Trabajo Práctico - Tolerancia a Fallos

Integrantes:
Erick Martinez - 103745
Miguel Angel Vasquez Jimenez - 107378

Sistemas Distribuidos
75.74

1C 2024

Introducción

Se presenta a continuación la documentación que cubre la solución del trabajo práctico de Tolerancia a Fallos para la materia Sistemas Distribuidos.

El mismo consiste en la continuación del desarrollo de Distributed Books Analyzer, un sistema distribuido escalable que permite realizar procesamiento de una extensa cantidad de datos, provenientes de datasets de libros y reseñas publicados en la plataforma de Amazon, con fines de análisis y aprovechamiento de multicomputing para la realización de consultas al sistema, con el objetivo de proponer campañas de marketing según los resultados obtenidos.

El objetivo principal es desarrollar requerimientos adicionales a partir del Trabajo Práctico Escalabilidad, que permiten que el sistema sea tolerante a fallos, particularmente a caída de servicios particulares de los diferentes nodos del sistema que albergan a los distintos procesos controladores del mismo. Adicionalmente el sistema se adapta para soportar las peticiones de más de un cliente de forma concurrente, con lo cual se presentan las adaptaciones realizadas para lograr estos y varios objetivos adicionales.

Ejecución

Se brinda un archivo `Makefile` donde se definen los targets principales para la ejecución y monitoreo del sistema. A su vez se provee de un script `create-compose.sh` que parametriza la cantidad de workers, health checkers y clientes a ser ejecutados, el cual sobreescribe el archivo `docker-compose.yaml` apropiadamente.
Se pueden encontrar los targets relevantes en el archivo `README.md` del repositorio. Allí también se encuentran instrucciones para ejecutar un proceso `killer` encargado de forzar dadas de baja a un porcentaje de nodos cada cierto intervalo de tiempo.

Hipótesis / Supuestos

Se toman en cuenta los siguientes:

- El proceso servidor es el punto de entrada y salida del sistema distribuido.
- Los nodos health checker pueden ser dados de baja (por medio del script killer), sin embargo nunca se dan de baja todos a la vez; como mínimo uno de ellos debe permanecer activo.
- El sistema no es elástico, es decir que no permite integrar nodos nuevos una vez que ya está en marcha dado que requiere un debido ordenamiento sincronizado para su inicialización.
- El middleware RabbitMQ no termina su ejecución ni corta su conexión con los nodos, exceptuando cuando se da de baja el sistema distribuido.

Implementación de la Capa de Middleware

Se mantuvo la implementación de la iteración anterior, no sufrió cambios:

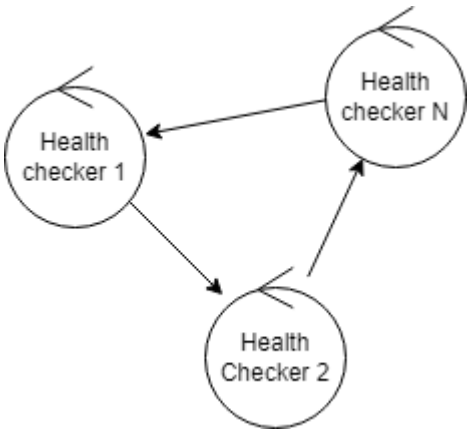
> *Se implementó de forma común la capa encargada de inicialización de conexiones, envío y recepción de mensajes del middleware. Esta capa provee una interfaz para abstraer a los distintos procesos sobre la declaración de colas y exchanges, soportando múltiples topologías de las mismas dependiendo de lo requerido por cada proceso.*
> *Para hacer consistentes dichas definiciones se decidió declarar todo exchange como de tipo directo (ya que no imposibilita el uso como fan-out para el caso que lo requiriese). Como cada uno de los procesos se comporta (en su gran mayoría) tanto como consumidor como productor de mensajes, se tomó la decisión de declarar los bindings de colas desde el lado del proceso que toma el rol de productor de otro. De esta manera se evita cualquier pérdida de mensajes que puede ocurrir de no hacerse de esta forma, dada la naturaleza concurrente del sistema al inicializar el mismo.*

Adaptaciones realizadas

A continuación se detallan los cambios que son abordados para cumplir los nuevos objetivos.

- **Adecuación del sistema actual para múltiples clientes:**
  El Trabajo Práctico Escalabilidad no permitía tener varios clientes realizando solicitudes concurrentes al sistema. Se adecua el protocolo de comunicación entre clientes y servidor, la estructura general de los mensajes y el procesado de los mismos en cada controlador para soportar esta funcionalidad. A su vez se añade soporte a desconexiones (solo por parte de los clientes) durante la ingesta de datos. El servidor maneja las mismas propagando al sistema mensajes apropiados para limpieza de datos del cliente desconectado.

- **Tolerancia a fallas:** Se divide en los siguientes hitos:
  - **Desarrollo de controladores de monitoreo "Health Checkers"**:
    Encargados de solicitar periódicamente el estado de cada uno de los demás controladores del sistema. Para estos últimos se realiza adaptación para recibir dichas solicitudes por medio de herencia de la clase `MonitorableProcess`.
    Los health checkers son procesos *stateless*, por lo tanto son fácilmente replicables y no realizan persistencia.
    Cada controlador del sistema distribuido es monitoreado por un sólo health checker, con esto se evita race-conditions al intentar levantar algún controlador caído. Esto también aplica para los propios Health checkers, por lo que diseñó una relación circular entre ellos, de tal forma que si se cae uno de ellos sólo es posible que sea detectado y levantado por el anterior. Tall cómo se puede observar en el siguiente gráfico:



De esta manera el sistema puede seguir funcionando siempre que haya un Health Checker vivo, ya que con esto se podrán ir levantando los demás junto con sus procesos monitoreados de forma circular.
El monitoreo se realiza por medio de mensajes directos (utilizando la abstracción `SocketConnectionHandler` que encapsula una conexión `tcp`) desde los health checkers, que deben ser respondidos con un mensaje de tipo `ALIVE` (se detallarán los cambios aplicados a los mensajes del sistema más adelante). Los health checkers llevan a cabo esta revisión con un subproceso por nodo a monitorear, cada cierto intervalo de tiempo definido.

