Trabajo Práctico Final
Reddit Memes Analyzer (HA)

Sistemas Distribuidos I (75.74)

1^{er} Cuatrimestre 2022
Facultad de Ingeniería
Universidad de Buenos Aires

Integrantes
AGUERRE, Nicolás Federico (102145)
KLEIN, Santiago (102192)
PARAFATI, Mauro (102749)

Repositorio en GitHub

Índice

1. Introducción

En el presente informe, se explicarán los detalles de la arquitectura e implementación del sistema distribuido desarrollado en el marco del Trabajo Práctico Final de la asignatura *Sistemas Distribuidos I (75.74)* de la Facultad de Ingeniería, Universidad de Buenos Aires.

1.1. Sistema a desarrollar

La función del sistema es procesar publicaciones y comentarios de Reddit, obteniendo métricas de los mismos, con el agregado de que debe ser tolerante a caídas de los servicios (a excpeción del MOM) para proveer de una alta disponibilidad.

1.2. Naturaleza del problema

Dado el alto volumen de datos a procesar, así como la independencia de los mismos, resulta óptimo pensar en procesar cada post y comentario *en linea*, diagramando un **pipeline** que permita maximizar a su vez el paralelismo.

2. Arquitectura

2.1. Vista Lógica

A continuación, se observa un *DAG* que ilustra el flujo de información de posts y comentarios a través del sistema, y su procesamiento correspondiente:

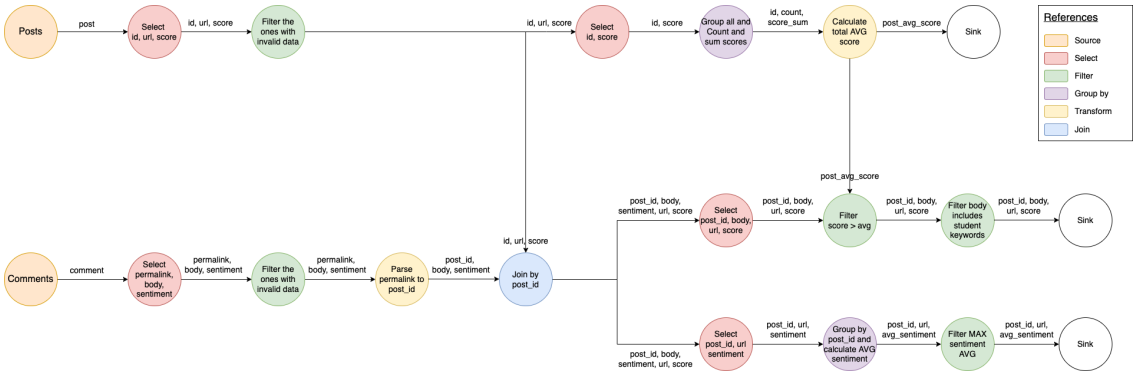

Figura 1: *DAG del sistema, donde se puede observar el flujo de la información y su procesamiento lógico*

Una vez ingresan al sistema los posts y comentarios, se extrae de los mismos los campos relevantes y se filtran aquellos que tengan información duplicada, efectuando transformaciones de ser necesarias. Luego, en el caso de los posts, se envíarán tanto al nodo que los agrupa y computa el score promedio de los mismos, como al joiner que les agregará a los comentarios la información de los mismos.

Los comentarios, por su parte, se van filtrando y transformando según el resultado que se quiera obtener: para el caso de obtener el post de máximo sentiment promedio, antes se efectúa una agrupación de comentarios con la información del post y luego se computa el sentiment promedio por post, para quedarse finalmente con el mayor valor. Por otra parte, para calcular aquellos posts que en su body incluyen palabras clave referidas a estudiantes, una vez hecho el join con los posts, se extrae la información necesaria, se efectúa el fitro, obteniendo aquellos que tienen score mayor al promedio, para luego filtrar el body de los mismos según las palaras clave.

Se decidió realizar el join de los comentarios con los posts tan temprano como sea posible en el sistema para poder descartar lo antes posible aquellos comentarios pertenecientes a posts inválidos (debido a que el flujo de comentarios es potencialmente muy alto).

2.2. Vista Física

A continuación, se detallará la vista física arquitectónica del sistema, presentando en primer lugar el diagrama de robustez completo, y luego separándolo en bloques para la facilidad del análisis:

