Informe: Sistemas Distribuidos I

Introduccion

El codigo de este proyecto utiliza un sistema distribuido para analizar dos datasets de la plataforma Steam, y responder cinco querys diferentes.

Correr el proyecto

Tomando como referencia el directorio base en donde se descarge el proyecto:

Levantar el sistema

1. `docker compose build`
2. `docker compose up`

Bajar el sistema

`docker compose down`

Limpieza del entorno

`docker system prune -a --force --volumes`

Iniciar Chaos Monkey script

1. `cd scripts`
2. `python3 kill_containers.py --min=45 --max=75`

Ejecutar script de resultados esperados

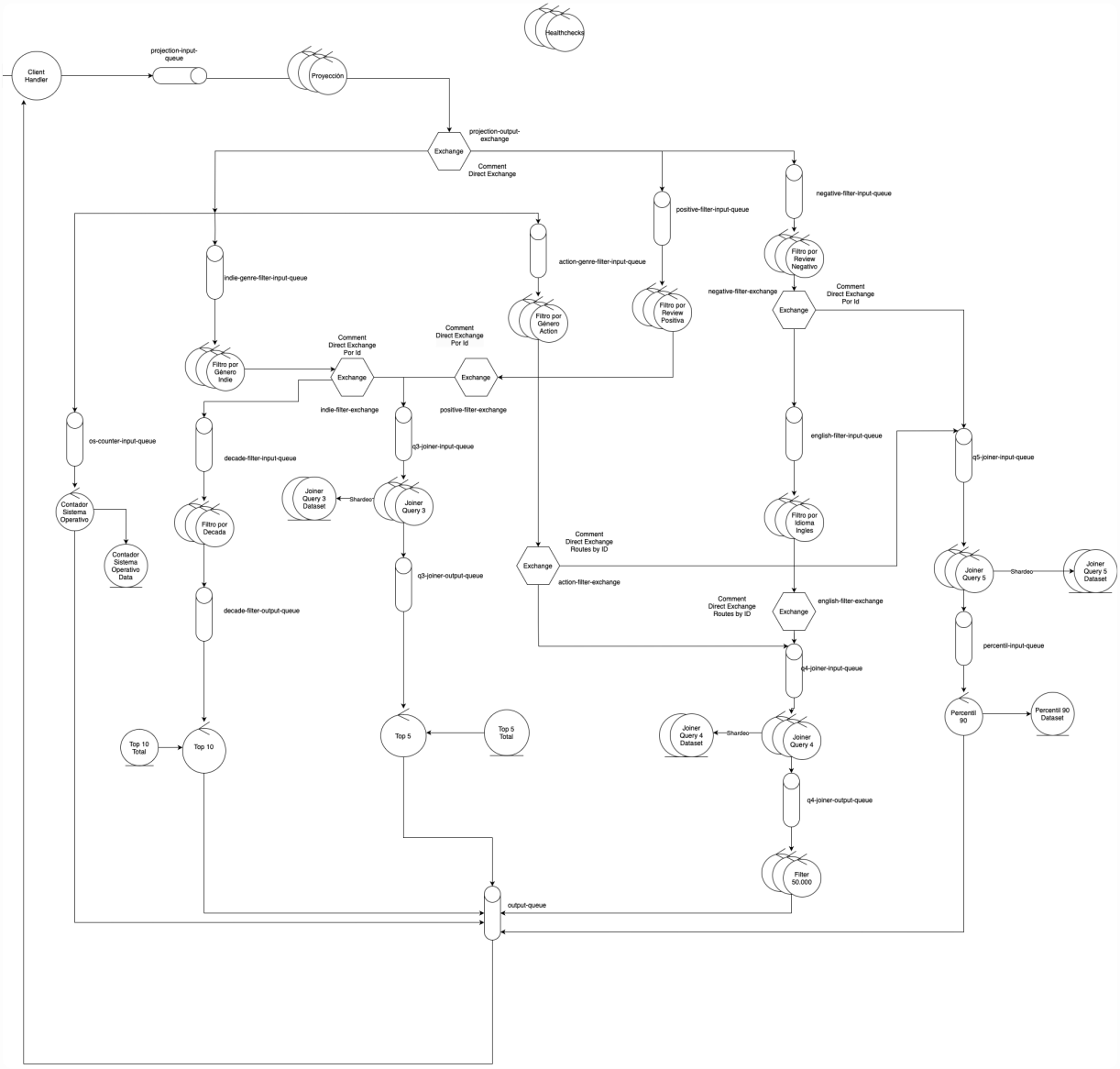
1. `cd scripts/query_solver`
2. `poetry install`
3. `cd ..`
4. `./query_solver.sh <output_file> <games_file> <reviews_file>`

Ejecutar script de comparacion de resultados

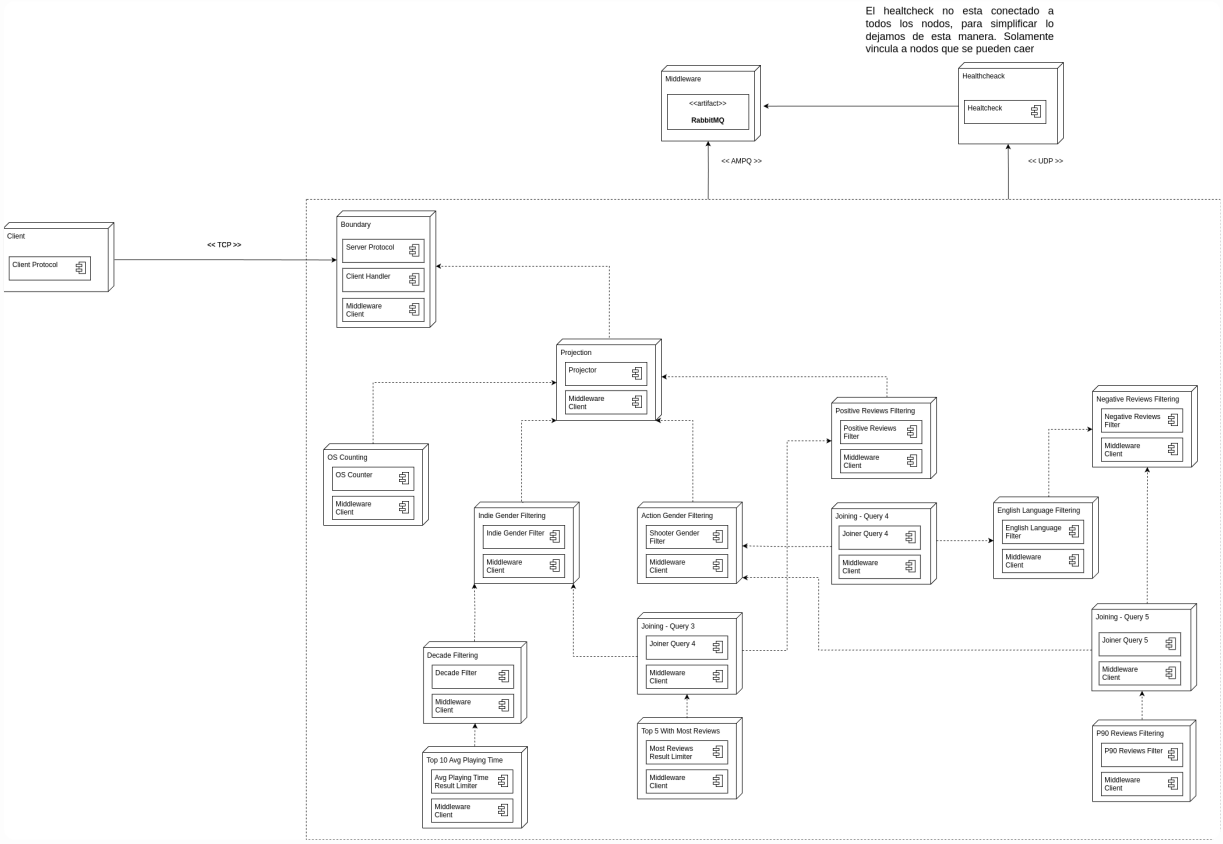
Luego de ejecutar el script anterior guardar el path del archivo resultante
y ejecutar