- **Persistencia de datos:** Como se prevé que los controladores pueden ser forzosamente interrumpidos, se realiza checkpointing periodico. Esta persistencia se realiza a manera de escritura atómica de forma que si un controlador se cae se tenga por lo menos un punto de partida anterior para su estado interno. Se prioriza la integridad de los datos de los clientes, por lo tanto todos los controladores persisten por lo menos su información relacionada a mensajes previamente procesados (sobre esto se entrará más a detalle en la sección *Comunicaciones* del informe). Esto habilita la posibilidad de existencia de mensajes duplicados, dado que en el peor de los casos un checkpoint puede contener información no tan reciente.
  Lo anterior da lugar al siguiente inciso.

- **Detección de mensajes duplicados**: Para mantener consistencia en los resultados de las consultas realizadas por los clientes se adaptan

los controladores del sistema para evaluar si ya procesaron un mensaje anteriormente y detener propagación cuando sea necesario (también se entrará a detalle más adelante).

- **Reducción de particionamiento de datos innecesario:** En la iteración anterior, existían controladores que durante el procesado de un batch enviaban mensajes conteniendo cargas muy pequeñas de datos (por lo general una sola review/libro). Esto generaba un volumen de mensajes enviados en magnitudes mucho mayores al de mensajes recibidos, lo cual no presentaba problemas dado que no se tenía presente ninguna consideración de falla. Adaptando el sistema se notó considerablemente un cuello de botella en cada uno de estos controladores, por lo cuál se optó por hacer consistente la propagación de datos, de tal forma que todo el sistema en conjunto realice siempre envío de batches.

- **Aumento de disponibilidad:** Se busca aumentar la cantidad de controladores que pueden ser escalables con más workers independientes, de forma que el sistema pueda minimizar downtime y mantenerse operable dentro de lo posible sin incurrir en demoras muy significativas en obtención de resultados. En particular se escaló el proceso Sentiment Analyzer, que realiza acumulación de datos similar a la de los controladores de tipo Counter. Cabe aclarar que otros candidatos a replicación eran los preprocesadores, dado que cumplen los requisitos para ello, sin embargo no se llegaron a escalar exclusivamente por cuestiones de tiempo, ya que se priorizó el correcto funcionamiento del resto de adaptaciones.

Comunicaciones

A continuación se detallan decisiones relacionadas a la comunicación entre los procesos del sistema distribuido.

La comunicación de los procesos con el middleware RabbitMQ permanece en general tal como en la iteración anterior; el envío de ACKs hacia el mismo se realiza de forma manual y al final de todas las operaciones tras recibir un mensajes, para asegurar que se pudo enviar al siguiente nodo y guardar el estado actual del controlador antes de confirmar que se puede remover un mensaje de forma segura de una cola.
Un detalle que fue agregado es la configuración de parámetros adicionales como `confirm_delivery` y el uso en conjunto de los flags `prefetch_count` y `mandatory` especificados en la documentación de RabbitMQ para tener garantías sobre cantidad de mensajes permitidos a consumir por cada proceso previo a realizar su correspondiente ACK (asegurando así que si un controlador no logra realizarlo antes de caerse, el mensaje permanezca encolado para consumo futuro); y sobre la publicación de mensajes a colas sin drop silencioso de mensajes (comportamiento por defecto del broker en casos borde de falla de envío).
Se hará referencia a la utilidad de estos supuestos brindados por el middleware un poco más adelante **[*]**.

En cuanto a la comunicación entre procesos en sí, se continua utilizando un protocolo de texto, sin embargo se vio afectado por cambios importantes:

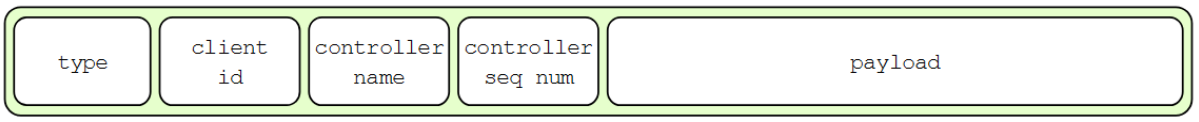
- Estructura:
  Se modela el formato de los mensajes de la siguiente manera:
  Se tienen dos abstracciones distintas de mensajes para aislar la lógica de comunicación entre cliente-servidor (`QueryMessage`) de la de uso interno del sistema distribuido (`SystemMessage`).

❖ `QueryMessage:`

Se introducen los siguientes campos:

- `type`: Utilizado para identificar el tipo del mensaje. Los mismos pueden ser: `DATA_B`, `DATA_R`, `DATA_ACK`, `EOF_B`, `EOF_R`, `WAIT_FOR_SV`, `CONTINUE`, `SV_RESULT`, `SV_FINISHED`.
- `client_id`: Utilizado para identificar a qué flujo corresponde cada mensaje y tener una adecuada separación de los datos obtenidos por los múltiples clientes. También se utiliza para tener un mapeo IP:identificador para manejo de desconexión por parte de los clientes.
- `payload`: Contenido del mensaje. Cuando se trata de datos del cliente, se utiliza un formato similar al de csv para este campo.

❖ `SystemMessage:`



Cuyos campos son:

- `type`: En este caso pueden ser: `DATA`, `EOF_B`, `EOF_R`, `HEALTH_CHECK`, `ALIVE`, `ABORT`,
- `client_id`: Utilizado por los controladores para identificar a qué flujo corresponde cada mensaje y tener una adecuada separación de los datos obtenidos por los múltiples clientes.
- `controller_name`: Utilizado por los controladores para identificar exactamente desde cuál de las réplicas de su predecesor están recibiendo el mensaje. Se emplea junto con la lógica de detección de mensajes duplicados.
- `payload`: Contenido del mensaje. Cuando se trata de datos del cliente, se utiliza el mismo formato que los `QueryMessages`. Se le adjunta un header especial cuando se trata de controladores `sink` para que los clientes puedan