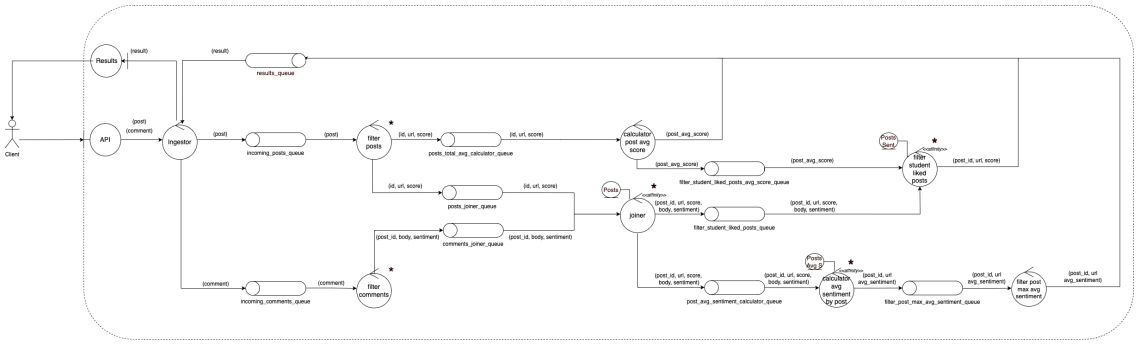

Figura 2: *Diagrama de robustez completo del sistema*

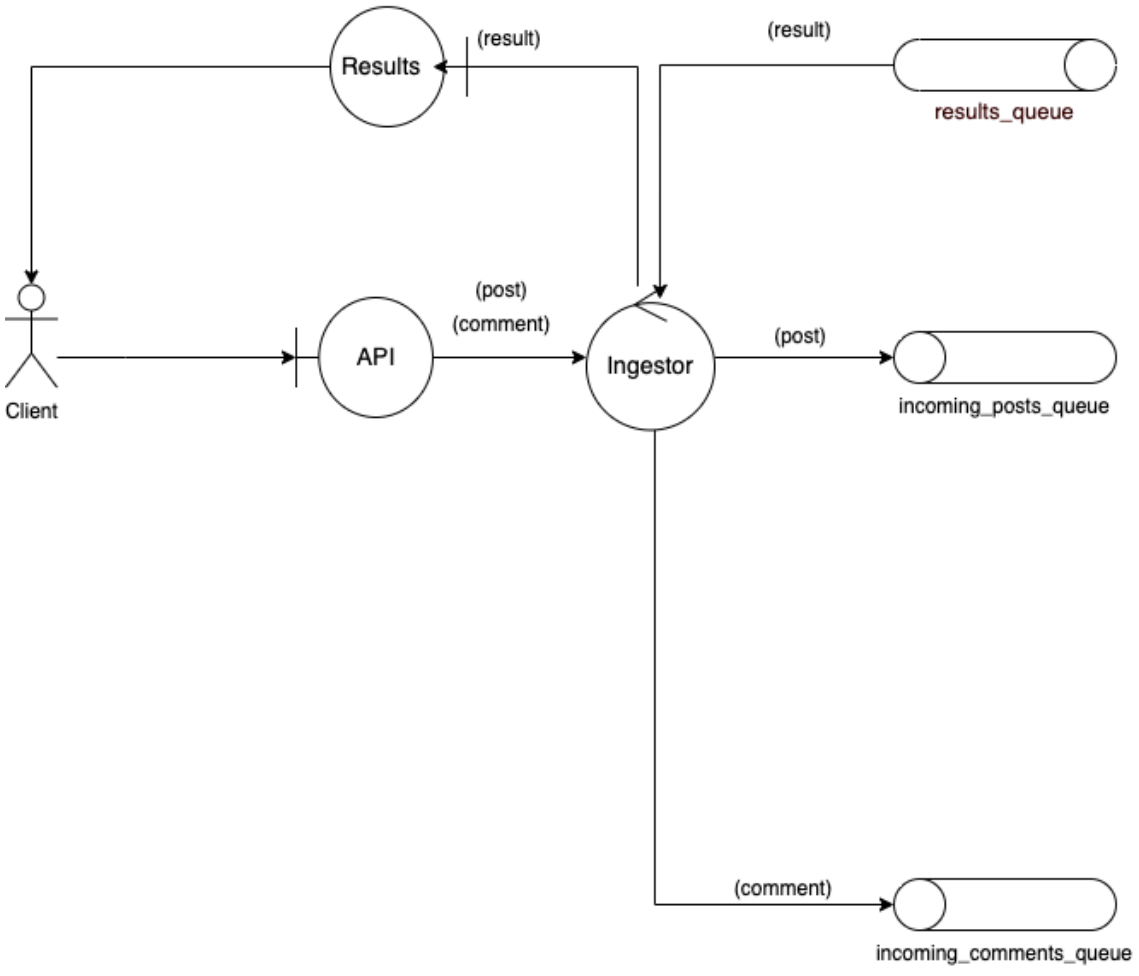

Figura 3: *Diagrama de robustez (Parte I)*

En principio, tenemos las boundaries del sistema: una API de ingreso de datos por parte del *Client* hacia el *Ingestor* y una salida de *Resultados* desde el *Ingestor* hacia el client. De esta manera, el *Ingestor* es el primer nodo y constituye la frontera del sistema con sus usuarios. El

mismo publica los posts y los comentarios en sus colas, y recibe los resultados de la cola de resultados.

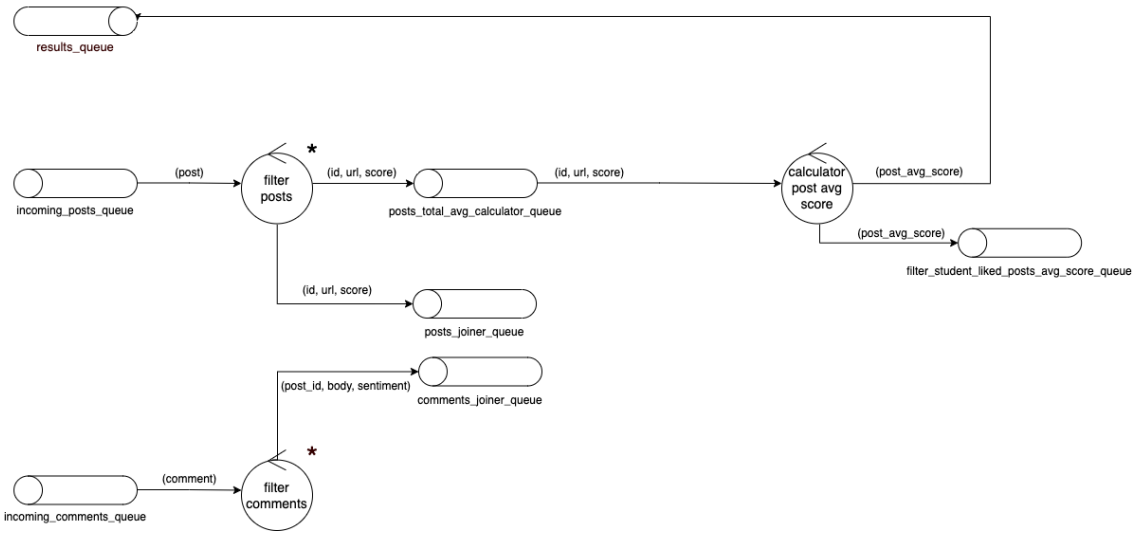

Figura 4: *Diagrama de robustez (Parte II)*

Luego, los posts y comentarios ingestados por el Ingestor en las colas son consmidos por los filtros *filter posts* y *filter comments*, respectivamente, quienes descartan aquellos con información inválida y filtran los campos innecesarios. Ambos filtros se encuentran replicados. El filtro de posts, por un lado publica los resultados en una cola que consume la entidad *calculator post avg score*, quien calculará el *score* promedio de los posts y publicará el resultado tanto en la cola de resultados como en otra cola que consumirá otra entidad (*filter student liked posts*) más adelante. Además, ambos filtros publican sus outputs en las colas que consumirá el *joiner*.

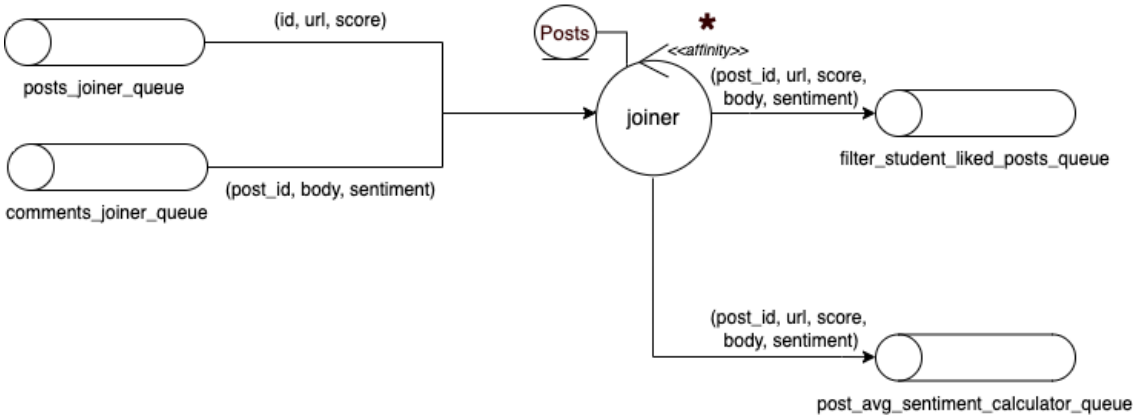

Figura 5: *Diagrama de robustez (Parte III)*

El *joiner*, por su parte, alimentado con las colas de posts y comments en las cuales publican los filtros, almacena en memoria y disco los *Posts*, a partir de los cuales agregará la información a los comentarios que recibe. Cabe destacar que el mismo se encuentra replicado **con affinity**, por

lo que cada réplica recibirá posts y comentarios en un determinado rango de ids. Luego, publicará los comentarios con la información de los posts a los que corresponden en dos colas: una que consumirá *filter student liked posts* y otra que consumirá *calculator avg sentiment by post*

