# U.B.A. FACULTAD DE INGENIERÍA

## Departamento de Informática

## Sistemas Distribuidos I 75.74

## TRABAJO PRÁCTICO N° 3

## Tolerancia a fallos

**Curso 2020 - 2do Cuatrimestre**

| APELLIDO, Nombres | Padrón |
|---|---|
| Avigliano, Patricio Andres | 98861 |
| Cuneo, Paulo César | 84840 |
| Fecha de Aprobación : | |
| Calificación : | |

**Observaciones:**

# Funcionamiento general

El sistema permite obtener información analitica sobre las reviews de yelp, a partir de analizar un stream finito de datos. A la vista de los clientes, el sistema atiende requests a los cuales responde si está o no disponible para procesar. En caso de estar disponible consumirá un stream con la información de los negocios registrados en Yelp y otro stream con las reseñas de los usuarios de Yelp propiamente, ambos generados por el cliente. Una vez procesados los datos el sistema almacenará el resultado para que el cliente pueda consultarlo y se establecerá como disponible para atender nuevos requests.

Conceptualmente el procesamiento tiene la siguiente etapas:

**Clientes:** Son las distintas entidades que intentan conectar con el sistema y en caso de recibir una respuesta positiva generan streams de datos, enviados a través de mensajes, con sus respectivas señales de fin de stream.

**Ruteo:** El sistema toma los mensajes de los streams en la etapa de ruteo donde se filtra la información necesaria para realizar cada consulta. Para simplificar primero se envían/procesan los negocios y luego las reseñas ya que es necesario que se hayan procesado primero todos los negocios para poder procesar las reseñas en una de las consultas.

**Mapeo:** En esta etapa se realiza una función de mapeo sobre cada una de las reseñas, puede esperarse a recibir la información agregada de los negocios para empezar a procesar en caso de ser necesaria.

**Agregación:** A continuación viene la etapa donde los datos se agregan/reducen según el criterio correspondiente hasta recibir la señal de fin del stream.

**Junta:** Una vez con los datos agregados, se realiza las operaciones de juntas necesarias sobre los mismos combinando 2 agregaciones distintas.

**Ordenamiento:** Finalmente en esta etapa se ordenan los datos ya agregados/juntados según el criterio que corresponda.

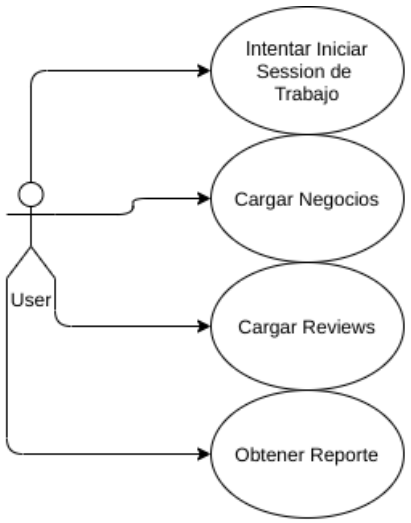
**Mapeo:** Puede ser necesario realizar nuevamente una operación de mapeo esta vez sobre los datos ya agregados/juntados.

# Vistas de Arquitectura 4+1

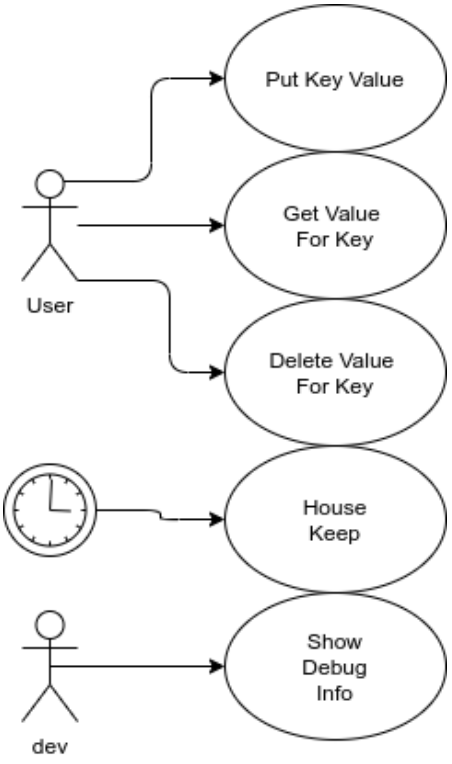
## Escenarios

El cliente del sistema deberá recorrer 3 fases para su uso:

- Que el sistema acepte su solicitud (si no está disponible la rechazará)
- Carga de datos
- Consulta del resultado de la solicitud