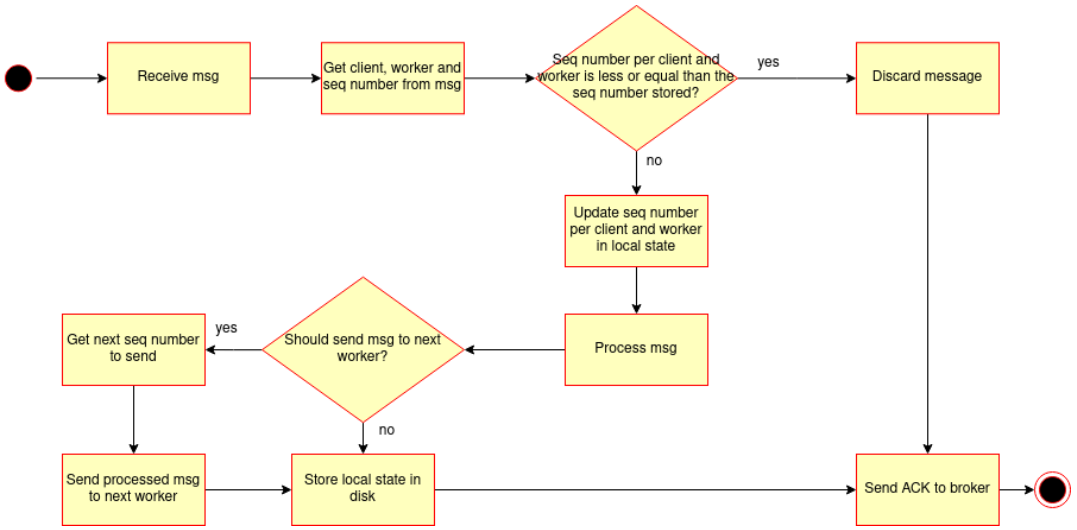
diferenciar resultados según por consulta y almacenarlos de forma separada.

- Mensajes duplicados:

  Se emplea entonces el uso de un modelo de comunicación con la propiedad "*at-least-once delivery*", de tal forma que se prioriza la integridad de los datos que envían los clientes. Esto, sin embargo, como mencionamos anteriormente, introduce la posibilidad de tener mensajes duplicados en el sistema distribuido dada la volatilidad de cada proceso y su nuevo procedimiento de checkpoints periódicos. Teniendo todo esto en cuenta, se realizó la incorporación de mecanismos de identificación de duplicados por medio de `sequence numbers` únicos por controlador.
  Cada controlador tiene presente el ***state*** de cada flujo de clientes y subsecuentemente de cada `sequence number` por nombre de controlador para dividir de forma efectiva las distintas combinaciones de mensajes entrantes que puede tener cualquier controlador. En particular se hace uso de esto en conjunto con la configuración mencionada previamente (ver nota en sección *Comunicaciones* **[*]**) para obtener como resultado una secuencialidad asegurada de mensajes dentro de un flujo específico.
  Esta diferenciación jerárquica por cliente y por controlador es necesaria dado que flujos individuales requieren un manejo fino para evitar romper con esta secuencialidad en las secciones de la topología del sistema en las que un controlador reciba mensajes de más de un predecesor.
  En la implementación, dicho ***state*** es representado de la siguiente manera:

```
state: dict[ClientID_t, dict[BufferName_t, BufferContent_t]]
# Donde:
ClientID_t: TypeAlias = int
BufferName_t: TypeAlias = str
BufferContent_t: TypeAlias = dict[ControllerName_t, ControllerSeqNum_t] | Any
ControllerName_t: TypeAlias = str
ControllerSeqNum_t: TypeAlias = int
```

En el siguiente diagrama se muestra adicionalmente el flujo del algoritmo general de consumo de mensajes que se sigue para cada controlador:



Criterio de checkpointing y casos especiales

Siguiendo por la misma línea de pensamiento de los casos mencionados en la especificación de manejo de mensajes duplicados, se definieron los siguientes métodos de checkpointing de manera específica a cada proceso según sus necesidades de almacenamiento, ya que las mismas varían:

- **Controladores en general**: Por medio de la abstracción de `MonitorableProcess` se define un callback genérico de almacenamiento de estado como wrapper al callback de procesamiento interno definido por cada nodo. De esta forma los controladores son fácilmente adaptables a la lógica común de revisión de duplicados, manejo de mensajes `ABORT`, y actualización y guardado de `sequence numbers` locales y de predecesores; y adicionalmente se le delega a este wrapper la realización de ACKs finales hacia el broker, garantizando que se efectúe después de haber culminado con todo el procesado de un mensaje.
  La persistencia de estado es realizada una vez por cada mensaje que se procesa. Esto se decidió para mantener al mínimo posible la cantidad de mensajes duplicados que se generan en el caso en que un controlador reciba un kill y no logre persistir su estado actual a disco; ya que persistiendo una vez por mensaje a lo sumo se va a duplicar un mensaje

por output de ese controlador (dado el efecto combinado entre la secuencialidad de los mensajes y la atomicidad de escritura de archivos de checkpoint: un archivo de checkpoint solo sobrescribe al anterior si fue escrito por completo en un archivo temporal aparte).
Más adelante se mostrará un diagrama de secuencia que en conjunto con el diagrama de actividades representan este algoritmo.