1. `cd scripts`
2. `python3 diff/main.py --first <expected_file> --second <got_file>`

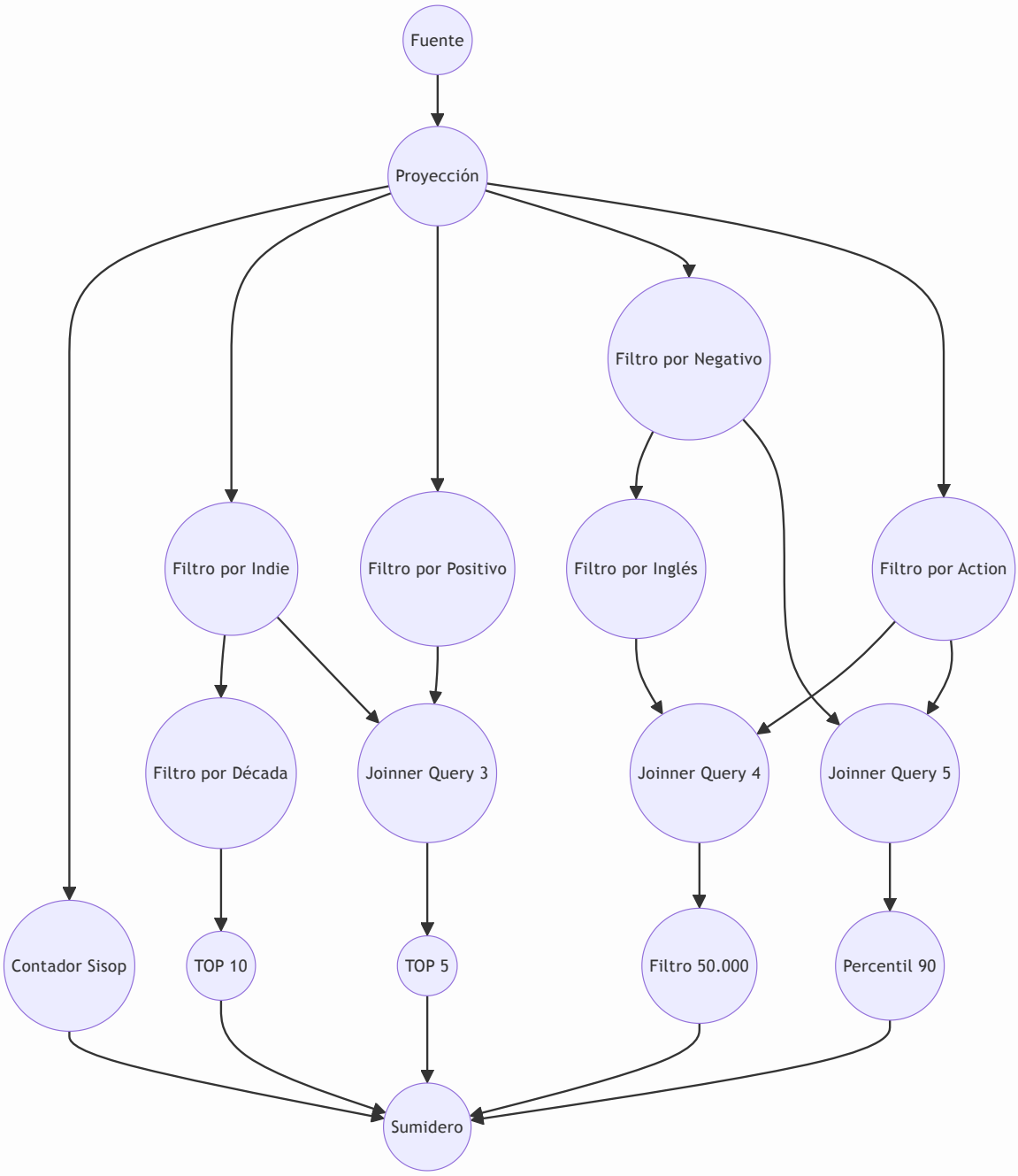
Diseño



– Diagrama de despliegue


El healtcheck no esta conectado a todos los nodos, para simplificar lo dejamos de esta manera. Solamente vincula a nodos que se pueden caer
Middleware
<<artifact>>
RabbitMQ
Healtcheack
Healtcheck
<< AMPQ >>
<< UDP >>
Client
Client Protocol
<< TCP >>
Boundary
Server Protocol
Client Handler
Middleware Client
Projection
Projector
Middleware Client
OS Counting
OS Counter
Middleware Client
Indie Gender Filtering
Indie Gender Filter
Middleware Client
Action Gender Filtering
Shooter Gender Filter
Middleware Client
Positive Reviews Filtering
Positive Reviews Filter
Middleware Client
Joining - Query 4
Joiner Query 4
Middleware Client
English Language Filtering
English Language Filter
Middleware Client
Negative Reviews Filtering
Negative Reviews Filter
Middleware Client
Joining - Query 3
Joiner Query 4
Middleware Client
Joining - Query 5
Joiner Query 5
Middleware Client
Decade Filtering
Decade Filter
Middleware Client
Top 5 With Most Reviews
Most Reviews Result Limiter
Middleware Client
P90 Reviews Filtering
P90 Reviews Filter
Middleware Client
Top 10 Avg Playing Time
Avg Playing Time Result Limiter
Middleware Client