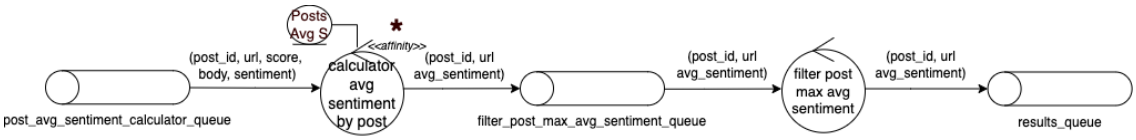

Figura 6: *Diagrama de robustez (Parte IV)*

Ahora se observa a la entidad *calculator avg sentiment by post*, también replicada **con affinity** (según post id), la cual recibe los comentarios con información de posts que publica el *joiner* y calcula el *sentiment* promedio de los mismos agrupando por post. Por ello, almacena los datos parciales de *sentiment* promedio por cada post en memoria y disco. Luego, publica los resultados en una cola que consume la entidad *filter post max avg sentiment*, que se queda con el post con sentiment de mayor valor y lo publica en la cola de resultados.

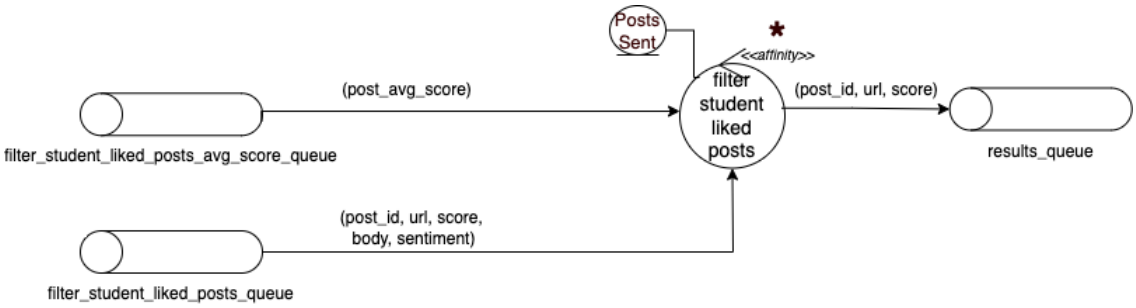

Figura 7: *Diagrama de robustez (Parte V)*

Por último, se analiza la entidad *filter student liked posts*, también replicada *con affinity* según posts, almacena en memoria y disco los posts ya publicados en la cola de resultados (para evitar repetidos), y el score promedio de todos los posts que recibió de parte del *calculator post avg score*, retroalimentándose con la cola de comentarios con información de posts a la que publica el *joiner*.

Si bien se omiten en los diagramas, cabe destacar que todas las entidades del pipeline también almacenan estado propio del middleware en memoria y disco. Este estado consiste, principalmente, de la cantidad de mensajes de terminación recibidos y enviados, y en algunos casos se agrega información adicional.

En los diagramas de robustez anteriores no se exhiben los detalles del monitoreo en pos de evitar sobrecargarlo, mas en el siguiente diagrama de despliegue podemos observar cómo se encuentran desplegados los monitores (líder y réplicas) y las entidades del pipeline (incluyendo sus réplicas), qué procesos corren dentro de esos containers y cómo interactúan.

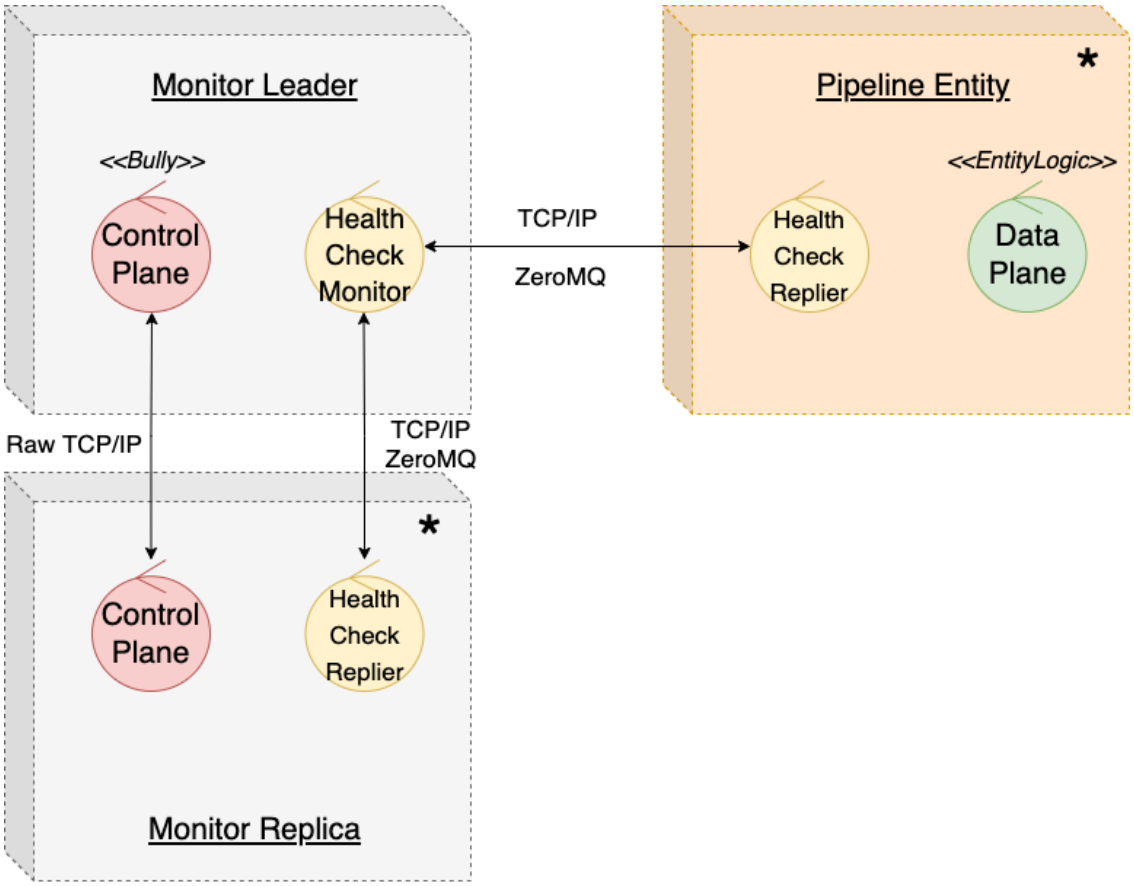

Figura 8: *Diagrama de despliegue del sistema (Parte I)*

Por un lado, observamos que hay un *Monitor Leader*, que es el único que corre el *Health Check Monitor* de los monitores. Este proceso se encarga de efectuar el health check de todos los servicios, tanto entidades del pipeline como las réplicas de monitores.