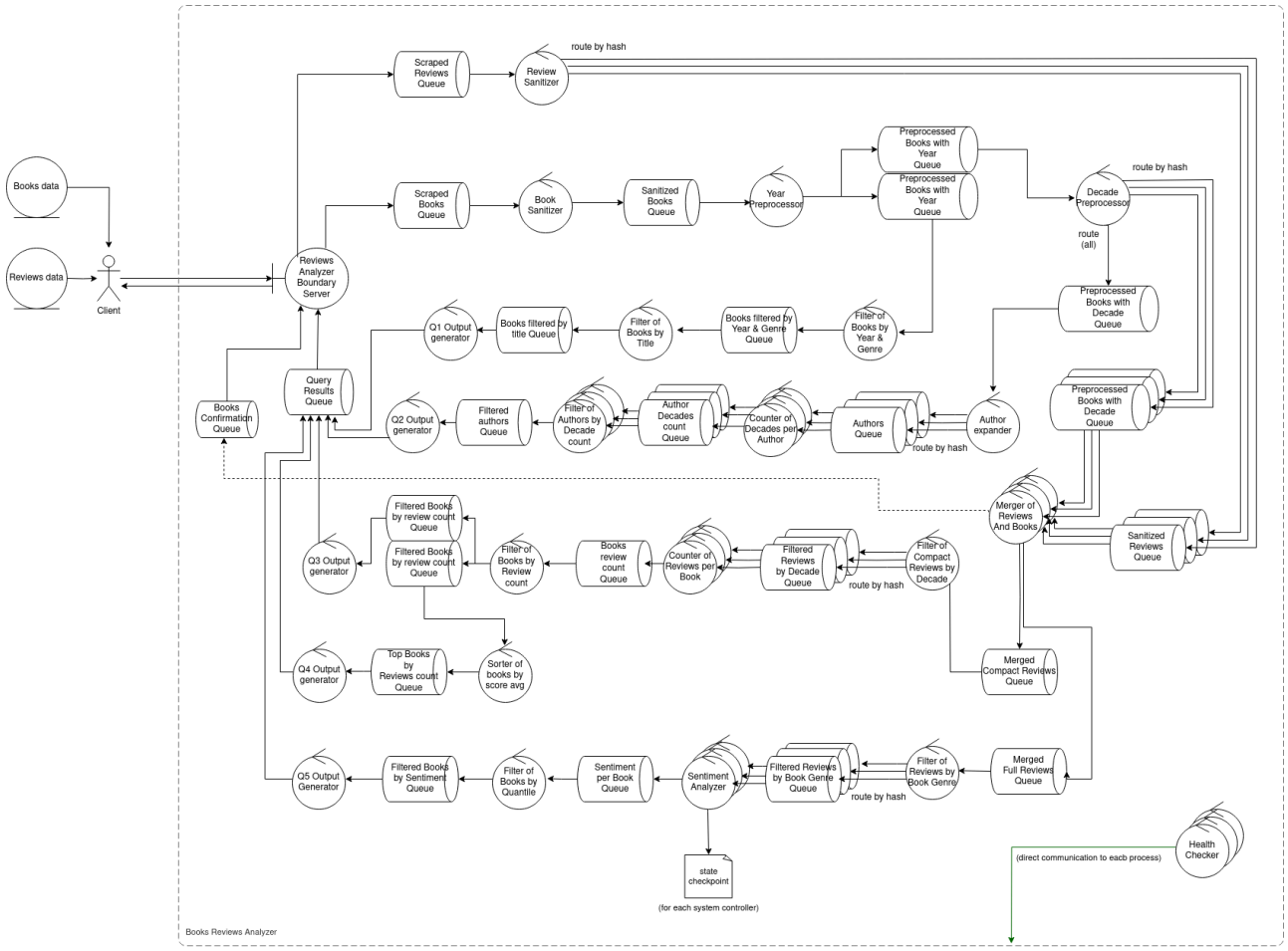
- **Mergers**: Estos controladores también utilizan la abstracción general mencionada anteriormente, sin embargo tienen como peculiaridad que poseen un estado aparte exclusivo a almacenamiento de datos de libros (también distinguidos por cliente). Este estado también es manipulado y persistido por cada batch de mensaje procesado, sin embargo a partir de la primera recepción de mensaje que indique que un cliente determinado finalizó con el envío de datos de libros, este estado adicional deja de persistirse para optimizar en gran medida los tiempos de procesado de las reviews. Otros detalles adicionales de los Mergers se mencionarán en el diagrama de robustez.

Vistas

Nota: Todos los diagramas se pueden observar con mayor claridad en el directorio **misc/** del repositorio.

