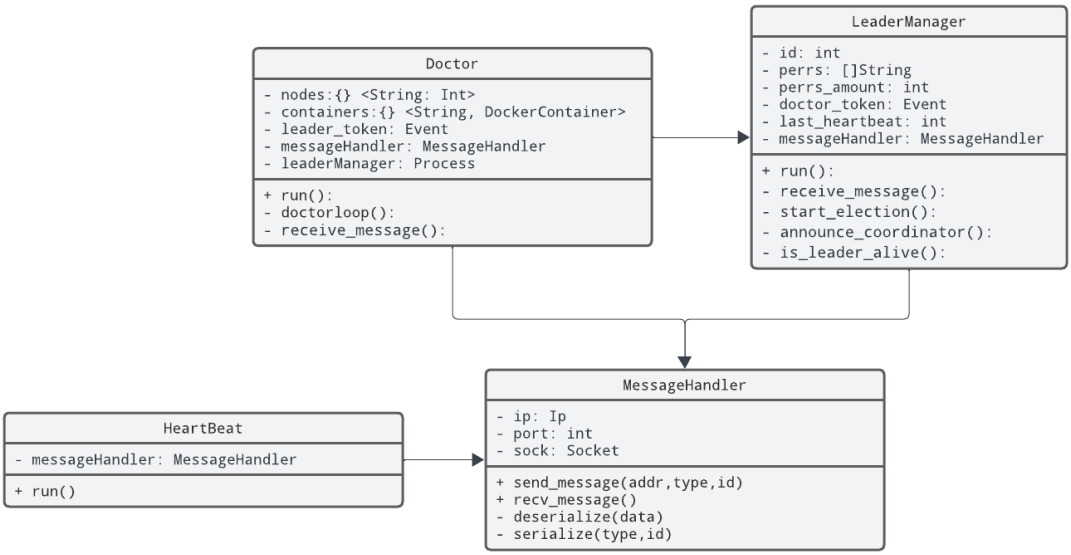
Para garantizar la disponibilidad del sistema se necesita un proceso que se encarga de detectar si otro esta caído y en caso de ser así poder reiniciarlo. Este proceso es llamado **Doctor**.

El principal problema incurre en que el **Doctor** también es vulnerable a caídas. De modo que para poder seguir garantizando la alta disponibilidad, el sistema debe contar con varias replicas de estos doctores.

Como el sistema cuenta con varios doctores, entre ellos deben coordinarse para que solo uno realice la tarea de doctor. Para ello se implementó el **algoritmo bully**.

Para separar las distintas funcionalidades se crearon varias clases:

- Doctor: Es el encargado de enviar un mensaje a todos los nodos del sistema y medir el tiempo de respuesta. En caso de que el tiempo de respuesta pase un cierto umbral, se considera que esta caido y lo levantara utilizando `docker-in-docker`.
- LeaderManager: Es el encargado de la coordinación con el resto de los nodos doctores para determinar quien es el líder. Es el que permite, en caso de ser lider, al doctor realizar el health check.
- MessageHandler: Encapsula el protoclo de mensaje por UDP y maneja el socket.
- Heartbeat: Escucha mensaje del doctor y reenvia el mismo mensaje como respuesta inmediata. Es utilizado por todos los nodos del sistema para responder a los health check.



Como se puede observar todas las clases utilizan MessageHandler para comunicarse. Además, también se pueden observar que tanto el doctor como el LeaderManager, tiene un atributo llamado leader_token y `doctor_token` respectivamente, que es una CondVar que permite a LeaderManager bloquar al doctor hasta que este sea el lider.

7.4.1. Algortimo Bully

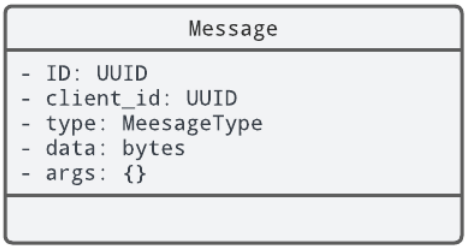
El algoritmo bully fue implementado utilizando mensaje UDP. Los mensajes que se envían contienen con un byte que indica el tipo y en el caso de COORDINATOR también contiene un byte con el ID del lider.

MessageType
ELECTION
OK
COORDINATOR
LEADER

Los mensajes de ELECTION, OK, COORDINATOR son los mensajes clásicos del algoritmo bully. Fue agregado el mensaje LEADER, el cual es enviado por el líder para confirmar que sigue activo al resto de los doctores. En caso de que el resto de los doctores no reciban dicho mensaje, luego de un plazo de tiempo lo declaran muerto y comenzaran una nueva elección.