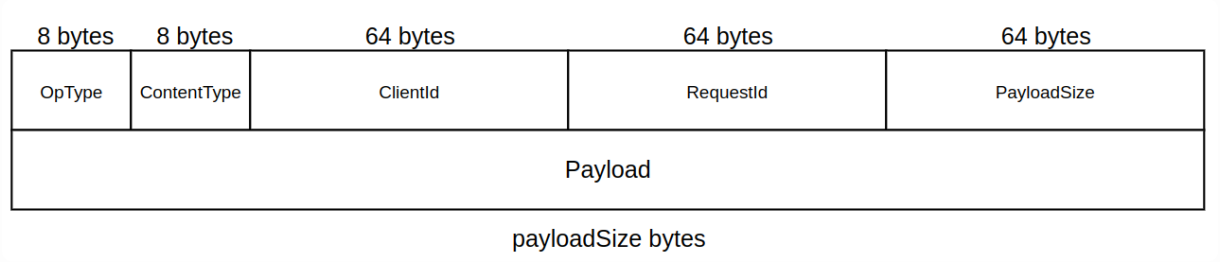
– DAG



Protocolos

Cliente boundary

A continuación se presenta la estructura de todos los paquetes que viajan entre cliente y boundary.



- OpType: Representa el tipo de operación que se quiere realizar
  - 0: Inicio de la transmición de datos
  - 1: Envio de datos
  - 2: Envio de resultados
  - 3: Mensaje de sincronización
  - 4: Mensaje de acknowledge a la sincronización
  - 5: Fin de la transmición de datos
- ContentType: Representa el contenido del mensaje en el payload
  - 0: Sin contenido
  - 1: Payload con juegos
  - 2: Payload con reseñas
  - 3: Payload con resultado de query 1
  - 4: Payload con resultado de query 2
  - 5: Payload con resultado de query 3
  - 6: Payload con resultado de query 4
  - 7: Payload con resultado de query 5
- ClientId: Id del cliente
- RequestId: Id de la petición
- PayloadSize: Tamaño de los datos que viajan en el payload

Para finalizar, se destaca el endianess para el protocolo, los campos de 64 bytes deben viajar en big endian.

Interno

El protocolo interno es aquel que se usa para la comunicacion entre controladores. Los mensajes dentro del protocolo, son todos mensajes binarios, codificados en little endian, los cuales tienen la siguiente forma

MessageType	ClientID	RequestID
MessageID	PayloadSize	
Payload		

el campo *MessageType* codifica una gran cantidad de informacion en un byte. Si enumeramos los bits de la siguiente forma

b7	b6	b5	b4	b3	b2	b1	b0

notamos lo siguiente:

- b0 y b1 me dicen el tipo de mensaje que es
  - b1b0=0b00, MessageType = END
  - b1b0=0b01, MessageType = Data
  - b1b0=0b10, MessageType = Results
- Si MessageType = Data y b2 = 1 entonces el payload contiene juegos
- Si MessageType = Data y b2 = 0 entonces el payload contiene reviews
- Los ultimos bits (b3,b4,b5,b6 y b7) me indican si el MessaType es Result, el numero de la query a la cual corresponden los resultados
  - Si esta en alto b3 entonces los resultados se correponde a la query 1
  - Si esta en alto b7 entonces los resultados se corresponden a la query 5

Luego los campos, ClientID, RequestID, MessageID indican respectivamente el id del cliente que solicito las querys, un numero reservado para uso interno, y el id asignado al mensaje (este id se lo asigna el boundary)

Por ultimo tenemos el payload, cuyos contenidos son elementos codificados en little endian, que pueden ser de los siguientes tipos:

- Int32
- Float32
- Byte
- Array<Byte>

Middlewares

Nuestro middleware sobre **rabbitmq** gira en torno a la abstraccion del `IOManager`, la cual se encarga de todo lo relacionado a la configuracion del entorno de rabbit y, la creacion y administracion de colas. Cada `IOManager` tiene una conexion de input y otra de output.
Las conexiones de inputs:

- `FanoutSubscriber`
- `InputWorker`
- `DirectSubscriber`

Las conexiones de output:

- `OutputWorker`
- `FanoutPublisher`
- `DirectPublisher`
- `Router`

Todos estos tipos de conexiones son configurables y se mapean a algun concepto de rabbit, pero reduciendo el boilerplate. Un caso particular es la salida de tipo *Router*, la misma es la que nos permite dirigir ciertos mensajes a una cola particular a traves de un tag y una estrategia de direccionamiento. Un ejemplo de estrategia puede ser round robin o un routeo por el hash del tag. Este tipo de conexion nos permitio realizar sharding en ciertos nodos.

Otra de las utilidades provistas por el middleware es la capacidad de de batchear mensajes provienentes de una cola, esto se logra haciendo acknowledge de multiples mensajes, a diferencia de hacerlo uno por uno.

Ademas de el middleware de rabbit creamos otro para la eleccion de lider, y el manejo del **coordinator**. Al crearlos como middlewares, nos permite agregar la lógica de estos servicios, sin la necesidad de cambiar la lógica del controlador. El mismo controlador no conoce, al igual que no interactua, con estos últimos 2 midlewares, permitiendonos agregarlos/modificarlos sin tener que cambiar el controlador en si. Por ultimo, proveen una manera de coordinar el cierre correcto del programa.
Todos los parametros de los middlewares son configurables por variable de entorno.

Controladores

En esta sección se lista cada uno de los controladores y una pequeña descripción del mismo:

- `Boundary`: Punto de entrada al sistema. Encargado del manejo de la conexión con el cliente.

- `Proyección`: Encargado de la traducción entre el protocolo externo utilizado por el cliente con el protocolo interno. Tambien esta a cargo de extraer solos los datos necesarios para el sistema. Este controlador no contiene estado.
- `Filtro`: Controlador a cargo de filtrar "Juegos" y "Reviews" en base a diferentes condiciones, como `Género` del juego, `Idioma` del review, etc. Este controlador no contiene estado.
- `OsCounter`: Controlador encargado de contar la cantidad de juegos que hay para cada sistema operativo. Este controlador contiene estado.
- `Top10`: Controlador encargado de calcular el Top 10 juegos. Este controlador contiene estado.
- `Top5`: Controlador encargado de calcular el Top 5 juegos. Este controlador contiene estado.
- `Joiner`: Controlador encargado de unir los games y reviews, parecido a un `Inner Join` en una base de datos relacional. Este controlador contiene estado.
- `Percentil`: Controlador encargado de calcular el percentil de un conjunto. Este controlador contiene estado.
- `Coordinator`: Controlador encargado del manejo de la sincronización para el envio de la señal de finalización del envió de datos. Este controlador contiene estado.
- `Healtcheck`: Controlador encargado de la supervisión de los otros controladores y reiniciar el mismo en caso de caidas.