Vista Física

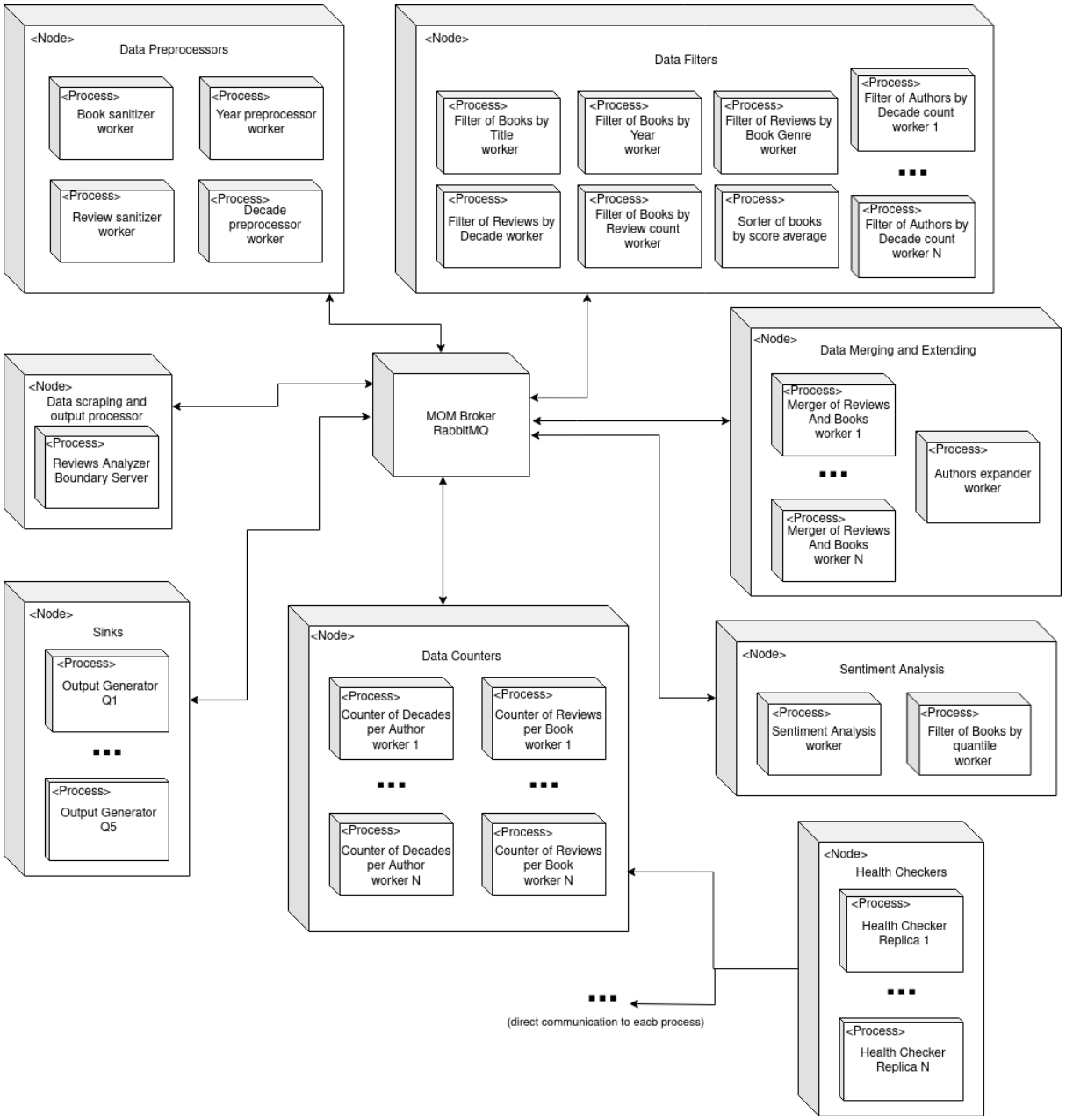
- Diagrama de Robustez



Se pueden observar los procesos y los flujos principales que componen el sistema.
Comparado a la anterior iteración del trabajo, se añaden los procesos de health check y las distintas persistencias realizadas por cada controlador. También se añadió la replicación de Sentiment Analyzers.
Además de esto, se observa que los Mergers ahora se encargan de comunicarse con el servidor de forma inmediata tras recibir el mensaje de finalización de ingesta de libros. Esta comunicación se da para que el servidor pueda sincronizar al cliente específico y que solo cuando los mergers hayan recibido toda la información de los libros se pueda empezar a enviar datos de reviews.

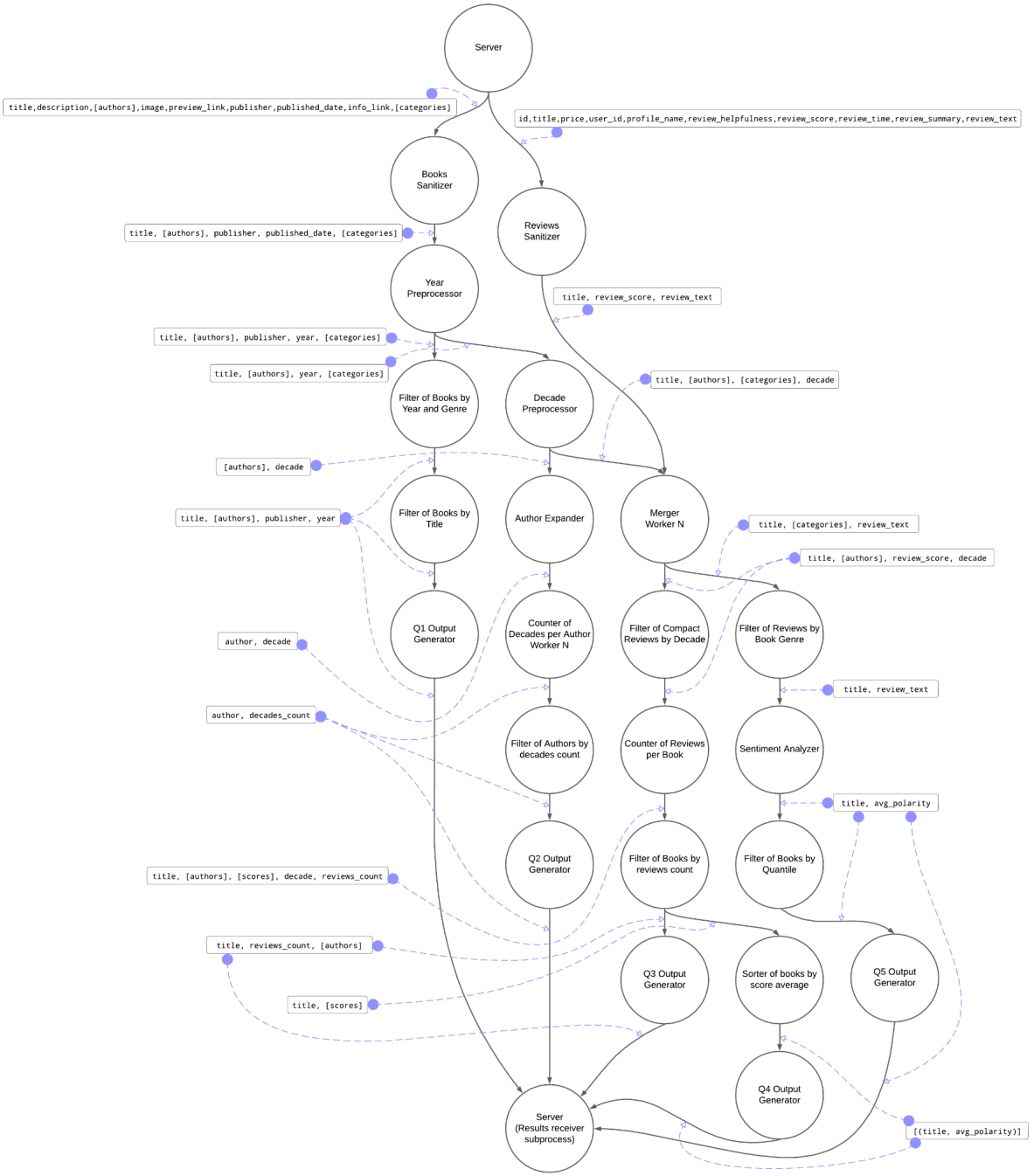
Este cambio con respecto a la iteración anterior se dio dado que, dada la naturaleza altamente concurrente del sistema, si se enviaban datos de reviews inmediatamente tras acabar con los libros, era muy probable que los mergers comenzaran a recibir dichas reviews antes de tener todos los libros disponibles. Esto ocasionaba que se tuvieran que almacenar las reviews que no fueran posibles de mergear mientras que se recibían todos los libros, generando un overhead importante de datos a bufferear, que para la instancia anterior del trabajo no presentaba inconvenientes, pero en contraste ahora se trata con un sistema con mucha mayor volatilidad que requiere guardados de estado constantes. Guardar el estado con un overhead tan grande podía llegar a ocasionar demoras de gran magnitud para todas las consultas relacionadas.