El *Control Plane* de los monitores es el proceso que ejecuta el algoritmo de elección de líder *Bully* entre ellos. La comunicación entre los *Control Planes* de los monitores se realiza a través de sockets TCP/IP.

Tanto las réplicas de los monitores como las entidades del pipeline contienen un proceso *Health Check Replier* que responde a los health checks que emite el proceso *Health Check Monitor* del monitor líder. La comunicación entre ellos se realiza a través de sockets de la librería ZMQ que corren por encima de TCP/IP.

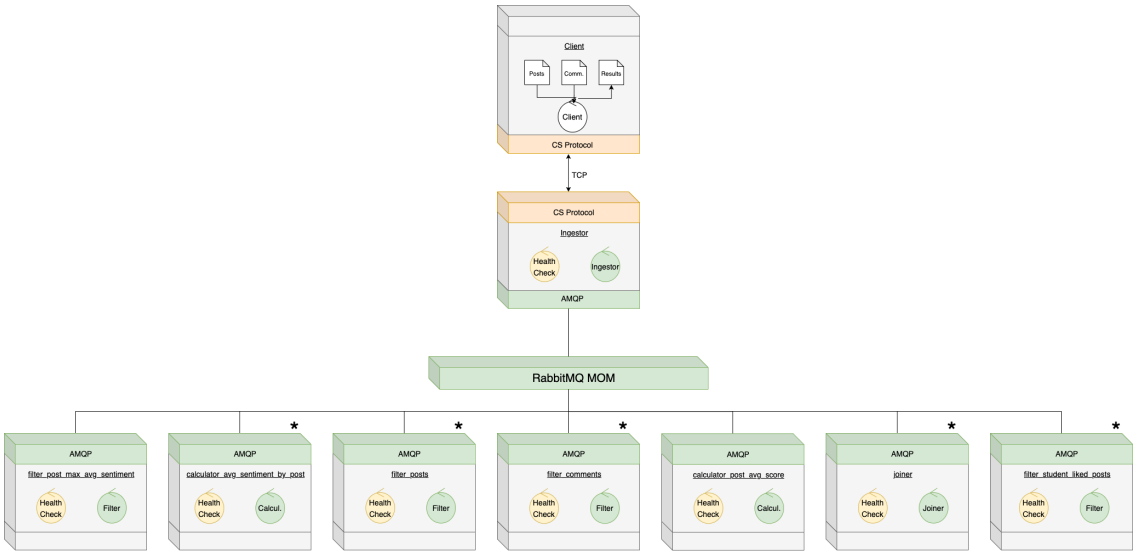

Figura 9: *Diagrama de despliegue del sistema (Parte II)*

Asimismo, se observa el diagrama de robustez del resto del sistema, en el que se muestra en la parte superior el despliegue y la comunicación entre el *cliente* activo y el *ingestor* a través de un protocolo binario TCP/IP. Luego, se detallan los containers de las entidades del pipeline, sus réplicas (de corresponder), y los procesos que corren vistos anteriormente. Se observa en el centro el MOM RabbitMQ, a través del cual se comunican las entidades del pipeline y sus procesos de lógica de negocio.

2.3. Vista de Desarrollo

Se ilustra a continuación la organización del código del sistema en un diagrama de paquetes.

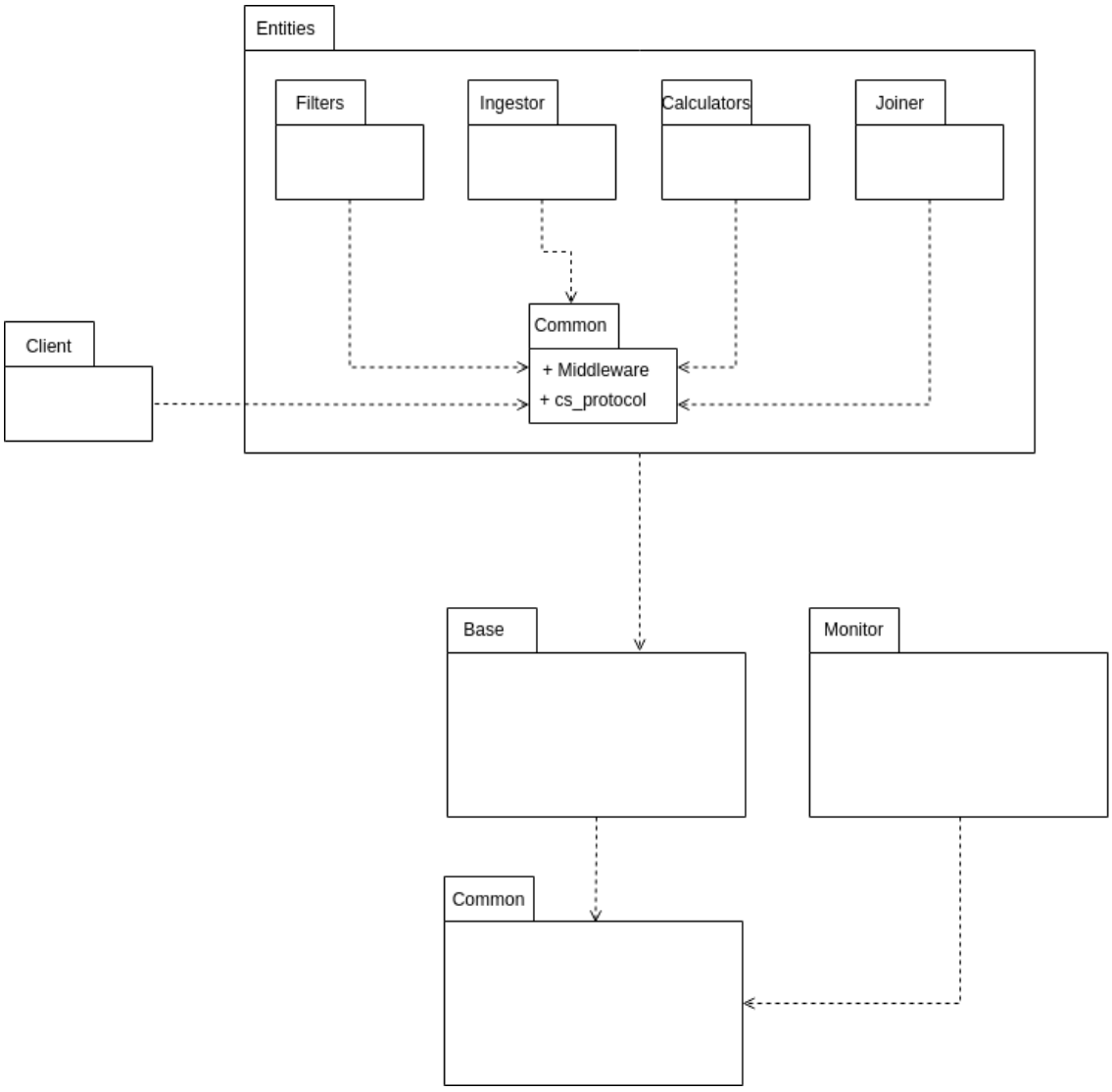

Figura 10: *Diagrama de paquetes del sistema.*

2.4. Vista de Procesos

Con el objetivo de ilustrar el funcionamiento del sistema, a continuación se muestra un diagrama de secuencia que detalla la secuencia de llamados entre las distintas entidades que se realizan para obtener los resultados esperados del procesamiento de posts y comentarios. Cabe destacar que este diagrama corresponde a la parte de procesamiento lógico del data plane, y omite los detalles de monitoreo y respaldo de datos (y su recuperación) de los nodos.