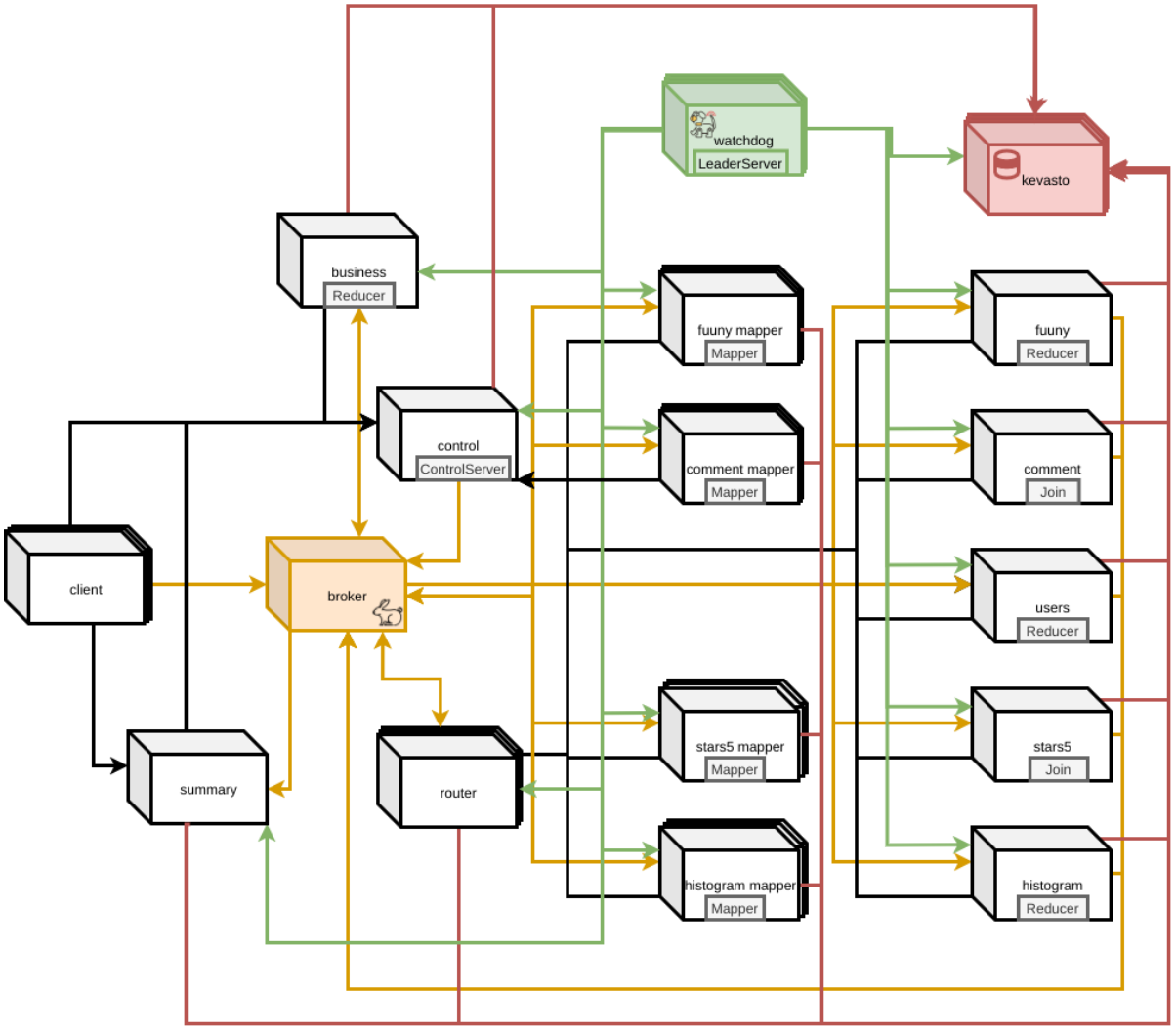

Como veremos más adelante los requerimientos técnicos nos llevarán a implementar un almacenamiento estable. El cual presenta los siguientes casos de uso.



# Vista física

Mostramos los nodos “físicos” en juego, comunicándose entre sí de las siguientes formas:

- Mediante colas con el broker de intermediario (líneas naranja).
- Mediante requests http (líneas rojas, verdes y negras).




Podemos ver que el proceso de junta se realiza en el mismo nodo que en donde se realiza una de las 2 agregaciones previas a la junta.

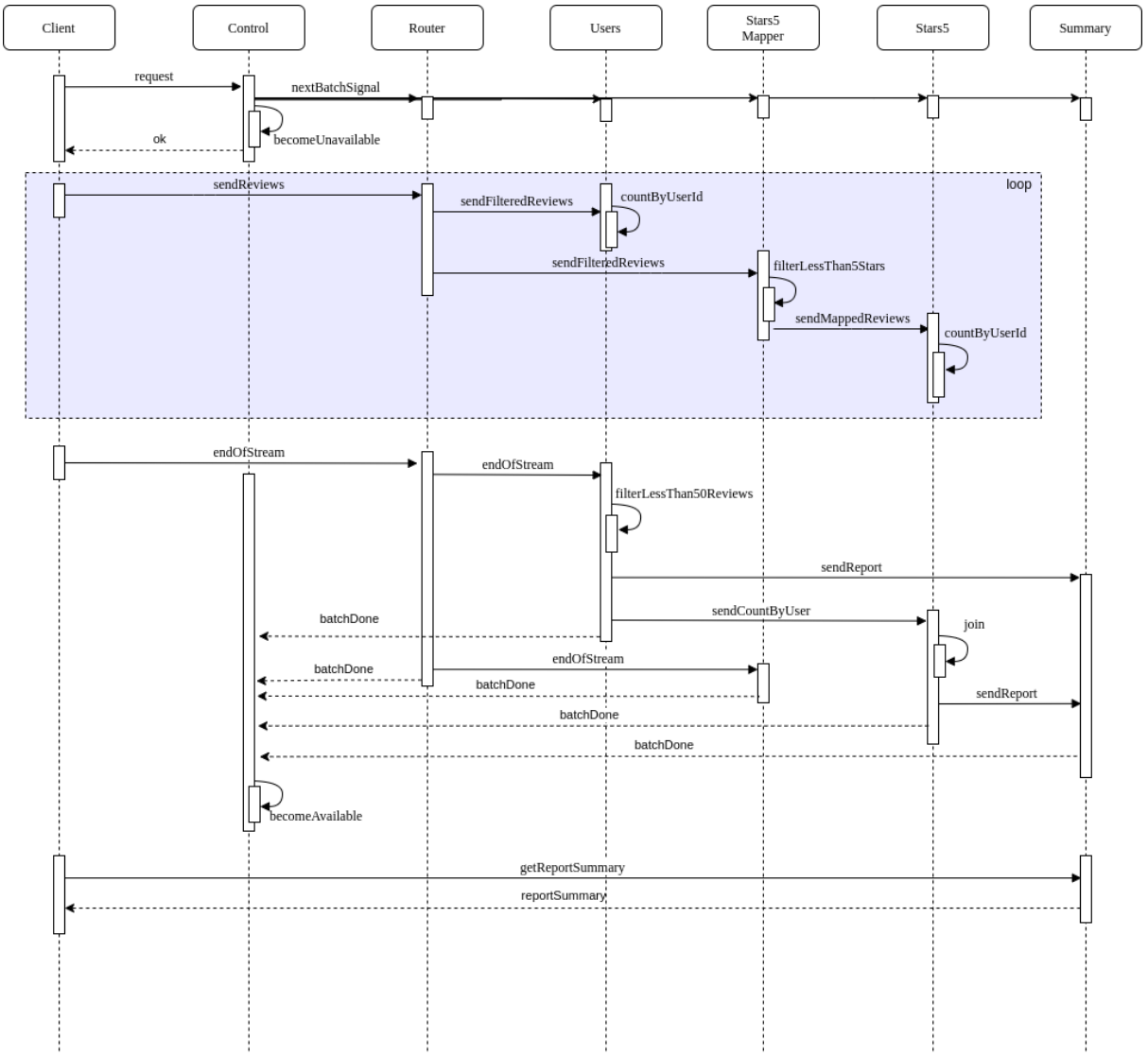
Con el fin de coordinarse, todos los nodos de las etapas de procesamiento se comunican con el nodo de control mediante http (líneas negras) para indicar que han finalizado el procesamiento de una solicitud.

