Trabajo Práctico 4

75.74 Sistemas Distribuidos I

Primer Cuatrimestre 2021

FIUBA

Nombre y Apellido	Padrón
Franco Giordano	100608
Julian Ferres	101483

Objetivos

Dados algunos datos sobre partidas del videojuego Age of Empires II, el sistema de alta disponibilidad planteado busca resolver 4 consultas particulares sobre el mismo, aprovechando conceptos de sistemas distribuidos:

- IDs de matches que excedieron las dos horas de juego por pro players(average_rating > 2000) en los servers koreacentral, southeastasia y eastus
- IDs de matches en partidas 1v1 donde el ganador tiene un rating 30% menor al perdedor y el rating del ganador es superior a 1000
- Porcentaje de victorias por civilización en partidas 1v1 (ladder == RM_1v1) con civilizaciones diferentes en mapa arena
- Top 5 civilizaciones más usadas por pro players (rating > 2000) en team games (ladder == RM_TEAM) en mapa islands.

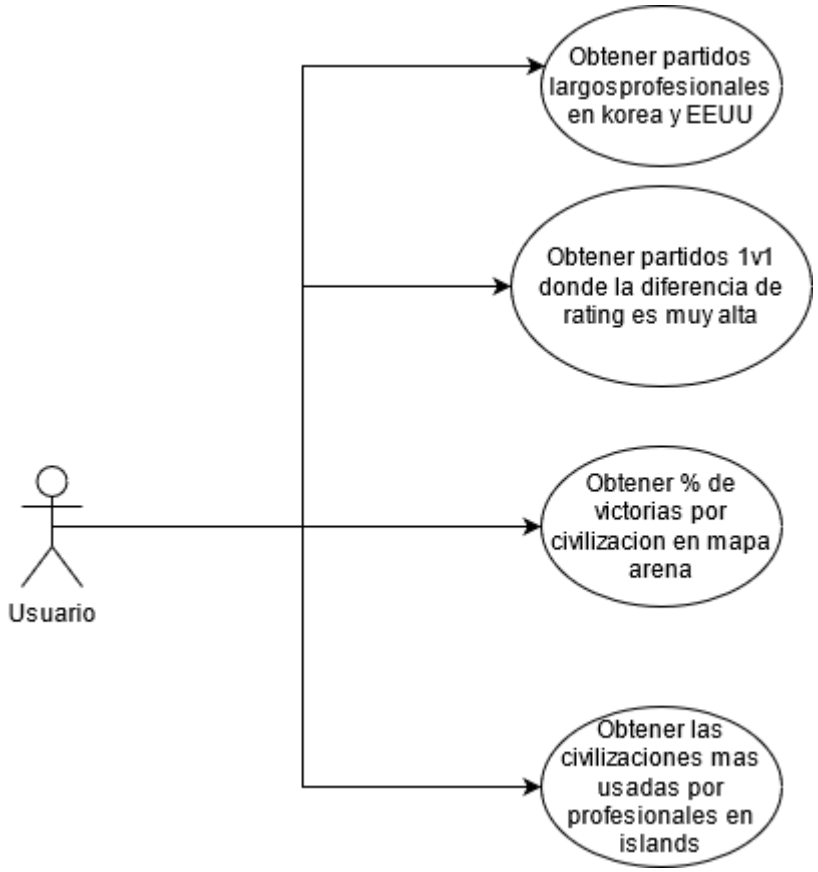
De ahora en más, nos referiremos a cada consulta como Query 1, 2, 3 y 4 para facilitar la lectura.

El sistema está planteado como uno de alta disponibilidad (HA), por lo que varias soluciones cambian con respecto a las planteadas en un sistema menos robusto. En el siguiente informe no solo se presentará qué problema resuelve el sistema, sino también las decisiones de diseño y protocolos implementados para asegurar una alta disponibilidad.

En la última sección Protocolos entraremos en detalle sobre el funcionamiento y propósito de los mismos.

Casos de Uso

Antes de comenzar, entenderemos el propósito del sistema mediante algunos casos de uso.