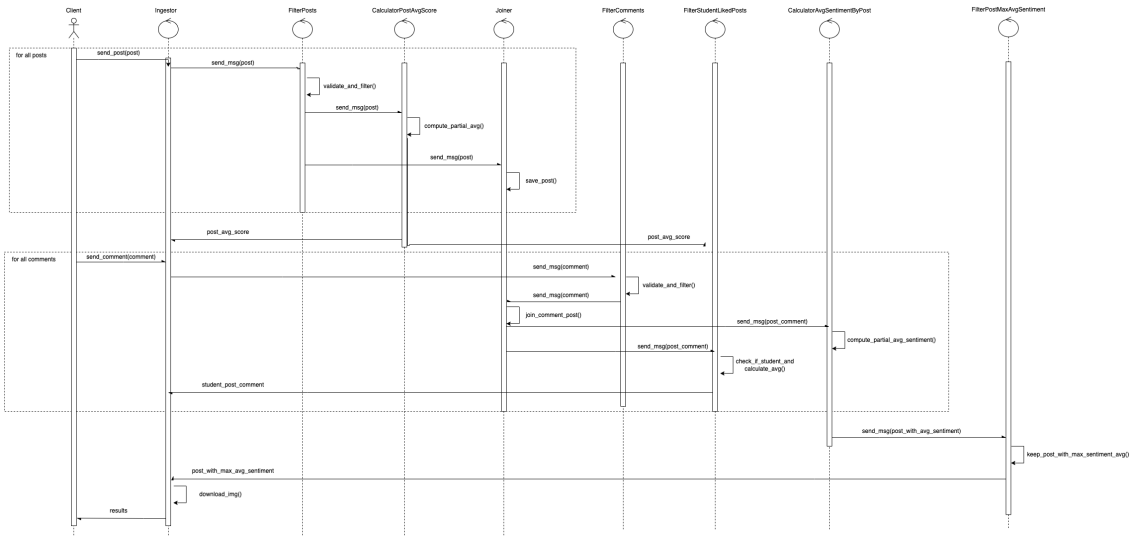

Figura 11: *Diagrama de secuencia del procesamiento de posts y comments y la obtención de los resultados.*

En el mismo, se observa que el *cliente* envía, secuencialmente, primero los posts en su totalidad y luego los comentarios en su totalidad. Tras la ingestión de posts y comentarios, se desencadenan los llamados entre entidades para realizar el procesamiento asincrónco, obteniéndose los resultados asincrónicamente.

Cabe aclarar que el diagrama corresponde a la corrida de un solo cliente, pero con el agregado de múltiples clientes secuenciales, en rigor, las distintas entidades que conforman el pipeline no finalizan su ejecución, sino que se ejecutan permanentemente.

Por otra parte, se incluye el siguiente diagrama de secuencia, en el cual se ilustra el proceso mediante el cual el estado de un nodo genérico del pipeline es persistido a disco ante la llegada de mensajes a través del broker.

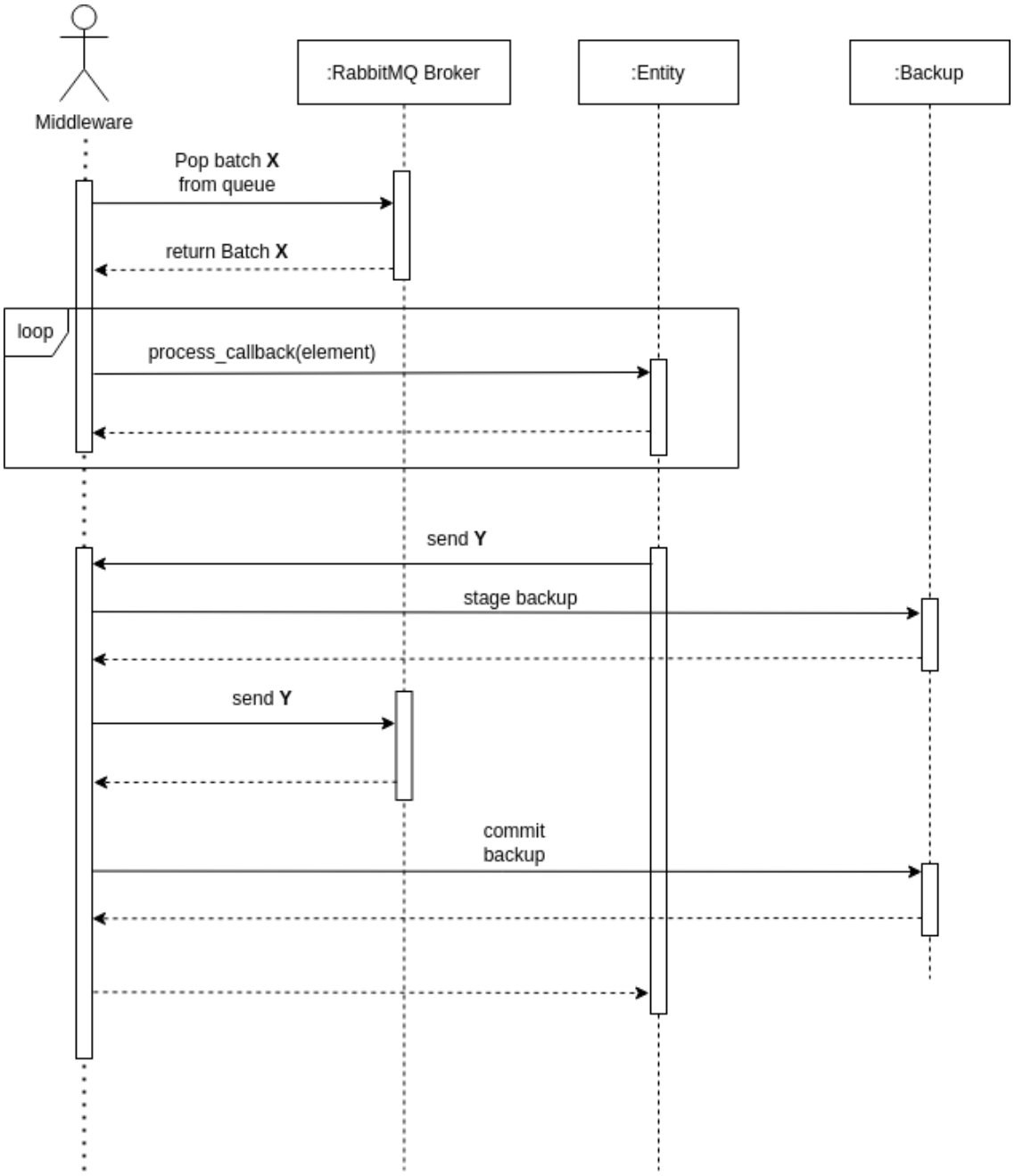

Figura 12: *Diagrama de secuencia de la creación de backup de estado en disco.*

Ver sección *Resguardo de estado en los nodos del pipeline* para mas detalles.