Kevasto es responsable de la persistencia del sistema. Los nodos tienen estado y se comunican con kevasto (líneas rojas) mediante http para persistirlo y recuperarlo.

A su vez todos los nodos propios del sistema están monitorizados por el watchdog mediante healthchecks http (líneas verdes).

## Vista de Procesos

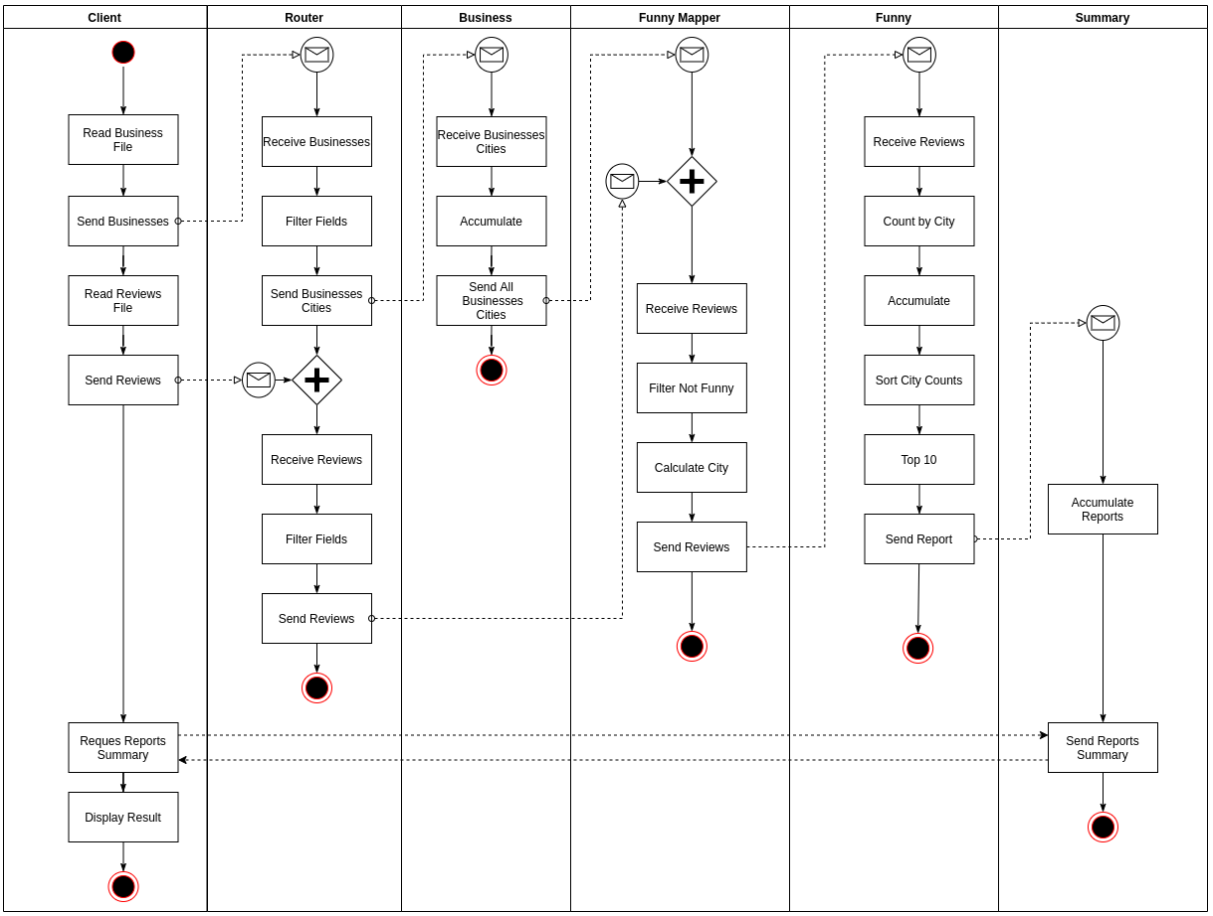
En el siguiente diagrama intentamos mostrar el flujo de control de procesamiento de un request de un cliente, también mostramos en particular la secuencia del procesamiento de la consulta de usuarios con más de 50 reseñas todas con máxima puntuación.




Ante un request de un cliente, suponiendo que el sistema se encuentra disponible, la entidad de control envía una señal de inicio de procesamiento de batch a cada etapa, esto será de utilidad para sincronizar las etapas entre cada batch.
En un loop se envían todas las reseñas a los nodos Users y Star5 los cuales las cuentan por usuario, las que llegan a Star5 son filtradas en una etapa intermedia. Users filtra los

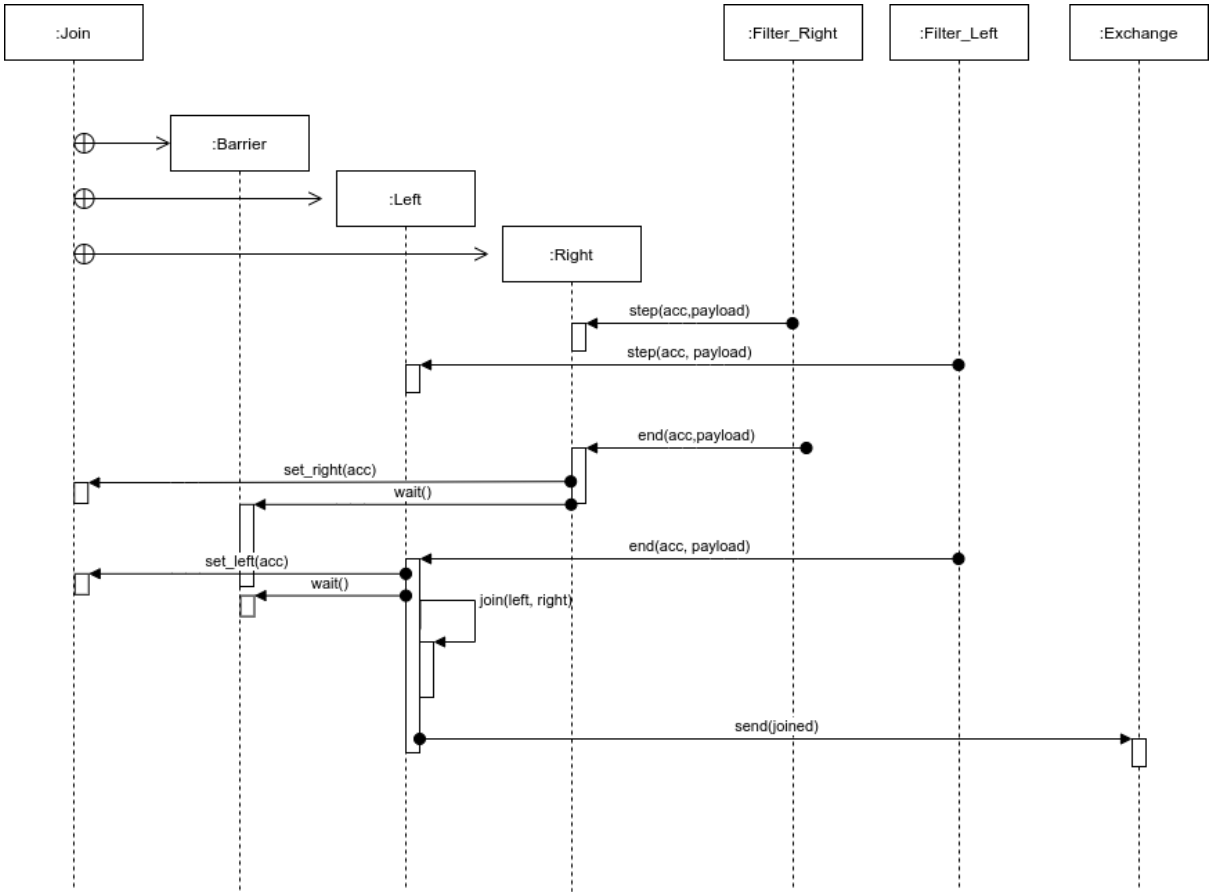
usuarios con más de 50 reseñas y se envían en un único mensaje a Star5 quien se encarga de hacer la junta por usuario y enviar el reporte a Summary. Esta última etapa una vez reunidos todos los los reportes, finalmente será capaz de responder al cliente con el resultado.