- Diagrama de Despliegue



Relacionado al diagrama de robustez, se observan los procesos a los que se les añadió replicabilidad en workers con respecto a la iteración anterior.
A su vez se definió un nuevo nodo para los health checkers, que se comunican directamente con los controladores como se mencionó anteriormente.

- DAG



Se puede visualizar el grafo acíclico dirigido del sistema, representando los flujos de datos que se dan a lo largo del mismo. Particularmente se destaca el comportamiento de:

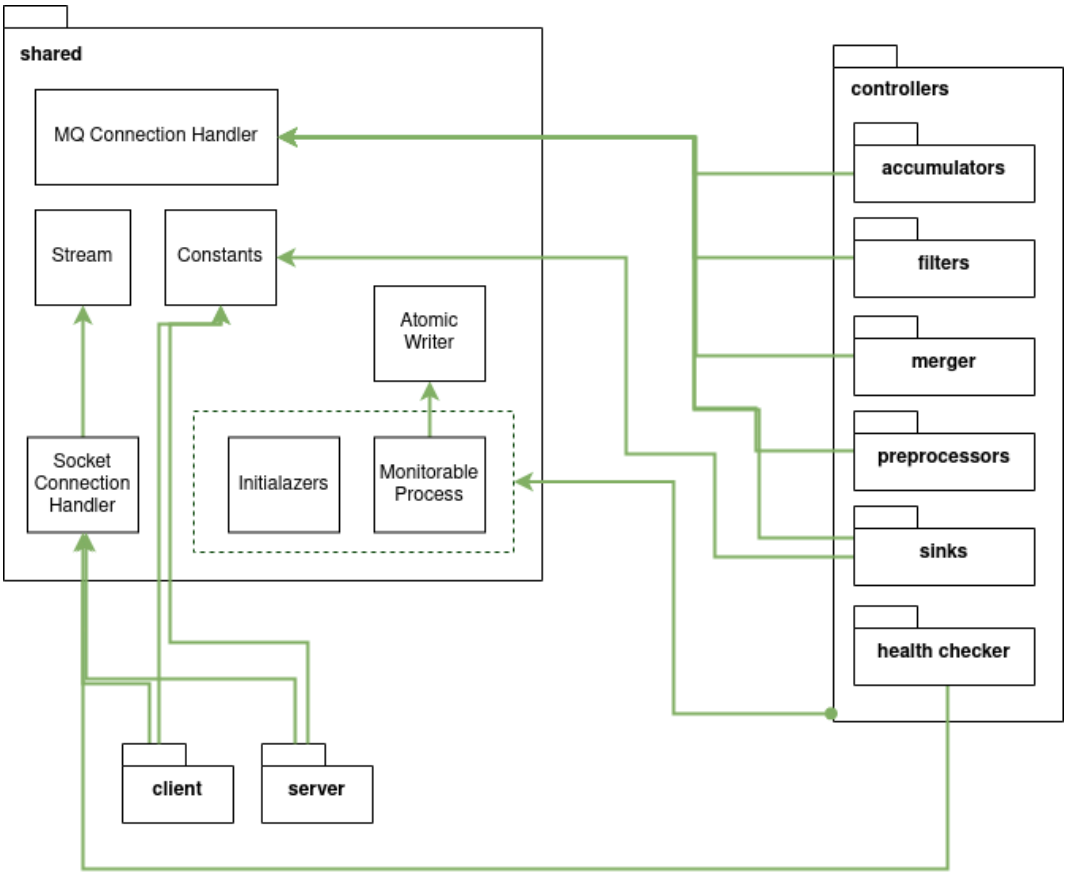
- Worker N: Son procesos que son independientes entre sí y por lo tanto cada réplica de worker realiza el mismo flujo del diagrama.
- Server: Es representado como dos nodos distintos dado que efectivamente el mismo tiene el proceso principal de donde consume los datos del

cliente, y un proceso hijo que se encarga de recibir concurrentemente los resultados y enviarlos al cliente.

- Merger: Reciben datos de dos fuentes distintas. Tener en cuenta las consideraciones explicadas anteriormente sobre la sincronización con el servidor.