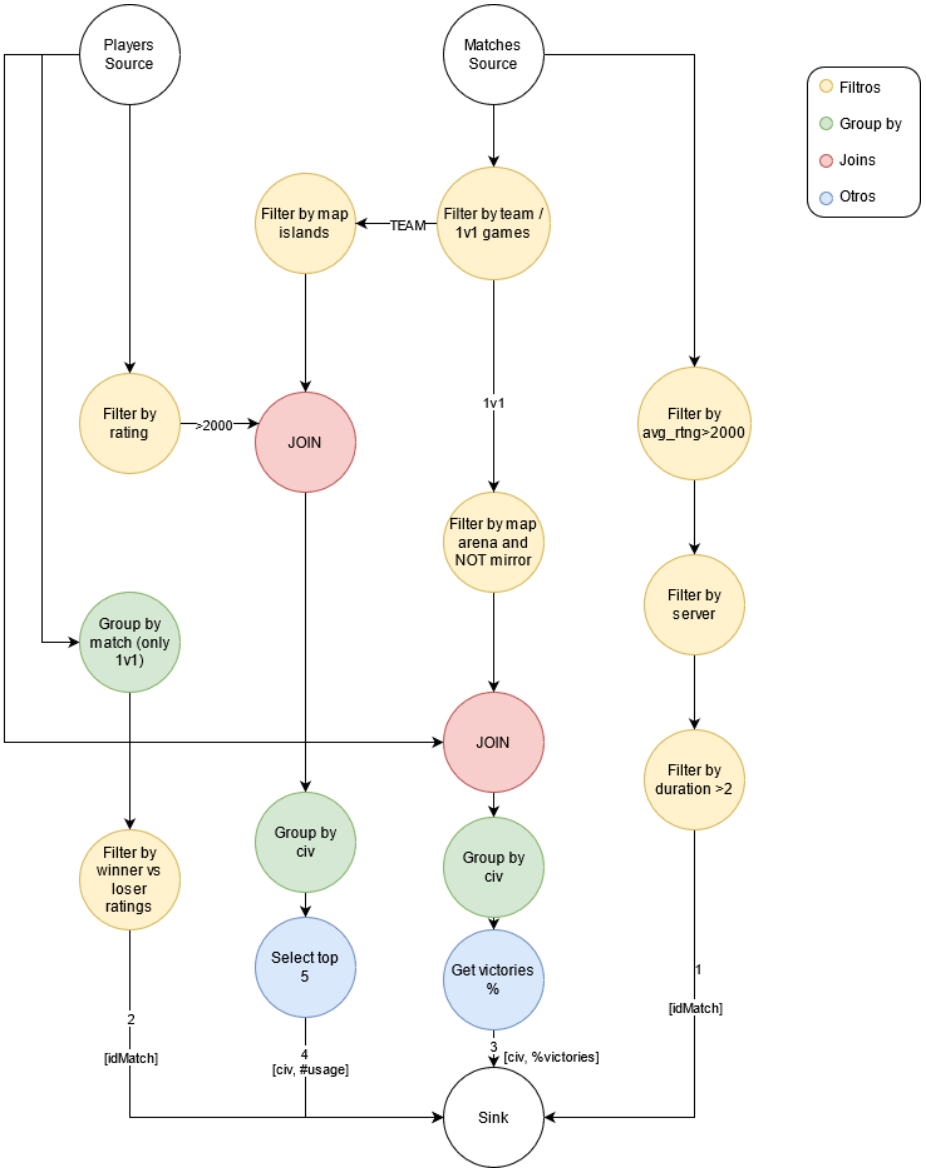


Vemos que un usuario de nuestro sistema busca usar el mismo para 4 casos distintos.
Estos mismos corresponden con las 4 queries planteadas en la sección anterior.
Para el usuario, en principio, no habrá mucha diferencia con respecto al trabajo práctico anterior, pues las consultas resueltas por el sistema seguirán siendo las mismas.

Vista Lógica

Entendiendo que usos intenta resolver nuestro sistema, vemos ahora el flujo de los datos a través del mismo a un alto nivel.

DAG



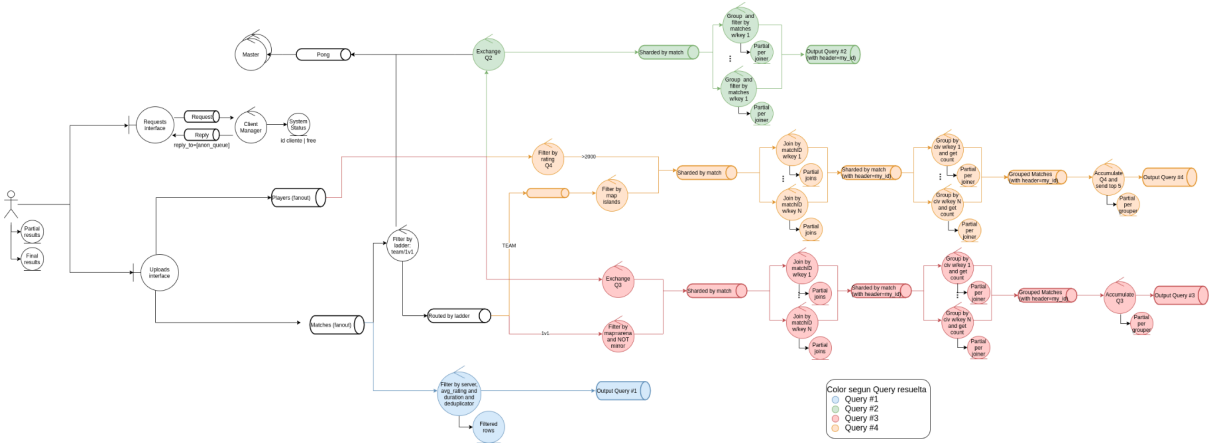
Se optó por filtrar lo más posible los datos al comienzo del procesamiento, con el objetivo de tener que realizar un join+groupby lo más ligero posible. Distinguimos con colores los roles que cumplen cada nodo en el flujo.

Vista Física

Buscaremos ahora comprender la topología del sistema planteado, desde las principales unidades de cómputo pensadas hasta el despliegue concreto efectuado.

Diagrama de Robustez

Trasladamos la vista lógica a términos concretos de nuestro sistema: hacemos ahora uso de múltiples controllers para distribuir las tareas. En algunos casos, como con la Query 1, múltiples filtros se colapsaron en una sola unidad para facilitar la implementación. En otros, como los Groupers de la Query 2, se ocupan de agrupar los datos además de filtrarlos según la diferencia de rating. En los casos de las Queries 3 y 4, se distribuyen las tareas de unión y agrupamiento en múltiples nodos para agilizar el proceso.