Manejo de multiples clientes

Para soportar multiples clientes el nodo "boundary" que une el sistema con el cliente debe generar un client id para separar los flujos de datos de cada cliente. Esto se logra con el protocolo de comunicación, el cual establece antes de comenzar la transferencia de datos un id para el cliente mediante un *handshake*, el cliente inicia la conexión, el servidor responde al cliente con un id y el cliente comienza la transmisión de datos incluyendo en cada paquete enviado su id.

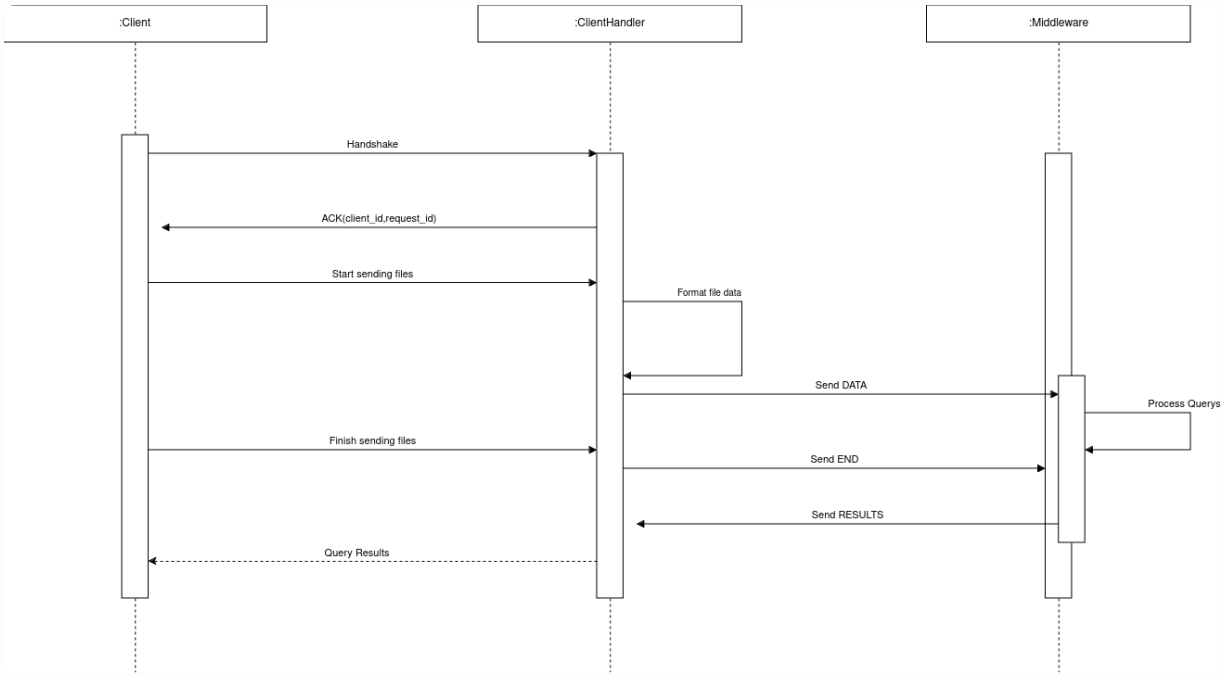
El "boundary" a su vez almacenará el estado de los clientes con el par **<client_id, connection>**, de esta forma identifica a que cliente pertenece la vuelta de los resultados al nodo y a su vez recupera la conexión TCP que espera los resultados.

Finalmente el "boundary" a su vez esta formado principalmente por 3 etapas, esto permite evitar que se bloquee esperando clientes, recibiendo mensajes o enviando resultados.

Etapas:

1. Lanzar una corutina con un proceso que espere resultados de RabbitMQ y envie al cliente.

2. Lanzar una corutina con un proceso que reciba mensajes del cliente y envie los datos a RabbitMQ.
3. Lanzar una corutina por cada cliente aceptado por el "boundary".



Manejo de estados en controladores

Para gestionar los estados en los controladores se optó por replicar un modelo en cada uno de ellos, existe una estructura interna en los controladores que mantiene todo el estado que debería tener un cliente, adicionalmente se mantiene una relacion entre el id de los cliente y su estructura de estado correspondiente, es decir, los controladores saben a que cliente corresponde cada estado según el client id que se propaga desde el "boundary". Esto se almacena en una estructura interna llamada *Store* que es clave para el funcionamiento de los nodos con estado, esta estructura se declara e inicializa en el momento en que se levanta cada controlador y en caso de falla, si existen datos persistidos en disco entonces seran agregados a la misma.