Vista de Desarrollo

- Diagrama de paquetes

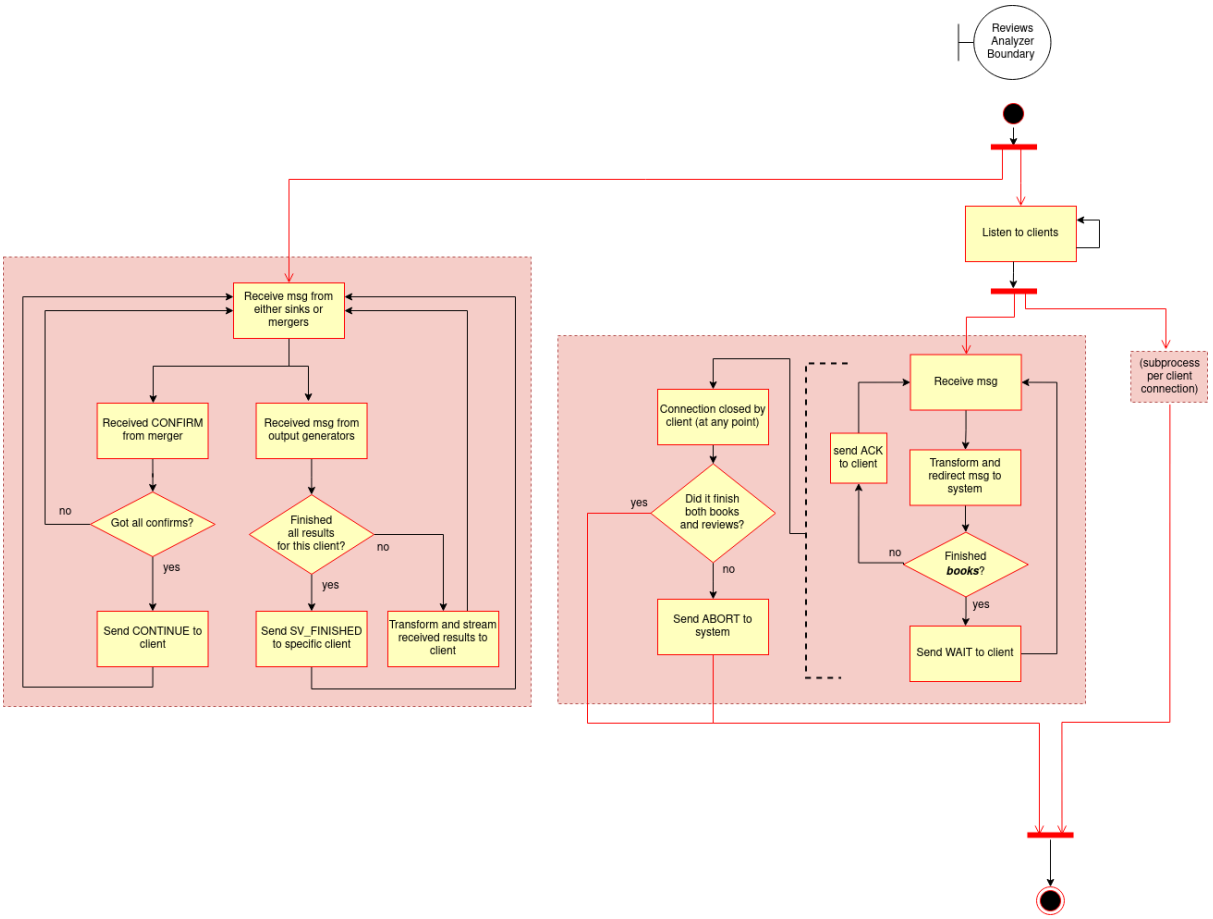


Podemos observar en el diagrama de paquetes que tanto cliente como servidor se comunican a través del `SocketConnectionHandler`, mientras que todos los demás procesos del sistema utilizan el `MQConnectionHandler` para la comunicación a través de las colas de RabbitMQ, siendo esta la capa de middleware común a cada uno de los componentes del sistema distribuido. Además se comparte la inicialización de configuraciones y logs para todos los procesos.

> También se puede ver como los controladores utilizan (heredan) de Monitorable Process para brindar todas las adaptaciones mencionadas anteriormente.