En este diagrama mostramos el flujo de la consulta de las 10 ciudades con reseñas de negocios divertidos para una única ronda de procesamiento. Es decir, excluyendo la lógica del flujo de control entre batches de procesamiento y manejo de solicitudes del cliente



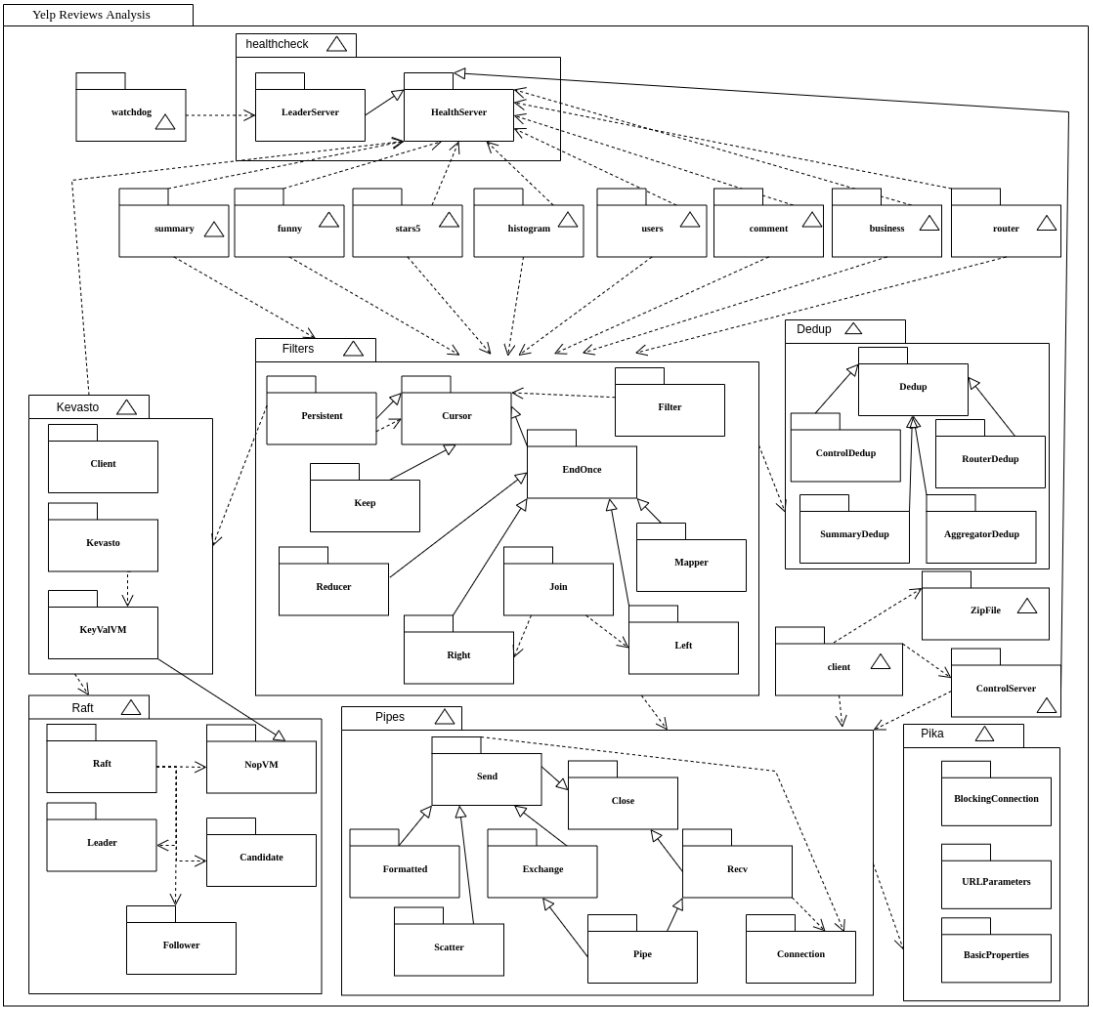

Se envían primero los negocios que son acumulados por el nodo Business y luego las reseñas que recién empiezan a ser procesadas cuando Funny Mapper tiene la información sobre la ciudad de cada negocio. Se filtran las reseñas que no fueron ingresadas como una experiencia divertida y las que si se les busca a qué ciudad corresponde. El nodo funny cuenta las reseñas por ciudad y al recibir la señal de fin, ordena y responde al cliente con las primeras 10.