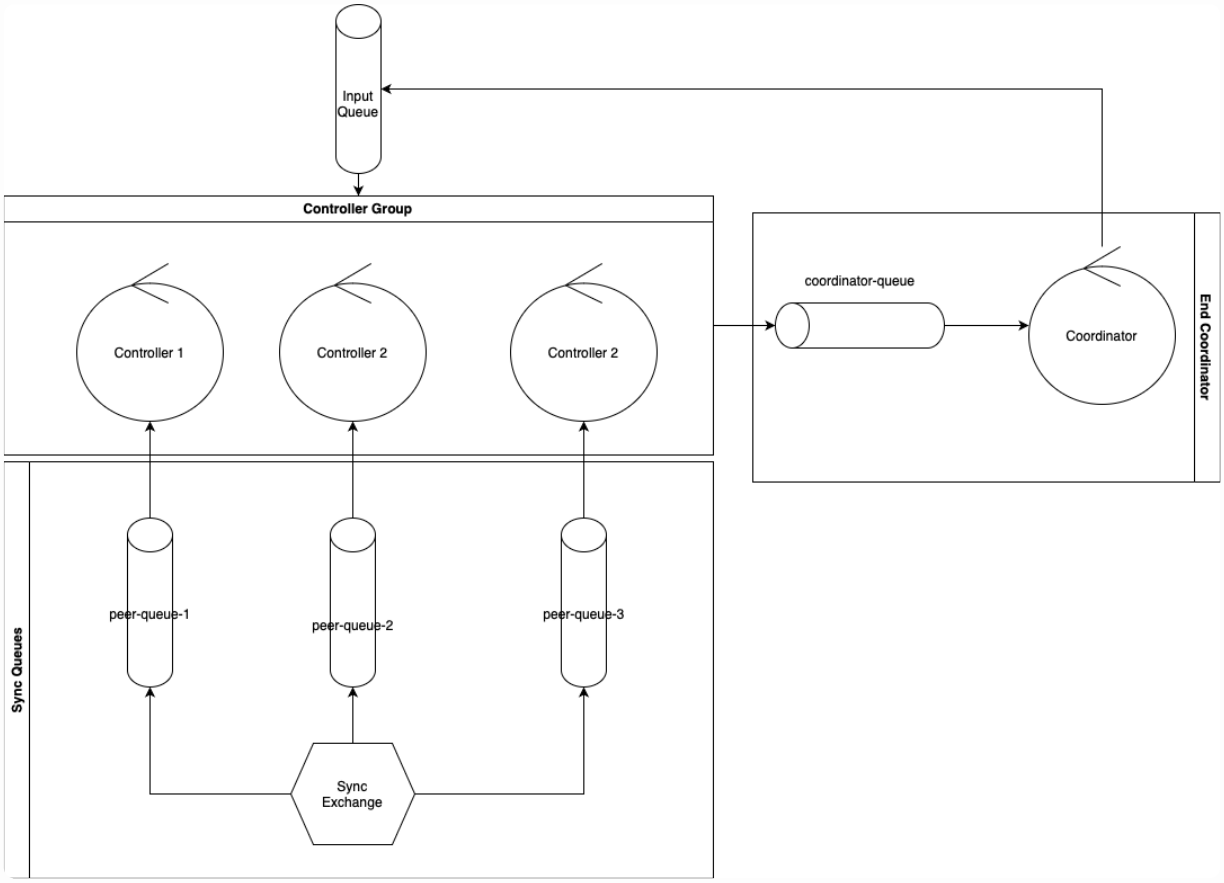
Como caso especial existe un id adicional que viaja en todos los mensajes del sistema y se genera en el "boundary", éste es el message id, es único para todos los mensajes que se generan y ayuda a identificar cada review que existe para cada juego ya que es una relacion 1 a n.

Finalmente, como se menciona en la sección de recuperación, lo que se almacena en los nodos son los mensajes que se van recibiendo en las colas, y se guardan segun su app id y el calculo de los resultados no ocurre hasta el momento en el que se terminen de enviar todos los mensajes de un flujo. Esto nos ayuda a garantizar idempotencia aunque nos limita la posibilidad de hacer optimizaciones.

Sincronización de Ends

Para el manejo y propagación del mensaje que señaliza la finalización de un archivo (al que llamaremos `END`) se utilizo un controlador para el manejo de la sincronización entre las distintas réplicas de un nodo y la propagación del mensaje. Se decidio crear un controlador para esta funcionalidad para evitar que cada uno de los controladores de la misma familia (nodos que cumple la misma función y estan escalados) propagen el mismo mensaje de finalización, causando que los nodos inferiores deban, o manejar estos duplicados, o esperar una cantidad arbitraria de mensajes para saber que pueden propagar el mensaje de `END`.

El siguiente diagrama describe como es la arquitectura para un controlador génerico:



Como se puede ver, hay varias `queues` de comunicación, una a la que llamamos **coordinator-queue** usada para que los controladores notifiquen al coordinador de que les llego un mensaje de `END` y la colas de `peer-queue` utilizadas entre los controladores para sincronizarse.

Tambien se encuentra el controlador **coordinator**. Este esta encargado de contar la cantidad de `END` por `ClienId` que va recibiendo para saber cuando propagar el mensaje. Para evitar duplicados, cada mensaje de `END` que se envia al **coordinator** contiene un `ClientId` y un `NodeId` que identifica el cliente y que nodo le mando ese mensaje, haciendo que el estado del **coordinator** sea idempotente. Si por alguna razón, un

controlador envia 2 veces el mismo mensaje, el estado del **coordinator** no cambia, evitando propagar el mensaje de `END` sin la sincronización correcta.

El algoritmo que se sigue es el siguiente:

1. Llega un mensaje `END` a la `Input Queue` y uno de los controladores toma este mensaje.
2. El controlador que toma el mensaje de finalización, lo envia a travez de el `sync exchange` a todos los nodos dentro del `Controller Group`, incluido a si mismo.
3. Cuando al controlador le llega un mensaje de `END` desde un `peer-queue`, este controlador le notifica al controlador **coordinator** que ya recibio el mensaje de `END`. 4.El controlador va recibiendo estos mensajes hasta recibir, para ese `clientId`, los mensajes de `END` de cada cola. Una vez que tenga todos los mensajes, envia al `Input Queue` un mensaje de `END` llamado `FinalEnd`.
4. Cuando un controlador recibe un `FinalEnd`, y este tiene el `NodeId` 1, propaga un `End` hacia la cola output, señalizando que la sincronización termino.

> Hoy en día, solo el `NodeId 1` puede propagar el `FinalEnd`. Este comportamiento se puede mejorar al usar algun algoritmo de selección de líder para elegir que nodo debe propagar el mensaje. Por temas de tiempo y complejidad, decidimos que la mejor opción era utilizar esta implementación, con el pequeño costo en tiempo en caso de que el nodo con `NodeId` caiga y la propagación del mensaje se demore.

Una pregunta que puede surgir es, ¿Por que el **coordinator** propaga para "arriba" el `END` en vez de para abajo? Esto se debe a 2 puntos:

1. Cada controlador puede enviar los mensajes de forma distinta, como por ejemplo, routeando con un `Round Robin` o por `ClientId`. Si el **coordinator** propaga el `END`, deberia saber manejar cada caso. Si el mismo controlador es el que propaga el `END`, entonces se evita tener que hacer una lógica especial en el **coordinator** para cada controlador.
2. La decisión de usar el mensaje de `FinalEnd` fue causado por la siguiente pregunta ¿Que pasa si el nodo se cae mientras manda el `END` al coordinator? Imaginemos el caso de que el coontrolador esta procesando 50 mensajes, 49 del `ClientId` 1 y el ultimo el `END` de ese `ClientId`. Si falla despues de enviar el `END` y antes del `ACK` del batch, todos los mensajes que vuelven a la cola y a volver a procesarse, casuando que el **coordinator** propage el `END` a pesar de que todavia hay mensajes a procesar de ese `clientId`.