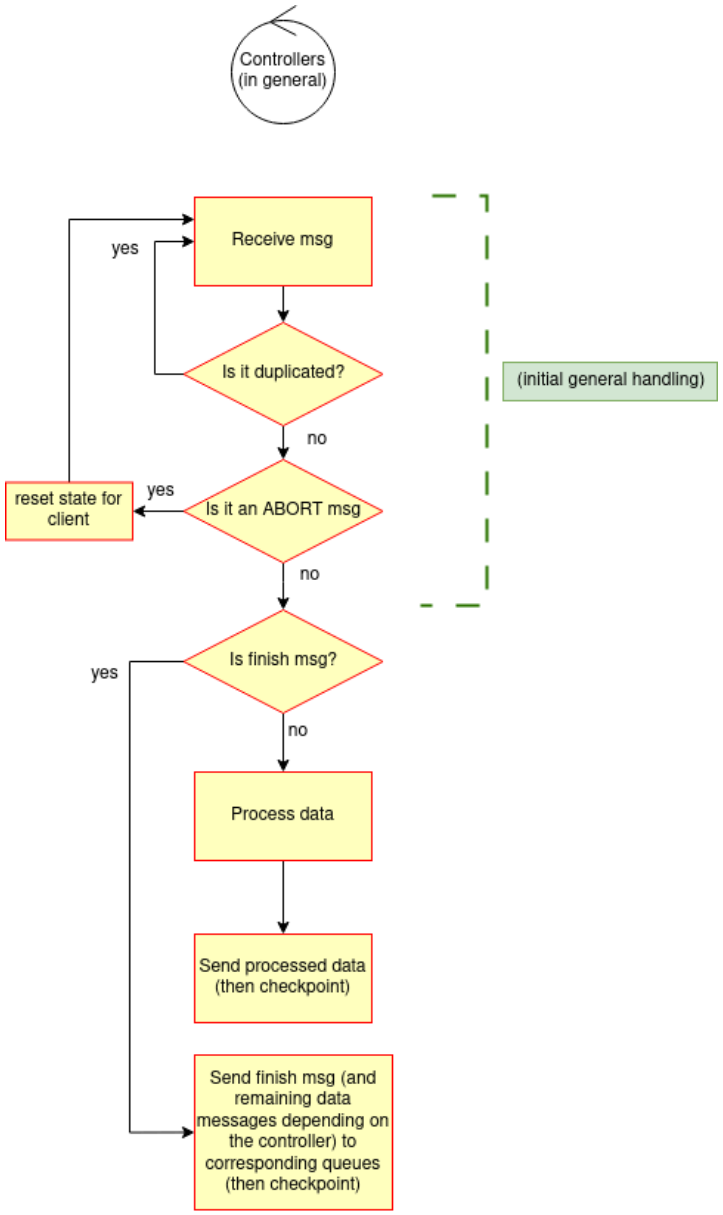
Vista de Procesos

- Diagrama de actividades: Boundary Server



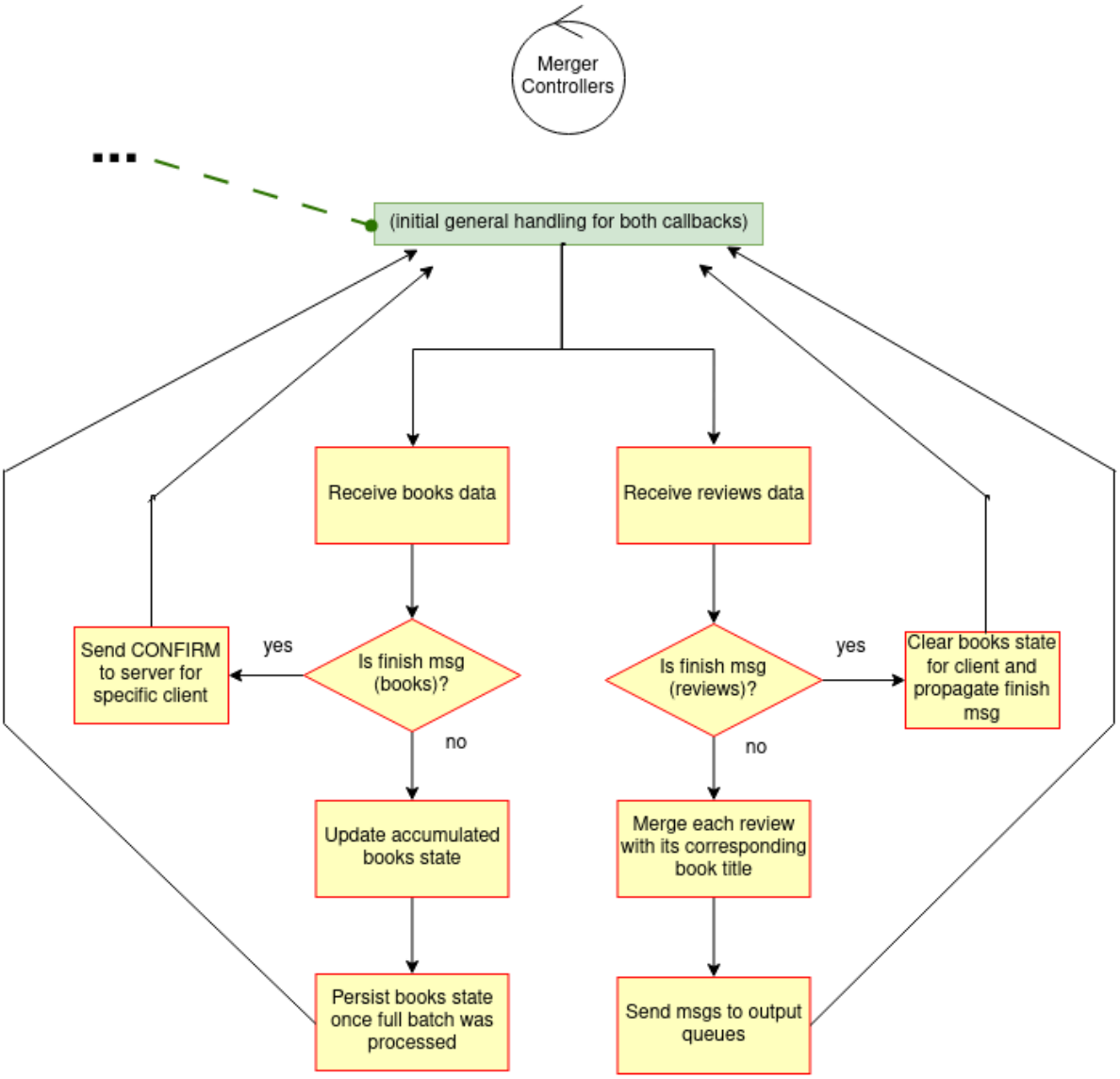
Se muestra el manejo general que realiza el servidor para aceptar el procesamiento de múltiples clientes y procesar de forma concurrente los resultados que llegan del sistema a modo de streaming, de tal forma que el cliente pueda obtener sus resultados en el menor tiempo posible. Cabe destacar que dicho streaming es realizado a su vez por medio de la abstracción `SocketConnectionHandler` entre servidor y cliente.

- Diagrama de actividades: Manejo general de mensajes en controladores



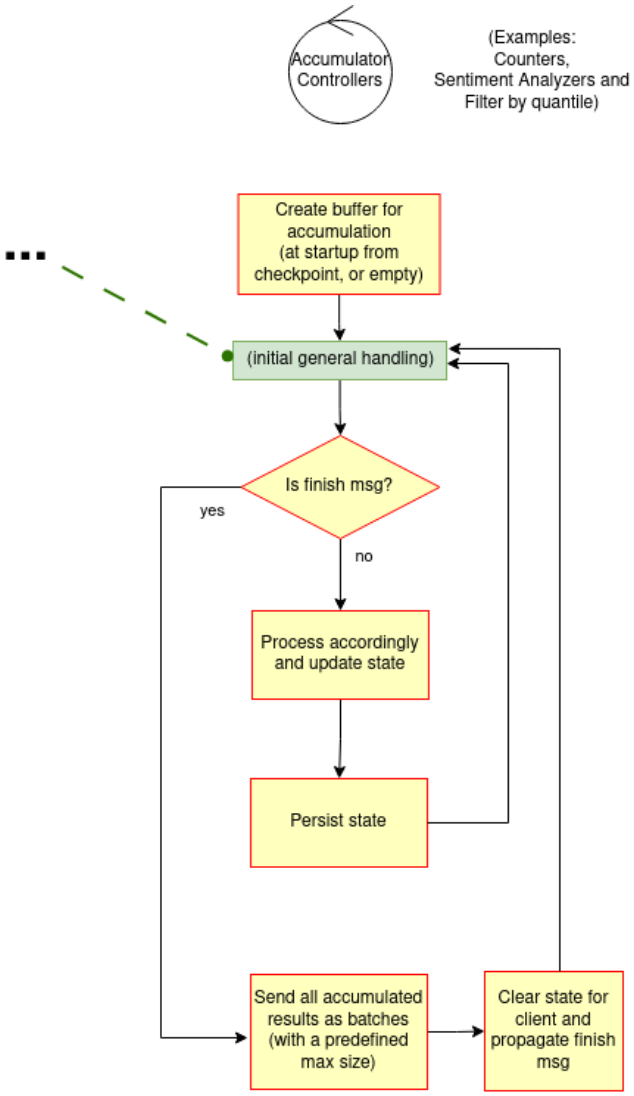
Relacionado al diagrama observado previamente (apartado *Mensajes duplicados* en la sección *Comunicaciones*), se muestra esta vista generalizada del manejo que hacen todos los controladores que derivan de `MonitorableProcess`, ya que su manejo inicial va a ser referenciado nuevamente en los diagramas siguientes.

- Diagrama de actividades: Mergers



Se puede visualizar el manejo de mensajes que se ejecuta según callback (recepción de libros/recepción de reviews) para los procesos Merger. Particularmente se observa lo mencionado anteriormente acerca del guardado de estado de libros exclusivamente para ese flujo correspondiente.

- Diagrama de actividades: Accumulator Controllers



Estos controladores se comportan de la misma manera en cuanto a su procesado de mensajes. Se destaca que en su etapa final de envío de resultados se tiene en cuenta lo mencionado en el apartado *Reducción de particionamiento de datos innecesarios* en la sección de *Adaptaciones realizadas* sobre la preferencia de utilización de batches por sobre registros individuales por payload.