En el caso de hace un Join, se utiliza un cursor Left y Right para unir los datos de ambos en un solo acumulador, donde el fin de ambos está coordinado por pertenecer a un mismo Join. En siguiente diagrama se puede ver la lógica resumida de coordinación:



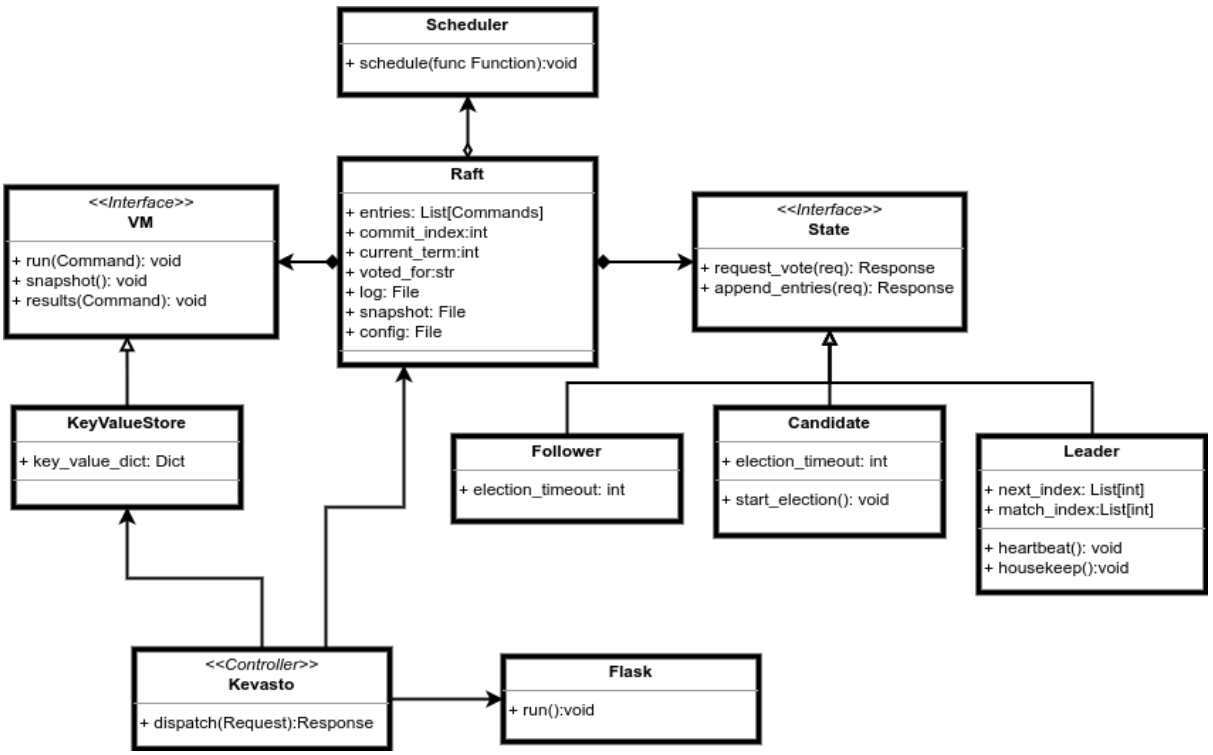
# Vista de Implementación

En el diagrama de paquetes tenemos la capa de pipes, que abstrae la lógica de comunicación mediante el broker de mensajes. Por otro lado tenemos la capa de filters donde se encapsula la lógica de procesamiento de datos.
Adicionalmente tenemos los paquetes correspondientes a la implementación de Raft, a la Kevasto(Key Value Store),  los watchdog.

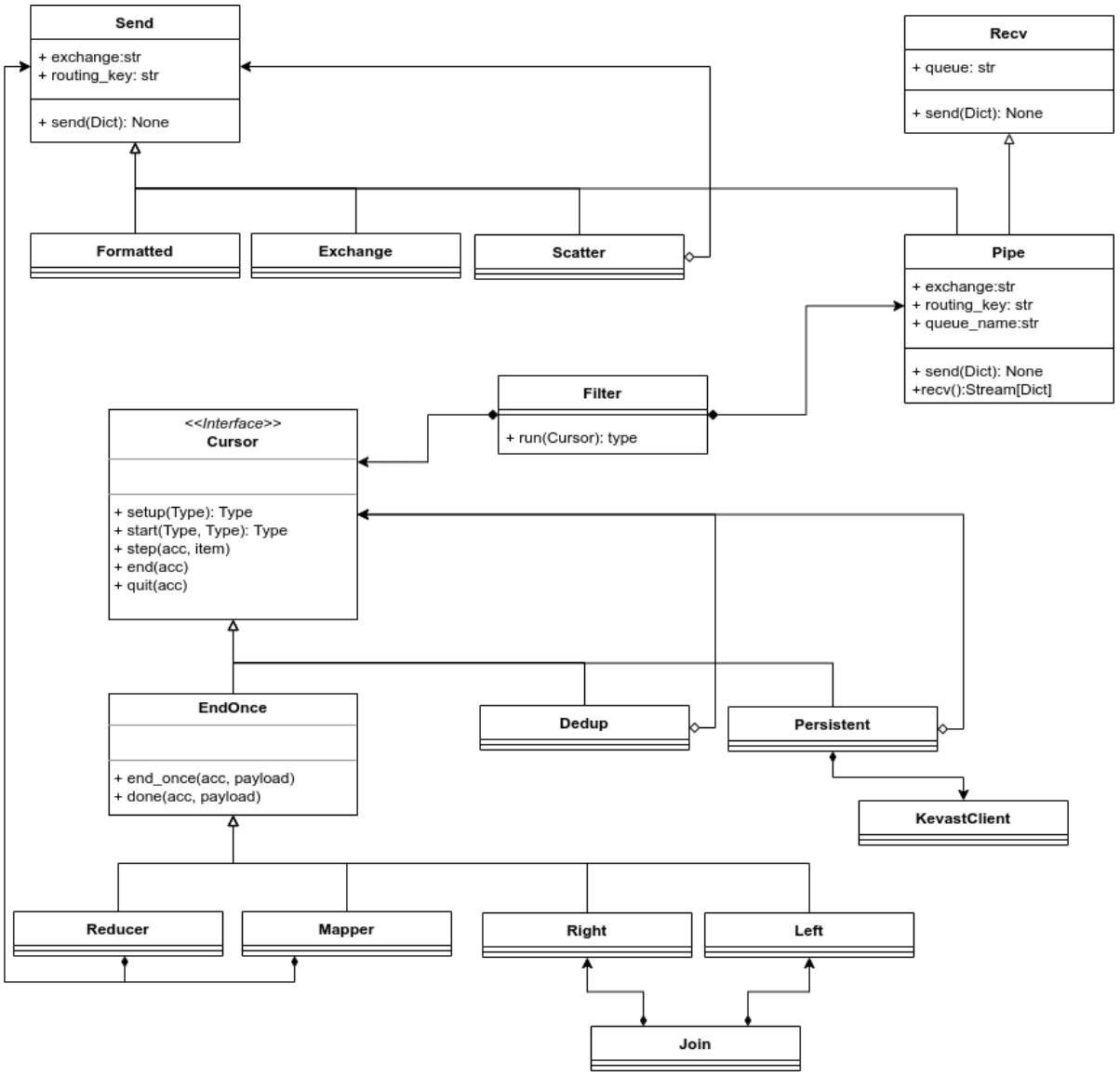


# Vista Lógica

En particular entre el algoritmo de Raft y Kevasto tenemos la siguiente relación entre clases