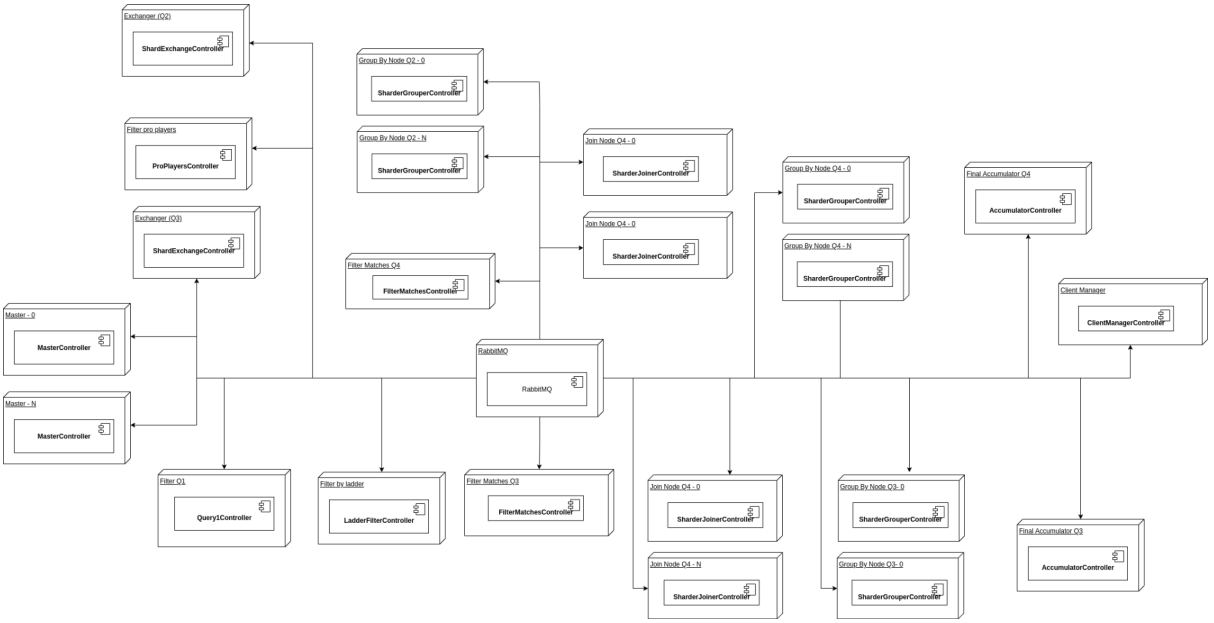


Para mantener la alta disponibilidad vemos como los nodos Masters, en particular el Líder, monitorea una cola Pongs donde periódicamente todos los nodos del sistema informaran su estado. El líder, en caso de no encontrar el pong (o heartbeat) de algún nodo, lo asume caído e intenta reiniciarlo aprovechando docker-in-docker.
Vemos que son algunos los nodos de nuestro sistema que requieren mantener un estado. Por ejemplo, ClientManager necesita saber si el sistema se encuentra libre u ocupado para atender consultas. Por otro lado, nodos como Joins y Groupers mantienen los datos recibidos para reconstruir los agrupamientos en caso de caídas y dar soporte al protocolo INICIO-FIN (explicado en la sección Protocolos).
Dado que nuestro sistema contempla potenciales caídas del Cliente, el mismo debe mantener los resultados parciales recibidos de su consulta para no perderlos. Una vez que reciba todos los resultados, los traslada a un archivo de Final Results exceptuando los mensajes de control para una fácil lectura.

Diagrama de Despliegue

Veremos ahora como distribuimos físicamente los nodos en las máquinas. En líneas generales, siempre optamos por multi-computing pues nos permite reutilizar la lógica a la hora de monitorear y levantar nodos. Recordemos que este reinicio de nodos se lleva a

cabo con docker-in-docker, por lo que resulta sumamente útil encapsular estos nodos lógicos como contenedores distintos.



Como se ve en el diagrama, el rol de RabbitMQ es vital para el sistema, pues cada nodo podrá transmitir y recibir información mediante las herramientas provistas por Rabbit. Notar como ningún contenedor se comunica directamente con otro, sino que siempre lo realiza mediante Rabbit. Esto no resulta deseable, pues trae complejidad innecesaria a tareas sensibles como el monitoreo de nodos, ya que agrega más puntos móviles a la comunicación. Queda entonces como trabajo a futuro transicionar dichas tareas a protocolos de comunicación más ‘ligeros’ como Sockets TCP/IP.

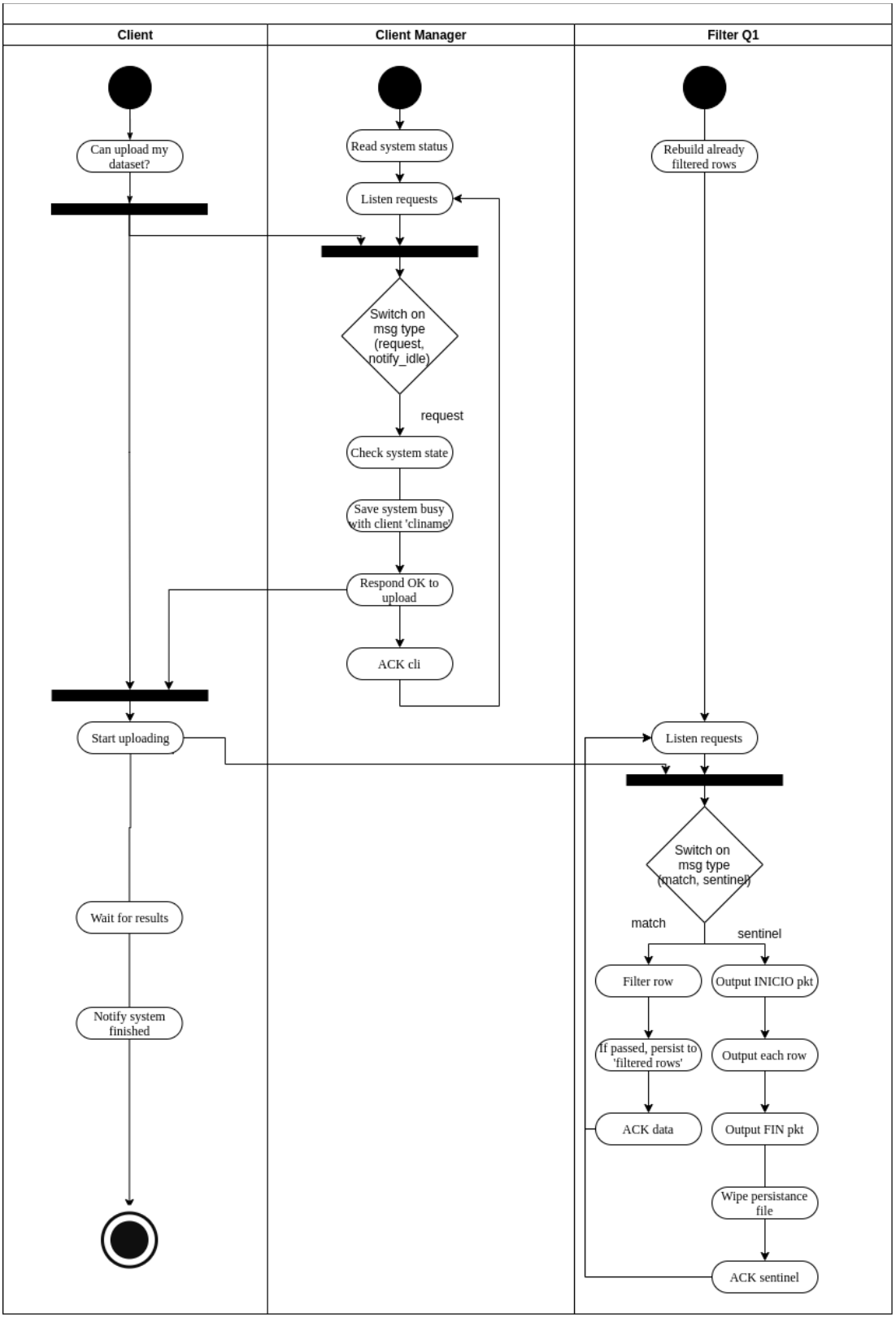
Para poder simular un esquema de persistencia durable, los archivos para persistencia están montados como volúmenes de docker con el sistema host. Ante eventuales fallas, estarán disponibles para el container en la misma ruta.