Finalmente, tenemos que analizar el casos del controlador **Joiner**, debido a que este controlador no solo debe mandar el `END`, si no tambien propagar el resultado acumulado. Para esto se agrega otra etapa de sincronización:

1. Llega un mensaje `END` a la `Input Queue` y uno de los controladores toma este mensaje.
2. El controlador que toma el mensaje de finalización, lo envia a travez de el `sync exchange` a todos los nodos dentro del `Controller Group`, incluido a si mismo.
3. Cuando al controlador le llega un mensaje de `END` desde un `peer-queue`, este controlador le notifica al controlador **coordinator** que ya recibio el mensaje de `END`.
4. El controlador va recibiendo estos mensajes hasta recibir, para ese `clientId`, los mensajes de `END` de cada cola. Una vez que tenga todos los mensajes, envia al `Input Queue` un mensaje de `END` llamado `FinalEnd`.
5. Cuando un controlador recibe un `FinalEnd`, envia su estado hacia la cola de output y le notifica al **coordinador**.
6. Una vez que el **cordinator** recibe la confirmación de cada uno de los nodos, propaga un mensaje `JoinerEnd` a la cola de `Input Queue`.
7. Cuando un controlador recibe un `JoinerEnd`, y este tiene el `NodeId` 1, propaga un `End` hacia la cola output, señalizando que la sincronización termino.

Se agrego este último paso ya que debemos asegurar que todos los mensajes fueron enviados antes de mandar el `END`. De lo contrario, el controlador de "abajo" puede no recibir todos los resultados antes del `END`, causando una pérdida de datos y la obtención de un resultado no correcto.

Tolerancia a fallas

Algoritmo de manejo de fallas

Antes de explicar el algoritmo, vamos a destacar un punto que nos ayudo en el desarrollo del mismo, el cual fue convertir a todos los nodos que guardan estado en idempotentes, de forma tal de que la generacion de duplicados no nos afecte en estos nodos. A continuacion explicamos el algoritmo

Recuperación
Para recuperar los nodos que continenen un estado en memoria ante una eventual falla, el algoritmo básico es el siguiente:

- Cuando un nodo esta iniciando lo primero que debe hacer es ir a leer su archivo de log en el cual los nodos bajan a disco los mensajes que se fueron recibiendo de RabbitMQ.
- Leído el archivo, si no existe es porque aún no hay nada que recuperar, por lo tanto el nodo continua el flujo normal de ejecución. En caso contrario el nodo comenzara a leer lo que esta en el log y a volcarlo nuevamente sobre una estructura en memoria destinada a mantener ese estado.

- Al finalizar el proceso de carga de mensajes en memoria, se retoma el flujo normal de ejecución. Habiendo así recuperado los datos que se encontraban previo a la falla y continuando con el procesamiento de nuevos mensajes.

Persistencia

Para persistir el estado de un nodo para su posterior recuperación lo que se debe garantizar es que la bajada de los datos a disco siempre sea con la última actualización del estado que el nodo debe realizar y previo a indicarle a RabbitMQ el *acknowledge* de los mensajes consumidos. Esto garantiza que:

- Si el fallo ocurre cuando el nodo se encuentra realizando operaciones de actualizacion o antes de estas, entonces, no va a haber nada en disco, los mensajes vuelven a la cola de RabbitMQ y nada se ha perdido.
- Si el fallo ocurre cuando el nodo ya realizo todas las operaciones de actualizacion de estado y además envia a disco los mensajes que lo generaron entonces tenemos la seguridad de que podemos reconocer a RabbitMQ estos mensajes para que los pueda eliminar de su cola. Si se diera el caso en que el nodo se cae antes de poder reconocer los mensajes a RabbitMQ entonces los mensajes los mensajes volverían a la cola y tendríamos duplicación de datos porque el algoritmo de recuperación se encargaría de cargar el estado de disco con los mismos mensajes que volverán de RabbitMQ (ver manejo de duplicados e idempotencia).
- Si el fallo ocurre luego de haber reconocido a RabbitMQ los mensajes, entonces no habrá problemas, ya que, en disco estaran los mismos y cuando RabbitMQ los elimine de su cola no se podrán perder.

```
Pseudocódigo:

for each message from rabbit.consume():
    .
    .
    newState = state.process(message)
    .
    .
    transactionLog.commit(newState)
    .
    .
    rabbit.ack(message)
```

Manejo de duplicados e Idempotencia

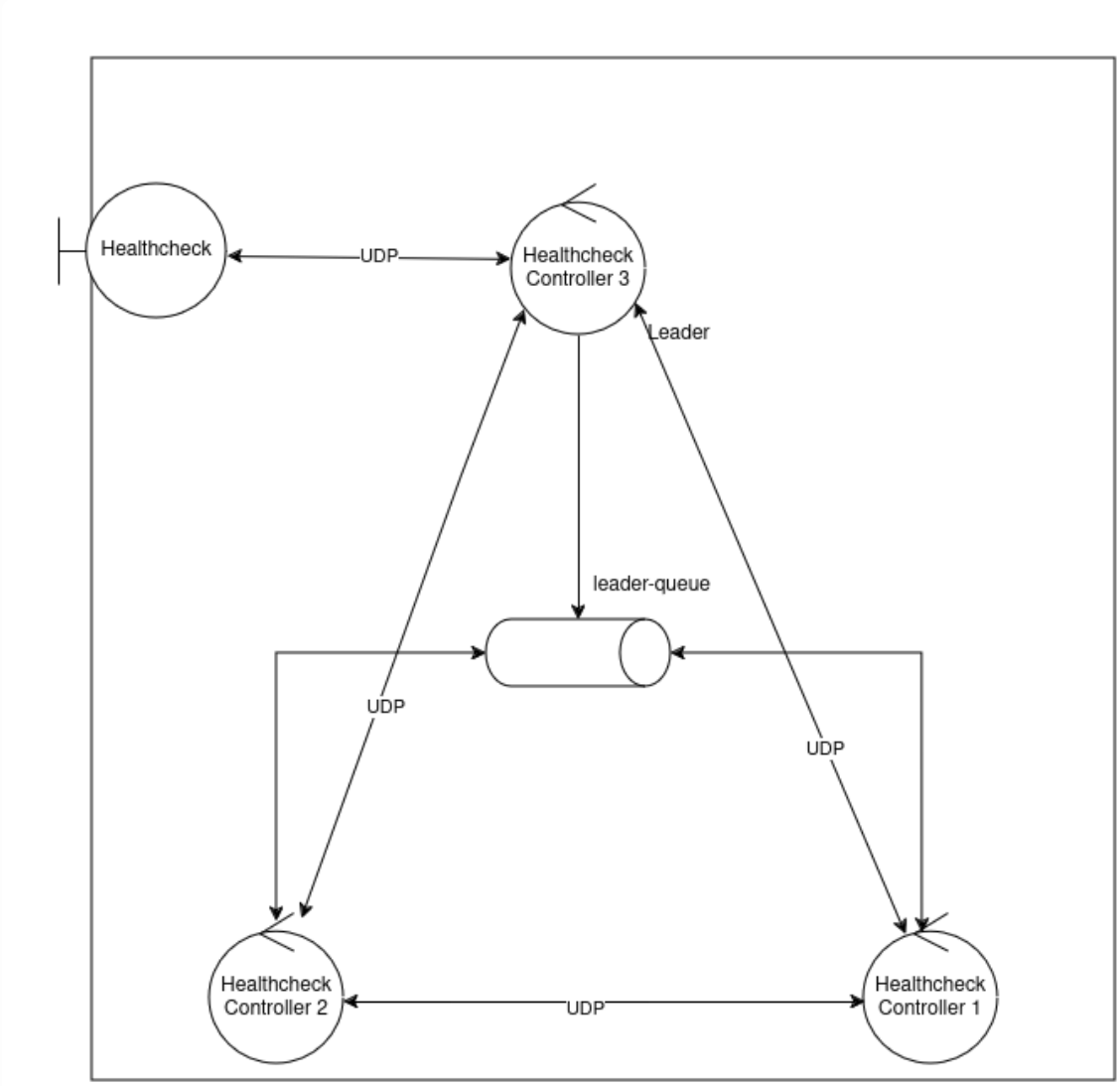
Pueden existir duplicados cuando un nodo persite datos a disco y no alcanza a reconocer los datos a la cola de RabbitMQ, haciendo que proximamente se inserten dos veces, una durante el proceso de recuperación y otra cuando se obtengan nuevamente los datos de RabbitMQ. Para evitar que esto suceda las operaciones se hicieron idempotentes.