Para la parte de de aggregators y mappers tenemos el siguiente diagrama:




Cada proceso utilizará un Filter que lee un Pipe alimentando un Cursor, los cursores reciben cada dato del pipe y aplican un paso sobre una acumulacion para el caso de los Reducer, o simplemente mapean un dato para los Mapper. Tanto Reducer como Mapper envían su resultado a la siguiente etapa pasándolo a un Send, el Send puede ser un Exchange decorado por un Formatter, para formatear el dato al momento de enviarlo. También es posible enviar el dato a más de una Send, utilizando el composite Scatter.

Para hacer persistentes los datos acumulados por un cursor, se puede decorar con Persistent que guarda el valor acumulado y los elementos procesados en Kevasto.
Para asegurar la consistencia de los datos y no procesar dos veces el mismo elemento producto de fallas en etapas previas, se puede decorar con un Dedup.
Los cursores que heredan de EndOnce, asegurar el envío de un solo fin a las etapas

sucesivas, utilizando un contador sobre el mensaje de eof y re-enviandolo por el pipe de entrada.

# Tolerancia a fallos

Para poder asegurar que el resultado del procesamiento no va a estar afectado por la potencial caída de nodos de nuestro sistema debemos considerar los siguientes puntos:

- Monitorización
- Consenso entre réplicas
- Manejo de duplicados
- Recuperación del estado ante una caída
- Persistencia

## Monitorización

Si un nodo del sistema se cae necesitamos ser capaces de detectarlo y una vez hecho volver a levantarlo. Para ello implementamos un proceso “watchdog” el cual se encarga de realizar periódicamente un request http del tipo “healthcheck” a cada nodo del sistema. En caso de recibir una respuesta KO o no recibir respuesta se encargará de disparar un restart del nodo en cuestión. Dentro de cada nodo del sistema corre un hilo encargado de atender los “healthcheck” requests.
Una limitación de la implementación actualmente es que el healthcheck es independiente del procesamiento, es decir este sistema de monitorización solo detecta si el nodo está caído pero no es capaz de garantizar que se encuentra verdaderamente operativo.
Como próximo paso para mejorar sistema de monitorización planteamos agregar al request de healthcheck una respuesta del estilo:

response = { isProcessing, lastProcessedTimestamp }

De modo tal que podamos verificar que si en cierto punto el nodo debería encontrarse activo, verdaderamente está procesando

## Consenso entre réplicas:

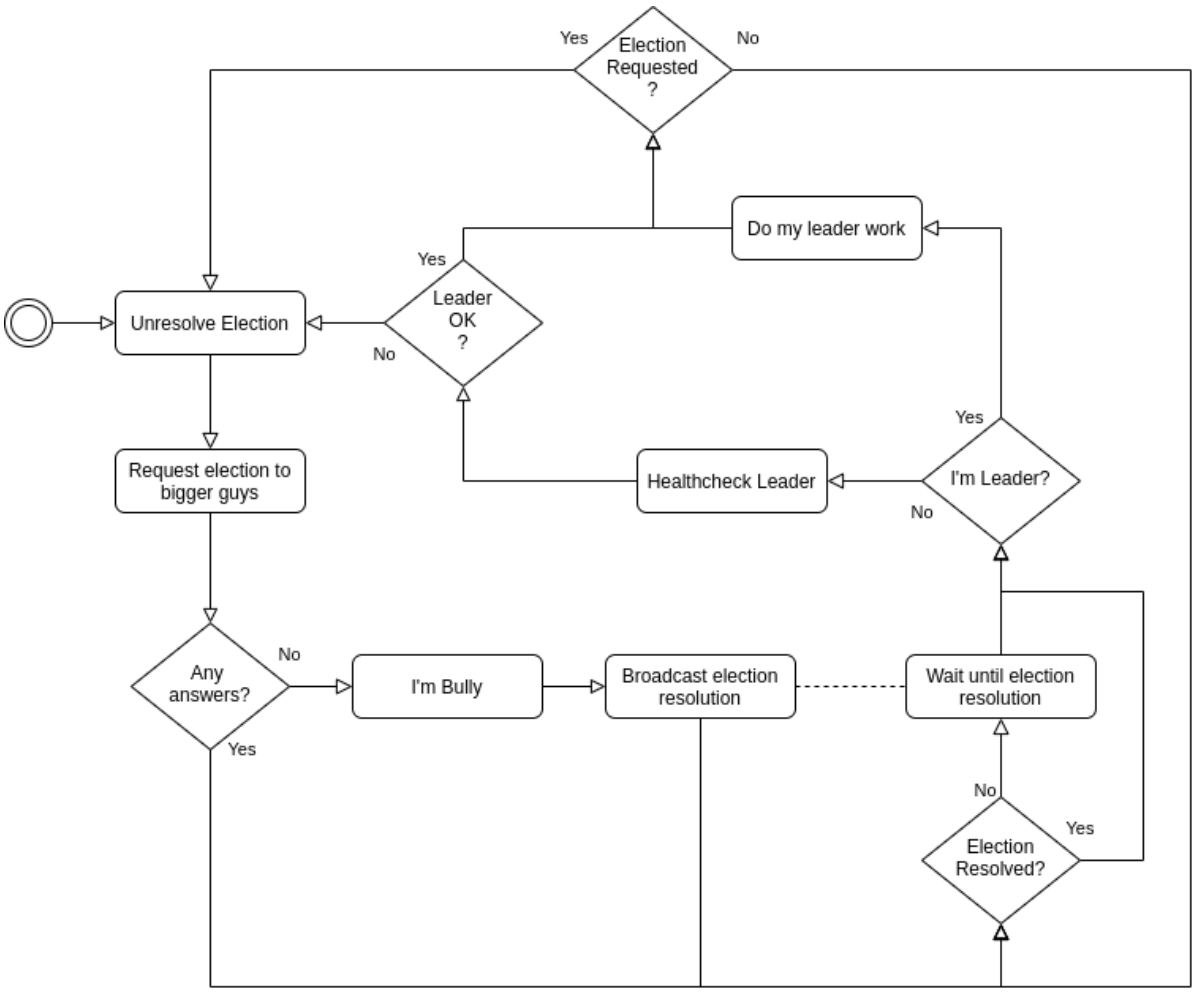
Considerando la posibilidad de que los procesos encargados de monitorizar también pueden fallar, hemos de replicarlos, sin embargo solo deseamos que una de las réplicas