Vista de Procesos

Estudiaremos ahora la comunicación del sistema al momento de ejecutarlo, observando los flujos dentro del mismo. Notar que muchos de estos procesos se muestran de forma simplificada para facilitar la lectura. Muchos de los detalles de implementación pueden verse en la sección Protocolos.

Diagrama de Actividad - Cliente

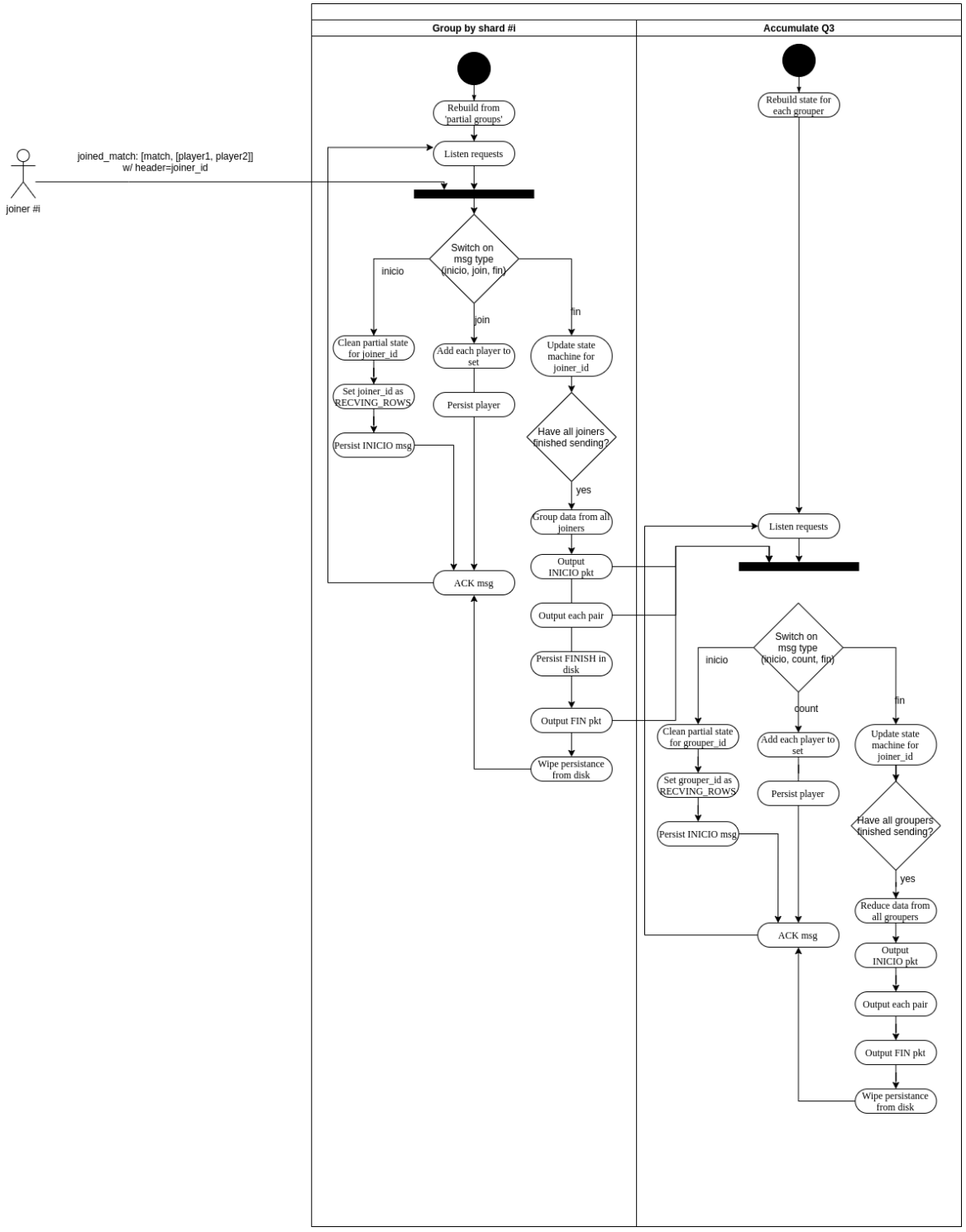
En este caso vemos el primer contacto del cliente con nuestro sistema, que se realiza mediante el Client Manager. De esta forma, pedirá una confirmación para comenzar a subir para no colisionar con otro cliente. Con fines didácticos, asumimos en el diagrama que el sistema se encuentra libre.



Nos enfocamos de momento solo en la Query 1. Vemos en particular cómo hace uso el Filtro Q1 del protocolo INICIO-FIN a la hora de emitir todos los resultados. Recordemos que este es el único filtro que debe mantener estado en el sistema pues es el último nodo antes de la interfaz final contra el cliente.

Diagrama de Actividad - Groupers Query 3

Estudiaremos ahora el flujo de los agrupadores para la query 3, que resultan muy similares a los de las query 2 y query 4. Comenzaremos el diagrama con el envío por parte de un Joiner i, lo cual puede contener los datos de un partido y todos los jugadores del mismo (como se ve en el diagrama), o paquetes de control indicando el INICIO y FIN.