Para hacer las operaciones idempotentes se almacenan en memoria cada uno de los datos que se estan procesando junto a un identificador único. El identificador para cada registro o dato es el id del juego del dataset de “juegos” ya que todos los datos que se transfieren entre controladores que necesitan un estado contienen este id. De esta manera se elimina el problema de duplicados pues el procesamiento se difiere hasta el final de la transmisión de datos y lo que se almacena en el estado de los nodos son los mensajes que se van recibiendo de RabbitMQ por `APPID`, logrando que solo se inserte en el estado una vez cada mensaje hasta el final de la transmisión, en donde comienza una etapa para calcular los resultados a transmitir al nodo "boundary" conectado con el cliente. La consecuencia de esta estrategia es una penalidad en el consumo de memoría pues pasamos a mantener todo el estado sin poder hacer optimizaciones de procesamiento.

Healthcheck

Para monitorear si un nodo esta caido se introdujo un nuevo controlador, el *Healthcheck*. El mismo usa UDP para enviar mensajes a los demas controladores, y verificar si siguen vivos, si no recibe respuesta en un periodo de tiempo, y una cierta cantidad de reintentos, asume que el nodo esta caido e intenta revivirlo. Para poder responder a estos mensajes los controladores tienen corriendo en un hilo de ejecucion aparte un servidor encargado de responder a los requests del healthcheck.

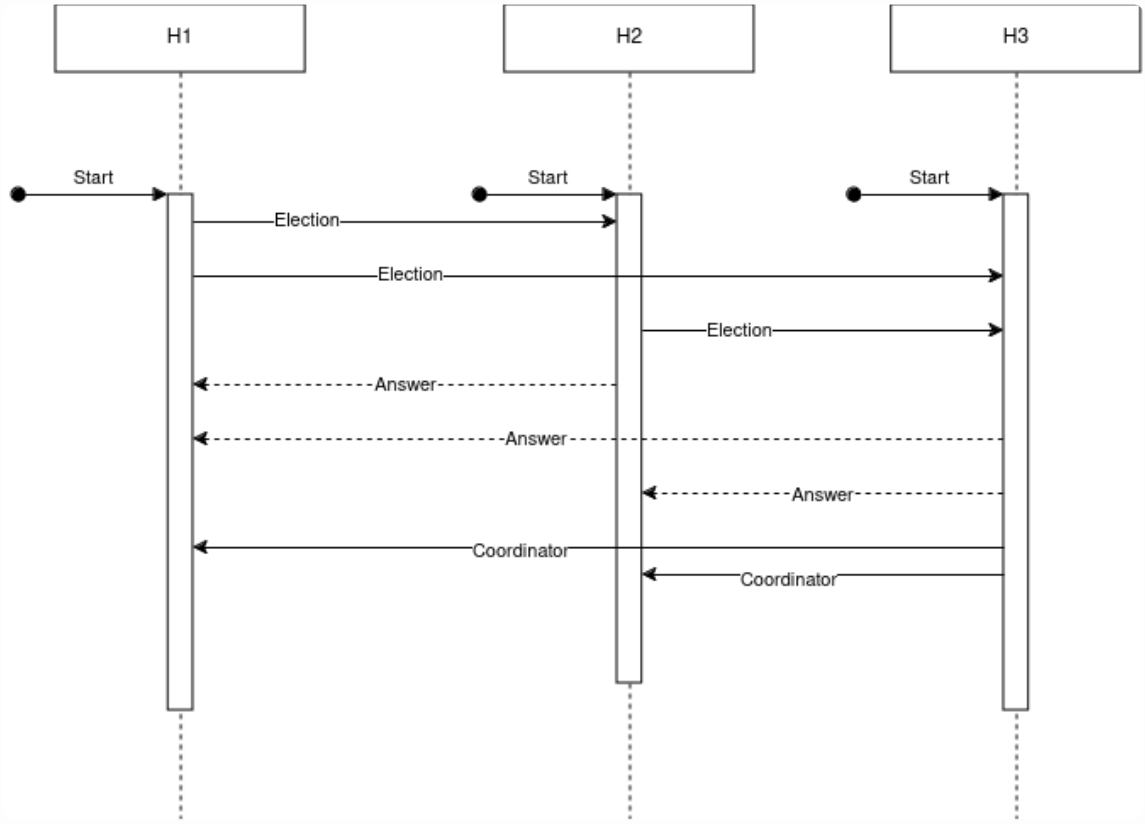
En el siguiente diagrama se puede ver que la comunicacion de los healthchecks con los demas nodos es a traves de UDP, al igual que la comunicacion entre los distintos nodos healthchecks para monitorearse entre si. Los mensajes de la eleccion se mandan a traves de la `leader-queue`. Nos decidimos por utilizar una cola para esta comunicacion, debido a que el algoritmo requiere de una conexion resiliente, y simplificaba el manejo de conexiones.