ejecute el restart de un nodo caído. Para ello diferenciamos a una de las réplicas (leader) del resto (workers) consensuando mediante una elección quién es leader. Para resolver el consenso planteamos el algoritmo de bully, en el cual el nodo con identificador mayor será elegido leader.

Las réplicas atienden en un thread solicitudes http que sirven para disparar 2 eventos entre réplicas:

- La solicitud de una elección
- La resolución de una elección

A continuación presentamos un diagrama que muestra el flujo de trabajo de las réplicas:




Siendo una réplica mi punto de entrada será iniciar una elección, que consiste en solicitar una elección para toda réplica con identificador mayor al mío. En caso de no recibir ninguna respuest significa que soy la réplica activa con identificador mayor por lo tanto me convertiré en leader y resolveré la elección de todas las réplicas anunciándome como tal. En caso contrario si alguien responde mi solicitud de elección significa que no seré leader y entonces me quedaré esperando que se resuelva mi elección si es que aún no lo está.
Una vez resuelta la elección tomaré 2 caminos distintos en función de si soy o no leader. Si soy leader haré cual fuera mi trabajo de leader (en este caso monitorizar a todos los nodos

del sistema), si soy worker me encargaré de monitorizar al leader y en caso de no recibir respuesta iniciaré una elección.
A partir de aquí repetiré indefinidamente mi ciclo de trabajo verificando siempre tras finalizarlo si se me ha solicitado iniciar una elección.

Ante la falla de un worker el leader se encargará de reiniciarlo.
Ante la falla del leader:

- Una de las réplicas ganará la elección convirtiéndose en el nuevo leader
- El nuevo leader reiniciará al leader original
- El leader original iniciará una elección y la ganará convirtiéndose nuevamente en leader

De esta manera garantizamos que mientras haya una réplica de monitorización activa, el sistema será capaz de levantarse.

## Manejo de duplicados:

Para mantener la consistencia de los datos durante el procesamiento ante fallas debemos ser capaces de identificar y descartar los mensajes duplicados que estas fallas potencialmente van a generar. Para ello las etapas hacen uso de una capa que se encarga de persistir el conjunto de identificadores de ítems (mensajes, batches, etc..) ya procesados. Esta capa permite:

- Consultar si un ítem fue procesado
- Marcar un ítem como procesado
- Resetear el conjunto de ítems procesados

A continuación discutiremos las particularidades de cada etapa sobre las estrategias planteadas para garantizar la disponibilidad del sistema

## Etapa de Control

Para empezar el procesamiento de una solicitud todas las etapas reciben por una cola de control un mensaje con identificador del “batch” a procesar, este identificador será clave para descartar mensajes duplicados, incluyendo las señales de fin de stream.

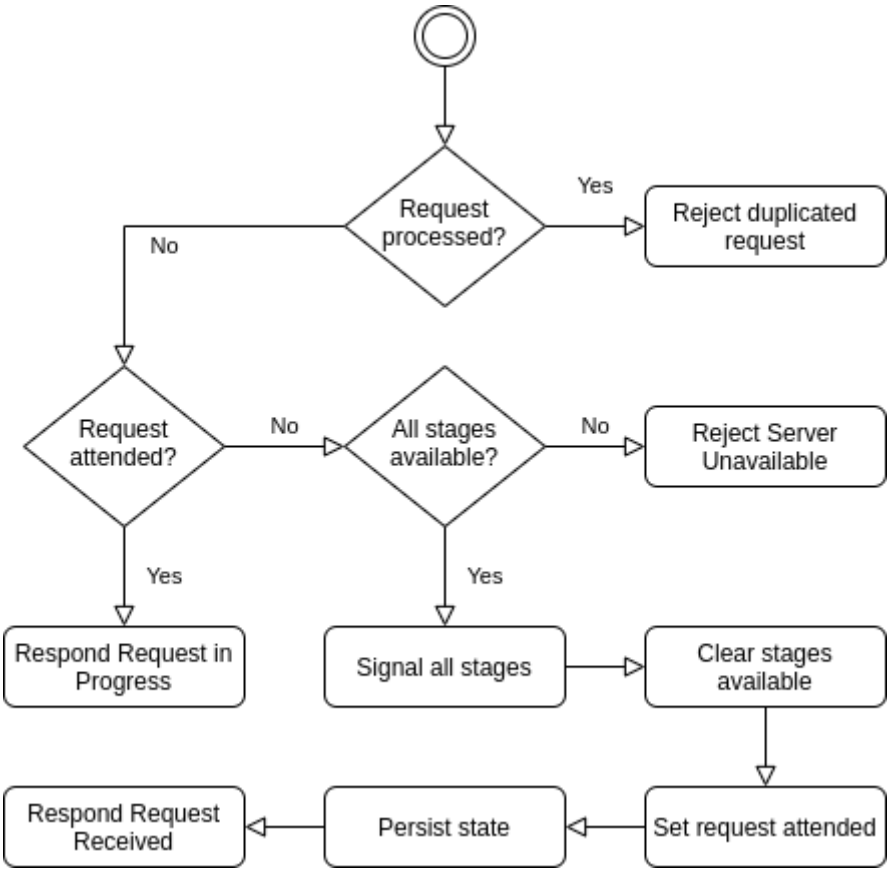
La etapa de control, al iniciar, recupera su estado que está compuesto por:

- Conjunto de solicitudes atendidas
- Conjunto de solicitudes (o batches) completadas
- Conjunto de procesos que han terminado de procesar la solicitud actual

Luego permanecerá a la espera de atender solicitudes de clientes o de recibir señales de fin de procesamiento de todos los nodos que realizan procesamiento en el sistema.