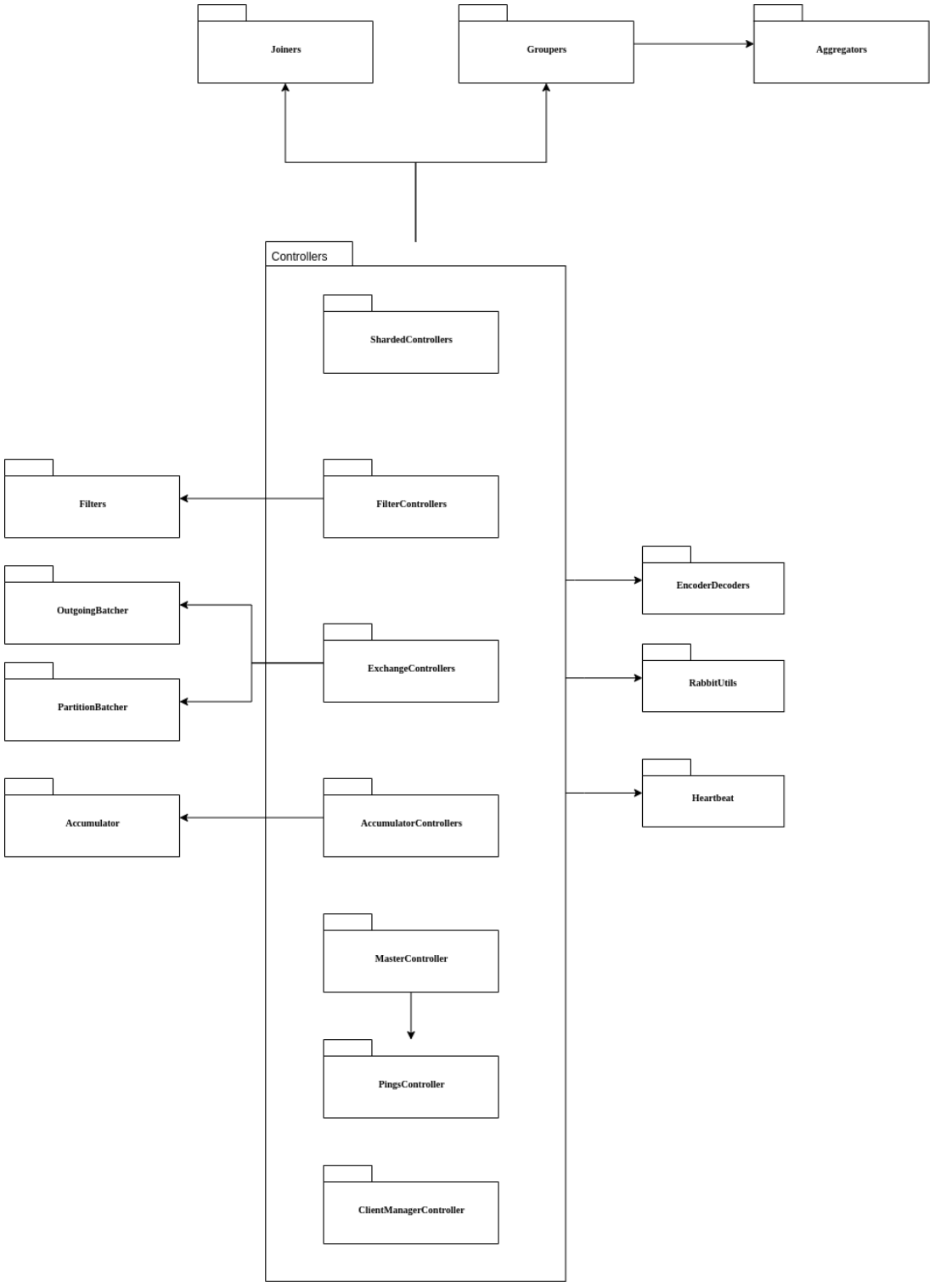
Para mantener el estado del protocolo del INICIO-FIN y soportar caídas, tanto los nodos group by como el accumulator mantienen en disco parte de la información. Así, pueden

mantener una máquina de estados para cada nodo anterior, permitiéndoles descartar información ante reinicios del protocolo INICIO-FIN.
Cabe destacar que en el caso de los nodos group by, si bien se encuentran shardeados y los joiners anteriores también, deben mantener una máquina de estados para cada nodo Join. Esto ocurre pues cualquier nodo Join puede enviar información a cualquier nodo Grouper. De esta forma, la cantidad de nodos joins elegidos no depende de la cantidad de nodos group que haya, brindando más flexibilidad al sistema.

Vista de Desarrollo

Veremos ahora, a grandes rasgos, los paquetes utilizados por la implementación y la organización de los mismos.

Diagrama de Paquetes



Tenemos paquetes encargados de serializar/deserializar la información (EncodersDecoders), controlar el flujo de recepción y envío de mensajes (controllers), y partes que colaboran con los controller o son parte del modelo, como Joiners, Filters, Accumulators, etc.
Tendremos por fuera los Heartbeats, encargados de avisar al Maestro que siguen vivos, y RabbitUtils, que encapsula gran parte de la lógica requerida para comunicarse con Rabbit.

Protocolos

Workers group-by de las querys 3 y 4

Los workers (shardeados) están encargados de recibir las filas previamente joineadas por alguno de los join worker, procesarlas según la civilización correspondiente, y luego despacharlas al accumulator correspondiente (es muy similar entre las querys 3 y 4, pero la 4 además de contar usos, tiene que contar también la cantidad de victorias).

En este caso, cada groupby mantendrá un archivo de persistencia por cada joiner (para poder borrar los mismos cuando llegue un [[INICIO]] del joiner correspondiente.

Además, de forma similar a la persistencia, se tendrá un diccionario, que primero se indexa por joiner_id, y luego por civilizacion.

De esta forma, en el caso de que algún joiner comience a mandar las filas nuevamente (podría haberse caído antes de terminar de mandarlas) se reseteara la información del joiner en cuestión, no del resto.

También se necesita saber en qué estado está cada joiner, para resetear, o simplemente ignorar algunos mensajes de control (INICIO, FIN) que puedan venir duplicados desde algún Joiner. Para eso, mantenemos un diccionario {id_joiner_i: state_joiner_i}, que tiene el estado de cada Joiner (se puede reconstruir con los mismos mensajes persistidos en caso de caída del Grouper). Los tres estados en los que puede estar un Joiner (es análogo en las demás estructuras [más-de-un-producer (cola) consumer]) son [WAITING_FOR_INICIO, RECVING_ROWS, FIN_RECVED].