Por último, se exhibe mediante un diagrama de actividad el ciclo de monitoreo de los nodos por parte del *Monitor líder*:

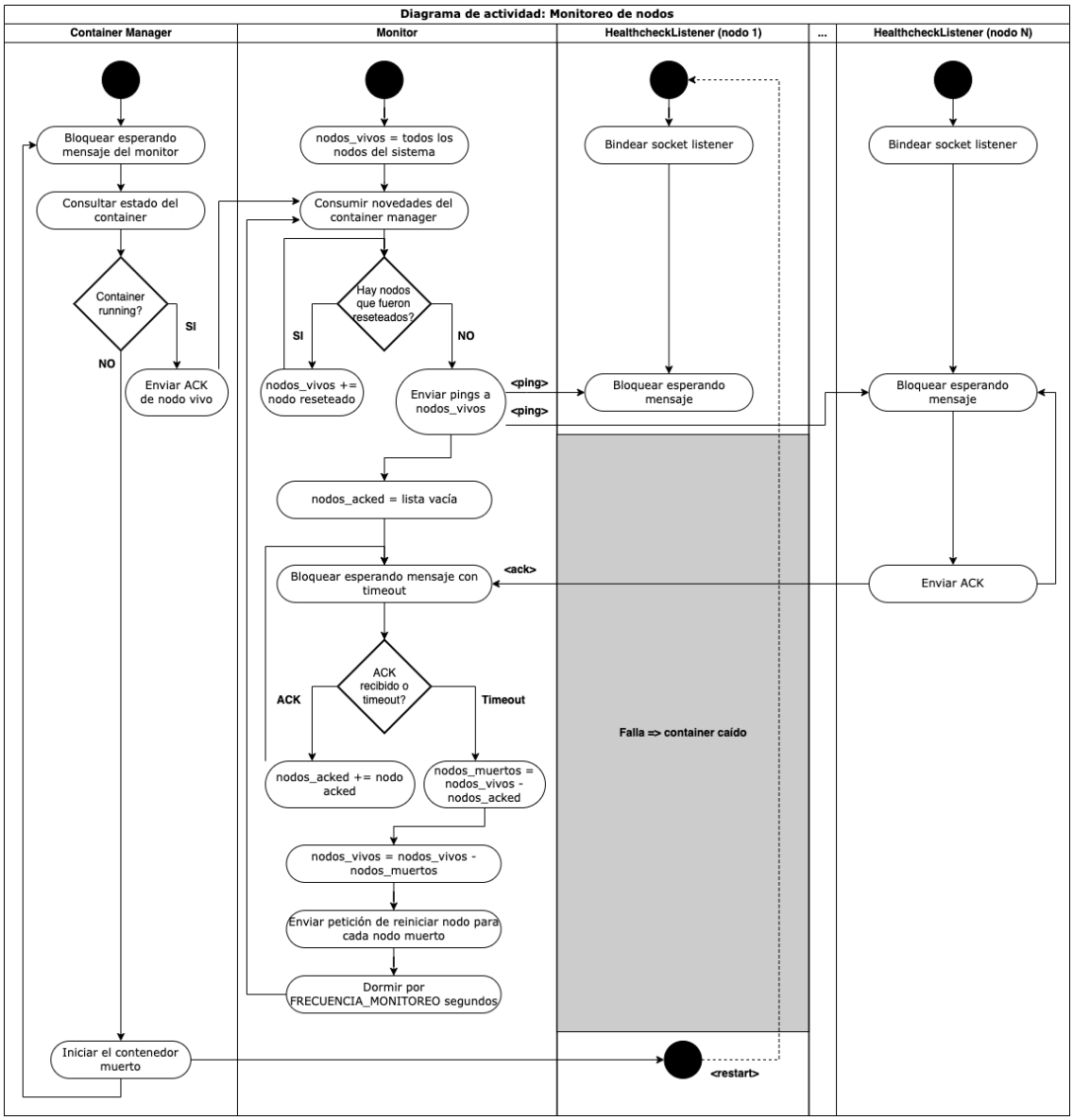

Figura 13: *Diagrama de actividad de montoreo de nodos del sistema por parte del monitor líder.*

2.5. Escenarios

Se ilustran a continuación los casos de uso del sistema.

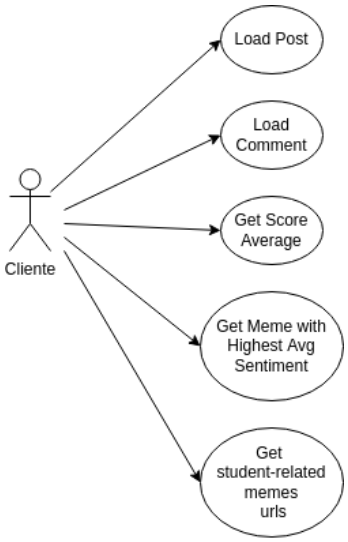

Figura 14: *Diagrama de escenarios del sistema.*

3. Implementación de tolerancia a fallos

El principal agregado del presente trabajo, comparándolo con el trabajo anterior, es la necesidad de construir un sistema altamente disponible, para lo cual es necesario implementar estrategias de tolerancia a fallos.

Como en general resulta muy difícil (o hasta imposible) asegurar un 100 % de tolerancia a fallos, se busca minimizar lo máximo posible la probabilidad de ocurrencia de los mismos, siempre considerando el caso de uso en cuestión.

3.1. Supuestos

Se planteó construir un sistema tolerante a fallos en los nodos, que pueden provocar la caída de cualquier nodo de procesamiento del pipeline o replica del monitor en cualquier momento y por un tiempo indefinido. Este es el caso de fallas más básico, al que podrían agregarse distintos escenarios que no fueron contemplados en el alcance del trabajo actual, como podrían ser fallas de particionamiento de red, fallas bizantinas, etc.

3.2. Clasificación de entidades

El sistema se compone por diversas entidades que se pueden dividir de la siguiente manera, según la función que cumplan:

- **Entidades que forman parte del pipeline de procesamiento de posts de Reddit**: el *ingestor*, los *filters*, las *calculators* y los *joiners*. Son **stateful**, ya que todas ellas contienen estado del *Middleware*, y algunas de ellas contienen además estado de lógica de procesamiento (por ejemplo, el promedio de score de posts parcial). Asimismo, hay ciertas que están replicadas sin affinity (la mayoría de los *filters*), otras que están replicadas con affinity (tales como el *joiner* o el *calculator avg sentiment by post*), mientras que otras están sin replicar (por ejemplo, el *ingestor*, o el *calculator post avg score*).
- **Monitores**: su implementación está dividida en *control plane* y *data plane*. Todas las réplicas ejecutan el control plane, donde se implementa la elección de líder *Bully*, mientras que solamente el líder ejecuta el *data plane*, donde se efectúa health check de todos los servicios y se reinician aquellos que se encuentran caídos, utilizando *Docker in Docker*.