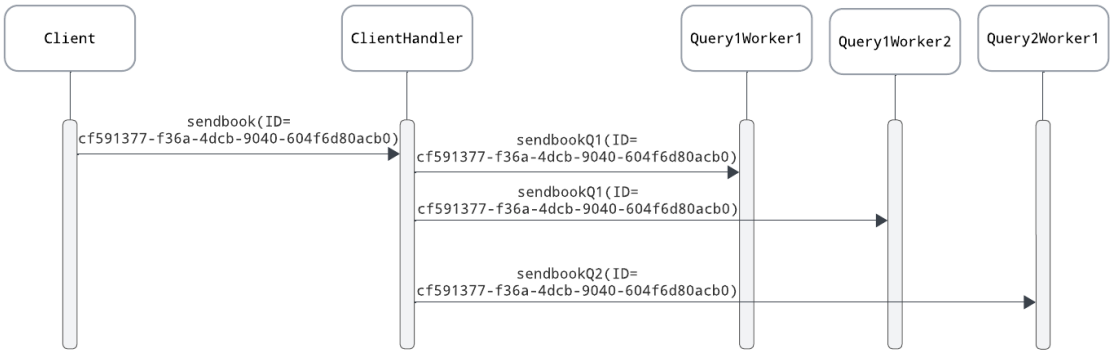
7.5. Filtro de mensajes duplicados

Dado que el sistema puede tener nodos caídos, se pueden llegar a presentar mensajes duplicados. Para no procesar dos veces la misma información, se implemento un sistema de `IDs` y un registro de los mensajes ya procesados. Un mensaje tiene los siguiente campos.

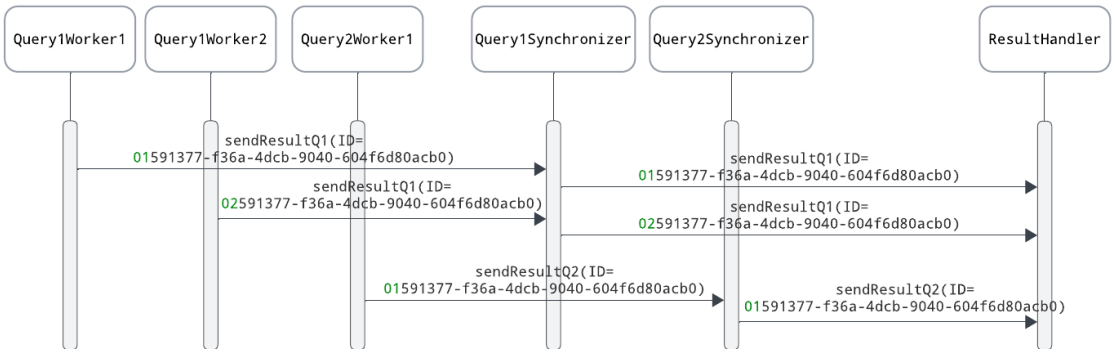


Para garantizar que los ids de los mensajes siempre sean los mismos para la misma información, estos ids son generados por el cliente.

A continuación se muestra un diagrama de como se envían un chunk de libros.



Como se puede observar el cliente envía un batch de libros al sistema y genera un uuid para dicho paquete. Una vez llegado al `clienteHandler`, es distribuido a las distintos workers de las distintas consultas con la información especializada para cada una de ellas. Además en el caso de que existan varios workers para la misma consulta la información es *shardeada* de modo que el mensaje es partido en `N` pedazos, siendo `N` la cantidad de workers de esa consulta en particular.



Una vez que cada worker termine de procesar un mensaje pueden ocurrir dos cosas:

- El worker envia directamente los resultados a la siguiente etapa:

  En este caso se envían los resultados con el mismos ID del mensaje original pero con una modificación. A este se le modifica el primer byte y se graban allí el ID del worker. De esta forma nos aseguramos que a la siguiente etapa no lleguen dos mensajes con el mismo ID que fueron procesados por distintos workers

- El worker debe esperar a recibir todos los libros y reviews antes de poder enviar resultados:

  En esta ocasión cuando ya tenga toda la información, el worker reutilizara los IDs de mensajes que ya le fueron enviados para enviar los resultados. De igual manera se modificara el primer byte de esos chunks marcando qué worker trabajó ese mensaje.

Por último, los mensajes llegan al *ResultHandler* quien no realiza verificación de repetidos según ID, sino que verifica para cada uno de los resultados en un mismo CHUNK si estos fueron escritor en el listado de resultados. Cada resultado en un mismo chunk puede o no haber sido agrergado previamente, ya que como se viene comentando desde el comienzo del informe, los nodos se pueden caer a la mitad de una escritura por ejemplo. En ese caso se verifica la última línea del archivo de resultados y se verifica si fue escrita por completo (si termina en salto de línea). En caso de no haber terminado en salto de línea simplemente se elimina dicho resultado del conjunto.

Cuando el chunk sea nuevamente recibido, los primeros elementos del mismo se encontrarán en el archivo y los nuevos simplemente se escribirán.

Conclusión sobre el filtrado de mensajes:
Esta implementación tiene como pro que el sistema no debe generar nuevos `IDs` para los mensajes que envía, salvo casos puntuales (EOFs). Ni tampoco debe agregar información extra de MetaData en los mensajes.

Pero tiene como contra que, al modificar sólo 1 byte del ID, se restringe el máximo de workers en una query a 256. Una solución rápida sería simplemente agregar en un header, un identificador único (no automáticamente generado) a cada nodo del sistema y enviar dicho identificador como MetaData en los mensajes.