Tenemos para los mismos, dos etapas distintas que tratar:

En la primera etapa, recibimos las filas, las procesamos (se agregan a un diccionario, indexado por joiner_id, y luego por nombre de civilización). Al mismo tiempo, persistimos el player que acaba de llegar (para que el nodo sea resistente a caídas). La persistencia se realiza con el persistor, que maneja la consistencia de la misma, en caso de que el nodo termine su ejecución en medio de una escritura.

Esta persistencia se realiza en el archivo correspondiente del Joiner emisor del mensaje, para poder borrarlo en caso de que se reciba un nuevo [[INICIO]] del mismo

Como el player es un objeto, se utilizan los encoders y decoders correspondientes para poder obtenerlo y persistirlo.

Las siguientes líneas representan el formato del archivo en disco y diccionario en memoria mantenidos, para un joiner_id en particular.

persistencia_del_joiner_i: [INICIO, CHECK, player1, CHECK, player2, CHECK, player3,...]
diccionario en memoria del joiner_id: {‘mongol’: set(player1, player2), ‘romano’: set(player3)}

Si una eventual caída del sistema ocurre, se puede reconstruir nuevamente el diccionario desde la información persistida.

Se pueden resumir las acciones del groupby en las siguientes:

1. Desencolo una fila (sin Auto ACK por eventuales caídas durante el proceso)
2. Proceso la fila (según el joiner_id del emisor, y la civilización)
3. Persisto la fila (con nuestro Persistor, en el archivo correspondiente al emisor)
4. ACK de la fila

En caso de caídas, como el último paso del protocolo es el ACK del mensaje, este se volverá a reprocesar, y la estructura en memoria permitirá eliminar players repetidos.

En la segunda etapa, se colapsan la información de los diferentes joiners y se envía al acumulator. Para que el accumulator pueda manejar varios group-bys a la vez, se envían mensajes especiales de INICIO y FIN, para poder delimitar las filas.

Se pueden resumir las acciones del groupby en las siguientes

1. Leer el último fin esperado de los joiners
2. Condensar, para cada civilización, los datos de los distintos joiners.
3. Enviar mensaje INICIO al accumulator
4. Envío filas
5. Persisto TERMINE (por eventuales caídas luego de mandar una vez el FIN)
6. Enviar mensaje FIN al accumulator
7. Vacío el archivo de persistencia
8. ACK del último mensaje

Joiner workers de las querys 3 y 4

En este caso, se controlará la cantidad de Sentinels esperados, y cuando arribe el último sentinel, se procedera a despachar las filas. Como las filas fueron previamente indexadas por el cliente (y hasta ahora no tuvieron ningún tipo de modificación interna, solo filtros y drops), se pueden guardar las mismas y el groupby, con ayuda de un set, se ocupará de de-duplicarlas.

El estado en memoria y la persistencia tienen la forma:

current_matches: {“match_id”: [match1, [player1, player2, player3]] }
persistencia: [player1, CHECK, match1, CHECK, player3,...]

Tanto el seguimiento de las filas como de los sentinels, debe tener una persistencia asociada para poder ser resistente a caídas. En este caso, persistimos las filas que nos llegan, así como también los sentinels. Con este esquema, podremos reconstruir el estado si hay alguna caída.

Se pueden resumir las acciones del joiner en las siguientes:

1. Desencolo una fila (sin Auto ACK por eventuales caídas durante el proceso)
2. Proceso la fila o sentinel (agregó según match_id al player correspondente)
3. Persisto la fila, o sentinel (con nuestro Persistor, en el archivo correspondiente al emisor)
4. ACK de la fila

En caso de caídas, como el último paso del protocolo es el ACK del mensaje, este se volverá a reprocesar, y la estructura en memoria permitirá eliminar players repetidos.

En la segunda etapa, se colapsan la información recibida y se envía al exchange de group-bys. Para que el accumulator pueda manejar varios group-bys a la vez, se envían mensajes especiales de INICIO y FIN para cada groupby (usando la sharding key correspondiente), para poder delimitar las filas, y que los mismos sepan cuando termina el dataset.

Se pueden resumir las acciones del groupby en las siguientes

1. Leer el último sentinel esperado.
2. Enviar mensaje INICIO a cada groupby
3. Envío filas segun sharding key
4. Persisto TERMINE (por eventuales caídas luego de mandar una vez el FIN)
5. Enviar mensaje FIN a cada groupby
6. Vacío el archivo de persistencia
7. ACK del último mensaje

Group-by query 2

Este grupo de controllers se comporta de forma similar a los group bys de las querys 3 y 4, pero tiene como diferencia que no espera filas joineadas de 2 datasets, sino que solo procesa filas de un dataset (por la naturaleza de la query en cuestión).

De esta manera, la entrada de información a los nodos se modeló de forma similar a los Joiners, solo que en este caso se procesan los datos de otra forma, solo se espera un sentinel.

Para la salida, se adoptó el mismo protocolo que en los groupbys de las querys 3 y 4, con el INICIO y FIN rodeando las filas procesadas, una vez que llega en sentinel. De igual manera, el resumen de acciones para despachar las filas es similar.

Client y Client-manager

Cuando un cliente ingresa al sistema, consulta mediante una cola (todos los clientes utilizan la misma) al Client-manager, sobre la disponibilidad del sistema.

Para esto utilizamos el protocolo descrito en el tutorial de RabbitMQ, sobre RPC

El client manager chequea si el sistema está libre, o si está siendo utilizado por el mismo cliente (el el proceso de envío de filas, el cliente pudo haberse caído). Este estado necesita ser persistido, por eventuales caídas del cli-manager.

Si el cliente recibe un mensaje de “SYS_BUSY”, simplemente no envía nada y termina su ejecución.

En caso de que el sistema estuviera libre, o siendo utilizado por el mismo cliente, el client manager autorizará al cliente a mandar los datasets.

Luego de enviar los datasets, el cliente lanza un proceso por query, para poder recibir las respuestas. Para saber cuando finalizan las filas respuestas, utiliza el protocolo de INICIO-FIN en cada una de las colas de salida.

El cliente tiene una persistencia parcial sobre las filas, para poder soportar las caídas.

Cuando recibe el FIN de alguna cola, dicho proceso termina.

Al recibir los FIN de todas las colas, el cliente procede a notificarle al sistema que se encuentra disponible, dejando al sistema de forma idéntica a como se encontraba antes del procesamiento.