3.3. Comunicación

El sistema se desarrolló utilizando el MOM *rabbitmq* para la comunicación de mensajes entre entidades del pipeline, *ZeroMQ* como transporte del mecanismo de health check entre el data plane del monitor y las entidades, y sockets TCP tanto para la comunicación entre el cliente y el ingestor, como para la comunicación entre el control plane de los monitores que ejecuta la elección de líder.

3.3.1. Topología

La topología del pipeline se encuentra definida en un archivo, y el middleware compartido por todas las entidades del pipeline ejecuta la creación de los *exchanges* y *queues* del mismo, las cuales son operaciones idempotentes.

3.4. Resguardo de estado en los nodos del pipeline

Para garantizar el correcto flujo de funcionamiento del sistema ante una caída y recuperación en uno de sus nodos, fue necesario implementar un sistema que permitiera a dicho nodo continuar su procesamiento a partir del punto donde estaba cuando ocurrió la caída.

Para ello, se decidió que cada nodo persistiera el estado necesario para su funcionamiento en un almacenamiento estable cada un cierto período. Ante una caída, el nodo simplemente recupera el estado persistido y continúa su ejecución desde donde había quedado.

3.4.1. Deduplicación de mensajes

Con el objetivo de mejorar la performance del sistema, el *backup* del estado a disco no se hace ante cada mensaje que el nodo recibe, sino que se hace con una cierta periodicidad (por ejemplo, cada 5 mensajes). Esto puede llevar a que el nodo envíe mensajes duplicados luego de recuperarse de una caída. Para tratar este problema, se utiliza un mecanismo de deduplicación basado en IDs incrementales. Esto permite a los nodos detectar aquellos mensajes que fueron duplicados.

Una dificultad surgida durante esta implementación es que los mensajes son en realidad *batches*, y para que la deduplicación basada en IDs incrementales funcione, es necesario garantizar que el *batch* con id X contenga **siempre los mismos mensajes**. Para lograr esto, fue necesario garantizar que los batches siempre se envíen cuando están llenos, excepto el último.

3.4.2. Secuencia de persistencia

Para garantizar la consistencia en el estado de persistencia y minimizar la probabilidad de terminar con un estado corrupto en almacenamiento persistente, la secuencia que se sigue a la hora de recibir un mensaje es la siguiente:

1. Extraer un batch de la cola de RabbitMQ
2. Procesar el mensaje, actualizando el estado interno.
3. Si ya se procesaron n batches, o si hay algún batch de salida lleno y listo para enviar, *stagear* el estado a persistir **mediante la creación de un archivo temporal.**
4. Enviar los batches llenos.
5. *Commitear* el backup, renombrando el archivo temporal.

Adicionalmente, también se persiste el estado siguiendo una lógica parecida cuando se reciben mensajes de **termination**, y también cuando se los envían.

3.5. Monitoreo de servicios caídos

Para la detección de fallas en los servicios de procesamiento del pipeline, se diseño implementar un **Monitor** encargado de dicha tarea, así como de reiniciar al nodo caído para que pueda continuar con su tarea.

Al tratarse de un sistema que debe ser tolerante a fallos, es importante que este monitor también lo sea, por lo que será necesaria la replicación del mismo. Sin embargo, en este caso y a diferencia de los nodos de procesamiento *stateless*, es necesario elegir un líder entre las replicas, y que el resto sean pasivas. Esto se debe a que las tareas de reiniciado de nodos y monitoreo

propiamente dicho deben ser realizadas sólo por un nodo, por un lado por consistencia de estado con Docker, y por otro, para evitar el flujo innecesario de mensajes en la red.

Como se adelantó anteriormente, para lograr estos objetivos, el monitor fue diseñado utilizando una división de alto nivel: **Control Plane** y **Data Plane**.

3.5.1. Control Plane

El Control Plane es el proceso que se encarga exclusivamente de la coordinación y el consenso entre las distintas réplicas del monitor. Este proceso será responsable por llevar a cabo las distintas elecciones de líder que sean necesarias, así como de detectar si el líder actual deja de responder, para proponer una nueva elección. Esto se hace de forma ortogonal al monitoreo de servicios de procesamiento, ya que esta tarea será trabajo del proceso de Data Plane.

Para la coordinación y el consenso entre réplicas de *Monitor*, se implementó el algoritmo de elección de líder Bully visto en la materia, sobre sockets TCP/IP. Así, el proceso de Control Plane de cada réplica tendrá un hilo acceptor, que recibirá conexiones entrantes, y un hilo encargado de atender las peticiones y respuestas que conlleva el algoritmo Bully.

Para la conexión entre las distintas réplicas, se optó por que todas las réplicas intenten conectarse con todas las otras, con una lógica de retries y de timeout (con varianza) de conexión de por medio. De esta manera, se pretendió mitigigar al mínimo el riesgo de que una réplica no puedo conectarse con otra réplica activa. Ante el establecimiento de una conexión, el nodo que se conecta informa su id al nodo que lo aceptó, respondiendo este último con un *DISCOVERY ACK*. El hilo acceptor y el hilo ejecutor del Bully comparten un diccionario que mapea el id del peer con el socket TCP, protegido mediante un lock.

Una vez establecidas las conexiones, el algoritmo se sustenta sobre la base de que si un nodo no responde o se interrumpe la conexión con él por algún motivo, entonces el mismo se cayó, por lo que ya no interviene en el proceso de elección de líder. Por supuesto, esto simplemente resiste a caídas de nodos y no a network partitions o delays, ya que se rompería la lógica de que un timeout equivale a un nodo caído.