Algoritmo de lider

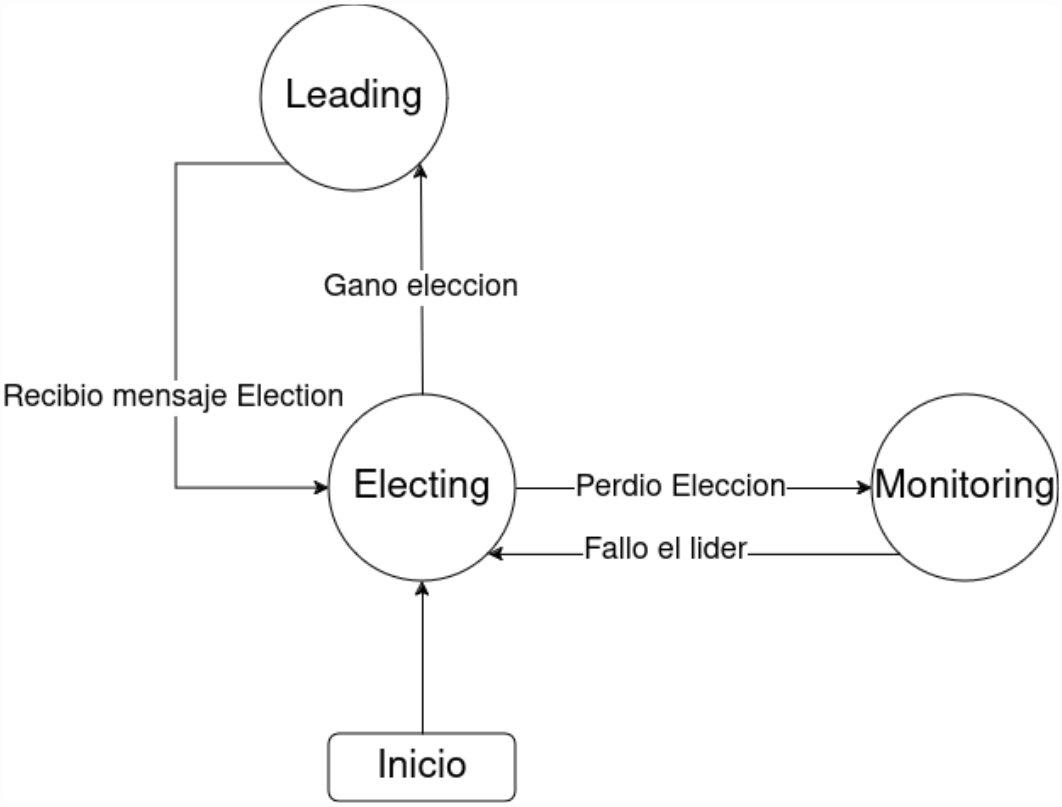
Los healthcheck al ser controladores tambien se pueden caer, entonces para hacer los mismos tolerantes a fallos se decidio por replicarlos. Ahora la pregunta es quien es el encargado, de todos estas replicas, de revivir y monitorear a los demas nodos; se utiliza un algoritmo de eleccion de lider, *Bully*, para elegir aquel unico nodo encargado de monitorear y revivir. Los demas healthchecks tienen la tarea de monitorear al lider y desencadenar una nueva eleccion si detectan que el mismo se cayo.

A continuacion se presenta un diagrama de secuencia del algoritmo



Primero notamos que H_i es el healthcheck numero i, donde i es el id numerico. El algoritmo comienza con los nodos enviandole a aquellos nodos con ids mayores un mensaje de *Election*, al cual responderan con un *Answer*,
si el nodo que envio el mensaje de *Election* no recibe una respuesta en un periodo determinado, asumira que es el lider y se lo notificara a los demas nodos a traves de un mensaje de *Coordinator*. Se puede ver en el diagrama que el nodo H_3 envia directamente un *Coordinator*, esto se debe a que detecto que es el nodo con id mas grande.

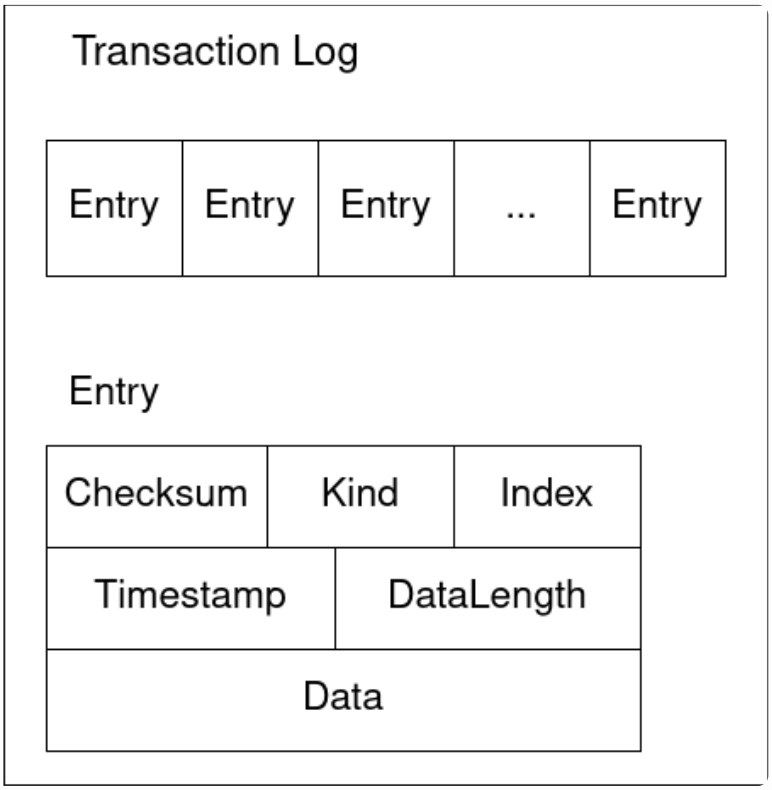
Por ultimo mostramos un diagrama de estados de los nodos durante su ejecucion



Persistencia

Transaction Log

Para persistir el estado de los distintos nodos en memoria se implemento un *Transaction Log*, con la siguiente forma



Un log no es mas que una lista de entries.

done cada nueva transaccion se agregara a la cabeza del log. La idea es que el controlador pueda reconstruir su estado aplicando las distintas transaccion guardadas en el log.

Escritura atomica en disco

La unica operacion de guardado en disco que es atomica en sistemas POSIX es el *Rename* de archivos. Por lo que el algoritmo que utilizamos para escribir atomicamente en disco es el siguiente

```
CREATE TEMP FILE
WRITE DATA TO TEMP FILE
SYNC TEMP FILE (Flushes kernel buffers)
CLOSE TEMP FILE
RENAME TEMP FILE TO FILENAME
```

Donde la idea es escribir a un archivo temporal y despues renombrarlo al archivo real, de manera de asentar los cambios.

Posibles mejoras