Las acciones del client manager y cliente se pueden resumir en las siguientes:

Protocolo de cli manager

1. Desencolar una consulta
    a. Si es id_cliente, procede a ejecutar 2, debido a que hay un cliente requiriendo el sistema
    b. Si es para liberar el sistema, marca el estado del mismo como FREE y termina
2. Chequear en memoria si está disponible el sistema.
3. Persistir que ahora le corresponde el sistema a client_id (o se ignora si el sistema sigue ocupado por otro cliente)
4. Mandar respuesta al cliente (ya sea SYS_BUS o OK_TO_UPLOAD, en la cola en donde le indico al cli manager donde responder).
5. ACK al mensaje de request cliente

Protocolo de cliente:

1. Agregar ids únicos a filas de dataset
2. Generar cola anónima para respuesta del cli-manager

3. Mandar consulta a client manager con respecto a asi puedo utilizar el sistema (con su node_name, y la cola a la que debe responder el cli-manager)
4. Si leo que puedo utilizar el sistema
    - Comienzo a subir los datasets
    - Se lanza un proceso por query a escuchar los resultados del sistema
    - Cuando todos los procesos reciben los datos correspondientes, se notifica al cli-manager que el sistema está libre.

En caso de que el cliente tenga alguna caída:

- El client manager responde a una cola que ya no existe (porque es anónima y exclusiva para ese cliente) y el cliente eventualmente volverá a mandar otra solicitud. En este caso, como su id no cambia (se utiliza el nombre del container), se le permitirá utilizarlo.

En caso de que el client manager tenga alguna caída:

- Lo último que hace en su protocolo es el ack a la request del cliente. En tal caso, reconstruye el estado del sistema y procesa de nuevo la request. Si ya había enviado un msg previamente, el cliente (o rabbit, en caso de que el cliente ya haya terminado su ejecución) lo descarta.

Filtro Query 1

La mayoría de los filtros envía las filas que tienen la condición deseada a la cola de salida (a costa de duplicar dichas filas, en caso de que haya una eventual caída de dicho nodo entre el envío y el ACK a la cola de entrada

El filtro de la query 1 es ligeramente distinto, debido a que alimenta directamente a la cola de salida (es el primer y único paso que ocurre en el flujo de la query). Para evitar posibles problemas ante la caída del mismo, y a su vez no tener problemas de duplicación de filas, el mismo debe persistir las filas previamente procesadas.

Para deduplicar, utiliza en memoria un set con las filas ya procesadas (y aceptadas, debido a que perder las demás filas no implica problema alguno, de todas maneras serían descartadas), y en disco se almacenan a medida que arriban.

Se muestra brevemente el formato de las filas de la persistencia y el estado en memoria:

Persistencia: [fila1, CHECK, fila4, CHECK, fila6, CHECK]
Estado: filas_aprobadas = set(fila1, fila4, fila6)

Las acciones del filtro de la query1 se pueden resumir en las siguientes:

Protocolo fila del dataset:

1. Desencolar fila
2. Chequeo si no la procesé (set en memoria) y si cumple la condición
3. Persisto la fila (solo las que cumplen la condición, las otras no hay problema en perderlas)
4. ACK fila

Protocolo sentinel:

1. Mando INICIO
2. Mando fila a fila del set
3. Mando FIN

Accumulators (Query 3 y 4)

El acumulador está encargado de recibir las filas previamente agrupadas por los distintos grupos, colapsarlas, procesarlas, y enviarlas a la cola correspondiente. Se utilizan al final de las querys 3 y 4.

En este caso, un acumulator mantendrá un archivo de persistencia por cada grouper (para poder borrar los mismos cuando llegue un [[INICIO]] del grouper correspondiente.

Además, de forma similar a la persistencia, se tendrá un diccionario, que primero se indexa por grouper_id, y luego por civilizacion.

De esta forma, en el caso de que algún grouper comience a mandar las filas nuevamente (podría haberse caído antes de terminar de mandarlas) se reseteara la información del grouper en cuestión, no del resto.

El estado y los datos persistidos tienen el formato:

Memoria: {‘id_grouper0’: {‘Romanos’: [3,4], ‘Turcos’: [2,7]}, ‘id_grouper1’: {‘Romanos’: [2,2], ‘Mongoles’: [7,10]}}
Persistido: --un archivo para cada grouper-- [Romanos, [3,4], CHECK, ‘Turcos’, [2,7], CHECK, FIN, CHECK]

Cuando inicio reconstruyo mi estado:

- Para cada archivo parcial de id_group:
    - Itero sobre cada item y lo agrego si !=CHECK
    - Si alguno tiene un FIN, lo marco como terminado

También se necesita saber en qué estado está cada grouper, para resetear, o simplemente ignorar algunos mensajes de control (INICIO, FIN) que puedan venir duplicados desde los mismos. Para eso, mantenemos un diccionario {id_grouper_i: state_grouper_i}, que tiene el estado de cada Grouper (se puede reconstruir con los mismos mensajes persistidos en caso de caída del Grouper). Los tres estados en los que puede estar un Grouper (es análogo en las demás estructuras [más-de-un-producer (cola) consumer]) son [WAITING_FOR_INICIO, RECVING_ROWS, FIN_RECVED].

Tenemos para los mismos, dos etapas distintas que tratar:

En la primera etapa, recibimos las filas, las procesamos (se agregan a un diccionario, indexado por group_id, y luego por nombre de civilización). Al mismo tiempo, persistimos el player que acaba de llegar (para que el nodo sea resistente a caídas). La persistencia se realiza con el persistor, que maneja la consistencia de la misma, en caso de que el nodo termine su ejecución en medio de una escritura.

Si una eventual caída del sistema ocurre, se puede reconstruir nuevamente el diccionario desde la información persistida.

Se pueden resumir las acciones del accumulator en las siguientes:

1. Desencolo una fila (sin Auto ACK por eventuales caídas durante el proceso)
2. Proceso la fila (según el grouper_id del emisor, y la civilización)
3. Persisto la fila (con nuestro Persistor, en el archivo correspondiente al emisor)
4. ACK de la fila

En caso de caídas, como el último paso del protocolo es el ACK del mensaje, este se volverá a reprocesar, y la estructura en memoria permitirá eliminar players repetidos.