El algoritmo se implementó a través de una máquina de estados, existiendo los siguientes estados y las transiciones:

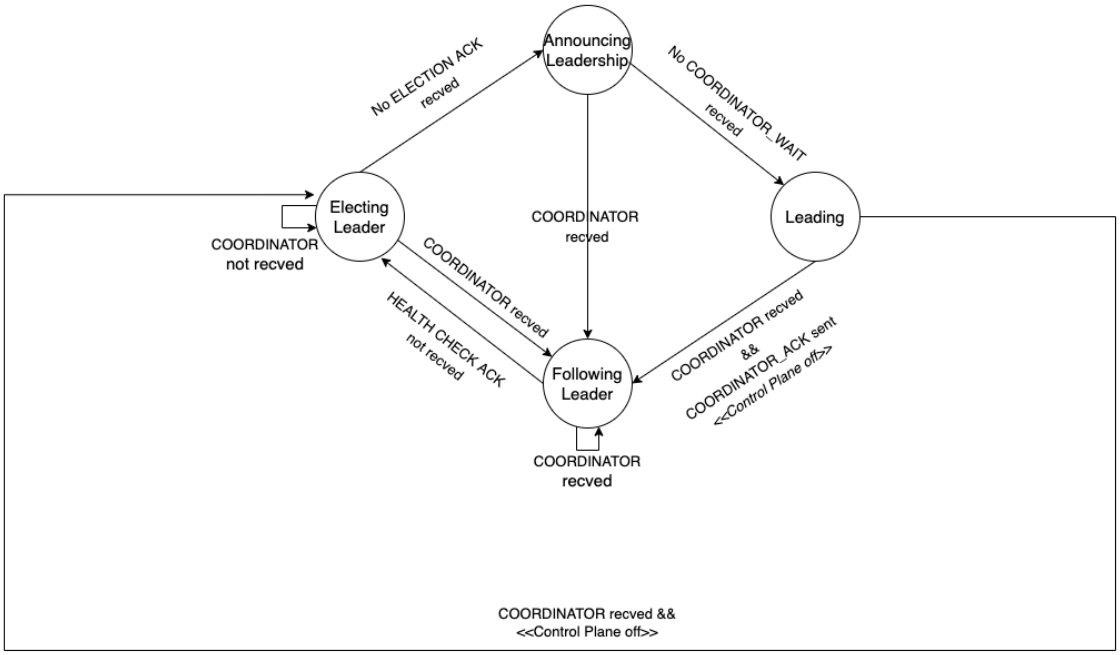

Figura 15: *Diagrama de estados de elección de líder Bully*

Una vez la réplica de monitor se levanta, ingresa en el estado **ElectingLeader**, por el cual se envía el mensaje *ELECTION* a las réplicas de mayor id. Si se obtiene la respuesta *ELECTION ACK*, se espera a que el coordinador nuevo se anuncie a través del mensaje *COORDINATOR*, con un timeout. De no anunciarse dentro del timeout, se repite el proceso. De no obtenerse ningún *ELECTION ACK*, el nodo se autoproclama líder, ingresando al estado **AnnouncingLeadership**, e informando a las réplicas que están conectadas su liderazgo a través del mensaje *COORDINATOR*.

En dicho estado **AnnouncingLeadership** pueden darse dos situaciones: en primer lugar, que no haya un coordinador de menor id activo anteriormente, caso en el cual se esperaría recibir los *COORDINATOR ACK* de todos los nodos de menor id, luego de lo cual pasaría al estado el líder al estado **Leading**, *spawneando* un proceso de tipo **Data Plane** para comenzar a ejercer como líder de monitoreo. En cambio, la segunda situación, que se sucede de existir un coordinador de menor id ejecutándose, el mismo responderá *COORDINATOR WAIT* al mensaje *COORDINATOR* en tanto y en cuanto no haya finalizado la ejecución del proceso *Data Plane*. Así, se evita que hayan dos coordinadores coexistiendo.

Una vez las réplicas que no ganaron el proceso reciben el mensaje *COORDINATOR*, pasan al estado **FollowingLeader**, en el cual envían cada cierto tiempo un *HEALTH CHECK* al coordinador, que debe responder dentro de un timeout un *HEALTH CHECK ACK* para no ser considerado caído (frente a lo cual las réplicas volverían al estado **ElectingLeader**, ejecutándose una nueva elección). Si en este estado se anuncia un nuevo coordinador, se comenzará a seguir al nuevo líder, enviándole *HEALTH CHECK* a este.

3.5.2. Data Plane

El Data Plane es el proceso que correrá en la replica líder de los monitores, y es el que efectivamente monitoreará a los servicios de procesamiento, para luego levantar a aquellos que fallen o que dejen de responder.

3.5.2.1 Monitor

Para la implementación de este proceso, se buscó seguir un diseño bastante sencillo que permita asegurar la fiabilidad del código:

1. Comenzar definiendo la lista de nodos vivos como todos los nodos.
2. Consumir mensajes del **ContainerManager** (de forma no bloqueante) para registrar nodos que hayan sido registrados como vivos.
3. Enviar ping a todos los nodos.
4. Esperar respuesta de los pings hasta un máximo de `TIMEOUT_SEGUNDOS`.
5. Luego de este tiempo (o antes en caso de que se hayan recibido todas las respuestas), chequear si algún nodo no respondió.
6. Considerar muerto a cada nodo que no respondió, y enviar un mensaje del tipo `RESETEAR_NODO` al **ContainerManager**.
7. Repetir el proceso desde **2**, sacando de la lista de nodos vivos a los que murieron y están siendo reseteados.

3.5.2.2 Container Manager

La lógica de este proceso es incluso más sencilla, y se centra en encapsular la interacción con Docker:

1. Esperar mensajes del monitor.
2. Cuando recibe la petición de resetear un nodo, intenta hacerlo siguiendo una política de timeouts y retries hasta que el nodo esté en estado `running`.
3. Confirmar el reinicio del nodo al monitor.
4. Repetir proceso.

3.5.2.3 Detalles de implementación

Algunos detalles de implementación que se consideran interesantes:

- Para la comunicación con los distintos nodos a monitorear, se establecen conexiones TCP persistentes, utilizando sockets **zmq** de tipo REQ-REP.
- Se utilizaron en total dos procesos: el monitor principal, y un **ContainerManager**.
- Ambos procesos se conectan mediante pipes, siguiendo el paradigma de mensajes.
- Se decidió utilizar dos procesos en lugar de uno único para poder independizar el tiempo que tarda resetear un container, y así mantener la frecuencia de monitoreo en el proceso principal lo más estable posible.

4. Conclusiones

Consideramos que el presente trabajo resultó muy interesante, ya que incluso dejando de lado la mayoría de problemas que pueden existir en un entorno real, y limitándose a atacar sólo los más sencillos, se tomó total **contacto con las implicancias que presenta un sistema distribuido** a la hora de buscar consistencia y alta disponibilidad. Gracias al empleo de distintas estrategias y algoritmos se pudo corroborar desde la práctica el teorema CAP, para finalmente encontrar soluciones que permitan lograr lo que se buscaba.

Resultó una parte fundamental también el hecho de aceptar que al desarrollar un sistema de estas características, introducir tolerancia a fallos **inevitablemente presenta consecuencias sobre la performance** del producto final, siendo este overhead notorio pero necesario para garantizar los objetivos.

Por otro lado, se considera valioso también el contacto y el enfoque que se tuvo para con las **herramientas de concurrencia**, esto es: normalmente suele utilizarse una herramienta sin cuestionarse mucho el por qué ni tampoco si es la mejor alternativa, mientras que para este trabajo, resultó esencial analizar cada una para cada caso a fin de tomar la decisión que a nuestro criterio era la más acertada. Esto se ve reflejado en que el trabajo final cuenta con **RabbitMQ, ZMQ, raw sockets TCP, pipes, procesos, mutexes y threads**.

Para finalizar, estamos muy conformes con el trabajo realizado y con lo aprendido del mismo, así como con los posibles puntos de mejora que nos llevamos para tener en cuenta en cualquier desarrollo posterior.