En la segunda etapa, se colapsa la información de los diferentes grupos, se calcula alguna métrica o ranking según el acumulador y se envía a la cola de salida. Luego se envían las filas, como en el resto del sistema, con un INICIO FIN, con el fin de evitar problemas ante caídas durante el envío de las mismas.

Se pueden resumir las acciones del accumulator en las siguientes

1. Leer el último FIN esperado de los joiners
2. Condensar, para cada civilización, los datos de los distintos grupos.
3. Enviar mensaje INICIO al cliente
4. Envío filas
5. Persisto TERMINE (por eventuales caídas luego de mandar una vez el FIN)
6. Enviar mensaje FIN al cliente
7. Vacío el archivo de persistencia
8. ACK del último mensaje

El protocolo para la recepción de INICIO-FINs es la siguiente:

Si recibe INICIO de id_grouper0:

- Resetear datos de ‘id_grouper0’
- Limpiar archivo de persistencia de id_grouper0
- ACK del mensaje

Si recibe fila resultado de id_grouper0:

- Agregar a la clave de ‘id_grouper0’ (que es un set)
- Append al archivo de persistencia correspondiente al id_grouper0 con la fila
- ACK del mensaje

Recibo FIN de id_grouper0:

- Se actualiza en memoria el estado de dicho grupo a FIN_RECVED
- Persisto un FIN en el archivo de id_grouper0
- Si ya me llegaron todos los FIN:

    1. acumulo toda la info de todas las civs en una sola variable
    2. Enviar un INICIO al cliente
    3. output resultados
    4. Enviar un FIN al cliente
    5. Reseteo los archivos de persistencia y variables
- ACK del mensaje

Masters y Elección de líder

Los nodos master pueden tener dos estados: Líder (solo uno entre los masters) y slave (el resto de los mismos).

El líder se ocupa de revisar la cola de heartbeats, en la cual los nodos del sistema (incluyendo los masters) envían heartbeats. El método pingsController se ocupa de controlar y relanzar los nodos si pasado cierto umbral de tiempo.
También persiste en un archivo simple de logs, los nodos relanzados.

A su vez, entre los masters hay una red de comunicación, en la cual el líder envía heartbeats a los slaves. En caso de que algún slave detecte como caído al líder, se celebrará una elección.

Cada nodo master tiene las siguientes tareas:
- Cuando me inicio, celebro una elección.
- Si soy líder, comienzo a monitorear a los nodos, y enviar heartbeats a los otros masters.
- Si es slave, monitorea al lider, en caso de detectar caída, iniciará una elección. A su vez, envía heartbeats al líder, para no ser detectado como caído y relanzado.

Masters y Elección de líder

Cada vez que un nodo se lanza, o cuando se detecta que el líder está caído, se celebra una elección. En la misma, se utiliza el algoritmo de Bully para elegir al nuevo coordinador.

El algoritmo usa tres tipos de mensajes:

- Mensaje de ELECTION: Enviado para anunciar una elección, sólo a los nodos con mayor id que el mío.
- Mensaje ALIVE: Respuesta al mensaje elección, proveniente de un nodo de mayor id. Si nadie me envía un mensaje ALIVE, y se cumple el tiempo, me declaro coordinador.
- Mensaje COORDINADOR: Enviado por el ganador al resto de los masters para anunciar su victoria. Todos los masters vuelven a sus tareas habituales, y el nuevo

coordinador comienza a monitorear el sistema (eventualmente relanzará el nodo master caído previamente)

Cuando un máster P se recupera de una fallo detecta que el líder ha caído, ejecuta las siguientes acciones:

- Si el nodo P tiene el ID más alto, envía un mensaje COORDINATOR al resto de los masters, declarándose ganador. En caso contrario, envía en mensaje ELECTION a los nodos con ID más alto que P (posibles líderes).
- Si P no recibe respuesta luego de enviar mensajes ELECTION, se declara victorioso y envía el mensaje COORDINATOR a todos los nodos.
- Si P recibe respuesta de un nodo con ID mayor, entonces no manda más mensajes y espera a un mensaje COORDINATOR (algún nodo ganando la elección). Si no hay victoria después de un periodo de tiempo, comienza un nuevo proceso de elección (puede pasar que el victorioso falle antes de enviar el mensaje COORDINATOR)
- Si P recibe un mensaje ELECTION de un nodo con ID menor, entonces envía un mensaje ALIVE a dicho nodo, y propaga el mensaje ELECTION a los nodos con ID superior a P.
- Si P recibe un mensaje COORDINATOR, comienza a tratar al emisor del mismo como el nodo Líder.

Trabajo a Futuro

Si bien el sistema presentado resuelve los casos de uso propuestos, quedan algunas mejoras para hacer a futuro:

- Paralelizar controllers intermedios, como filtros y exchanges: representan un gran cuello de botella para el sistema, cuando podrían estar paralelizados. Actualmente no está implementado pues provocaría incongruencias con el manejo de los Sentinels.
- Hacer uso de un agente en los nodos para el manejo de los Sentinels: actualmente, muchos nodos deben conocer cuantos otros tiene delante y detrás suyo en el pipeline, generando un fuerte acoplamiento entre ellos. Sumando un agente extra a cada nodo, con la única tarea de propagar los Sentinels, mejoraría esto.
- Dropear columnas extras: algunas columnas del dataset no se usan para ninguna query por lo que podrían tranquilamente obviarse, reduciendo el uso de red y memoria en todo el sistema.
- Reemplazar RabbitMQ por sockets para monitoreo: Como se mencionó antes, no es deseable usar un sistema como Rabbit para tareas sensibles como monitoreo. A futuro, deberían utilizarse sockets TCP/IP.
- Colas anónimas: Reemplazar las colas anónimas dentro del sistema, esto permite al nodo que es relanzado poder conectarse a la misma cola que antes, y que no se pierdan los mensajes del exchange.
- Posible race condition al declarar Master como COORDINATOR: Actualmente, si un nodo se declara como Coordinator inmediatamente comienza a monitorear a los nodos. Si ya hay otro líder en línea, puede ocurrir por un breve momento que ambos monitoreen y levanten nodos, corrompiendo estados. Para solucionar esto, debería implementarse un ACK al mensaje coordinator, de forma que el líder ceda el control y *luego* asuma el nuevo Master.