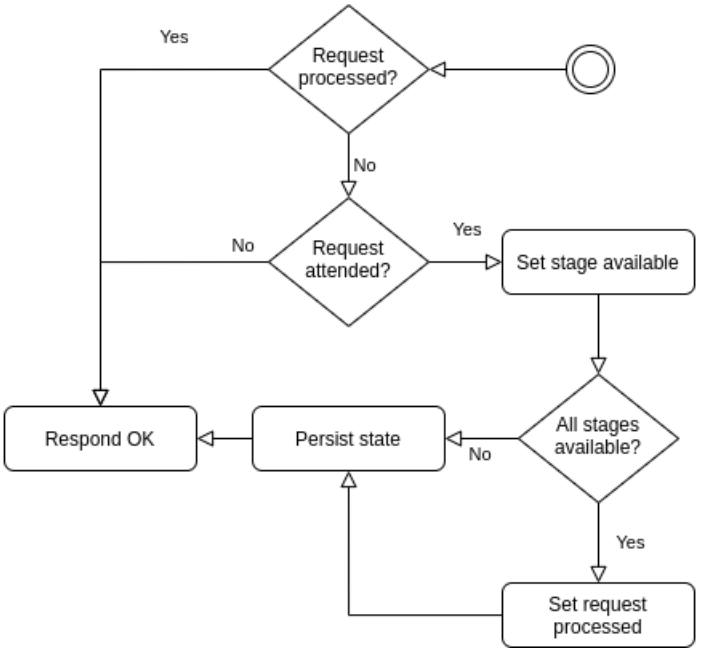
Las solicitudes de los clientes tienen un identificador que deberá ser único. El cliente tiene la responsabilidad de reintentar si el servidor no está disponible.

Al recibir una solicitud de un cliente la etapa de control transita el siguiente flujo:




Ante una caída del servidor de control después de la persistencia del estado, el cliente reintentará y se le responderá que su solicitud se está procesando.
Ante una caída antes de la persistencia del estado se enviará duplicada la señal de inicio de batch a todas las etapas, quienes serán responsables de descartarla por su identificador.

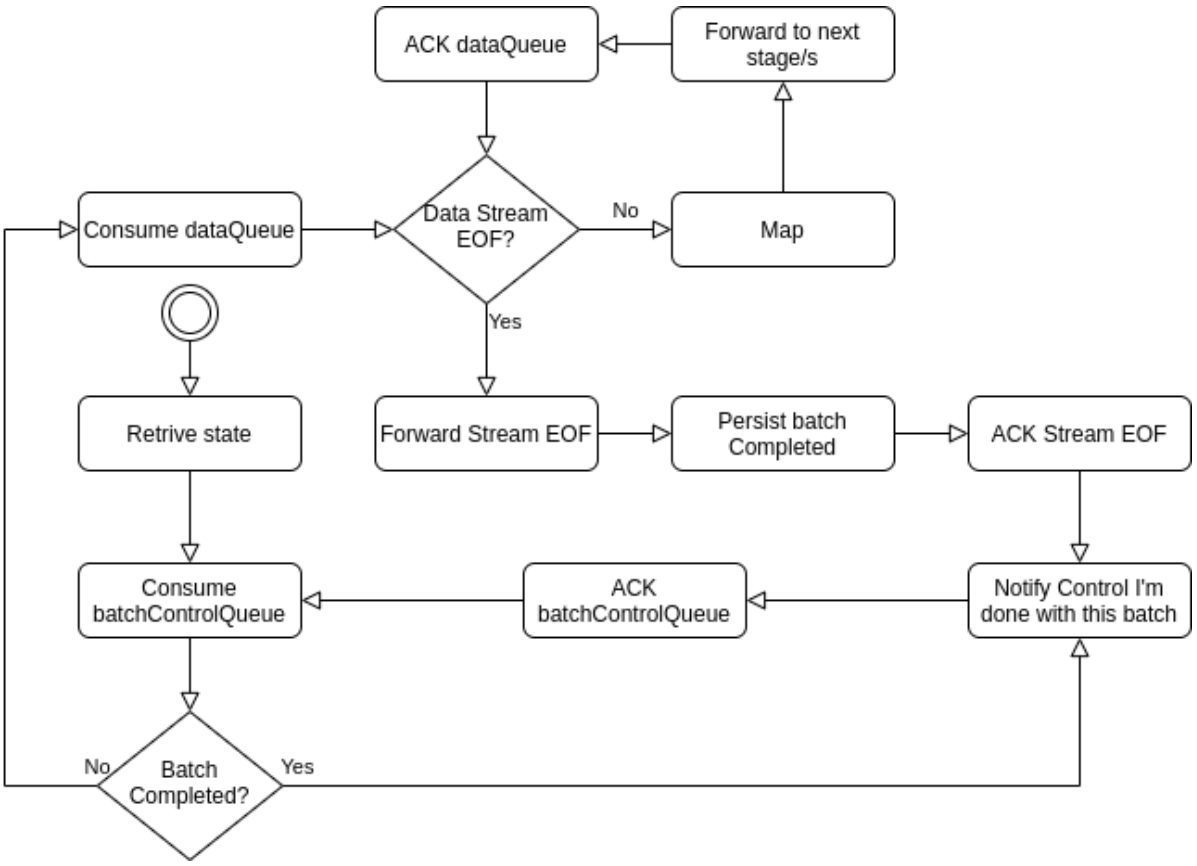
Al recibir una señal de fin del procesamiento de un batch/request de un nodo determinado el servidor de control transita el siguiente flujo:




Ante una caída en cualquier punto del flujo el nodo que envía la señal de fin tiene la responsabilidad de reintentar.

## Etapas de Mapeo

En el siguiente diagrama mostramos el flujo el cual atraviesan las etapas de mapeo:




Las etapas de mapeo preservan un estado compuesto por los identificadores de los batch ya procesados. Ante un nuevo batch a procesar la etapa comprueba que no es un duplicado y procede con el procesamiento de los datos.
Frente a una caída durante el procesamiento no se manejan los duplicados, asumimos un identificador único de mensaje por lo que se vuelve a mapear el mensaje delegando la responsabilidad de descartarlo a una etapa siguiente.
Una vez finalizado el procesamiento se propaga la señal de fin del stream y luego se persiste el identificador del batch como completado. Ante una caída en este punto se generará un duplicado de la señal de fin de stream, que será descartada por su identificador de batch.
Luego se procede a notificar al nodo de control el fin del procesamiento del batch tanto por http como con el ack a la cola.

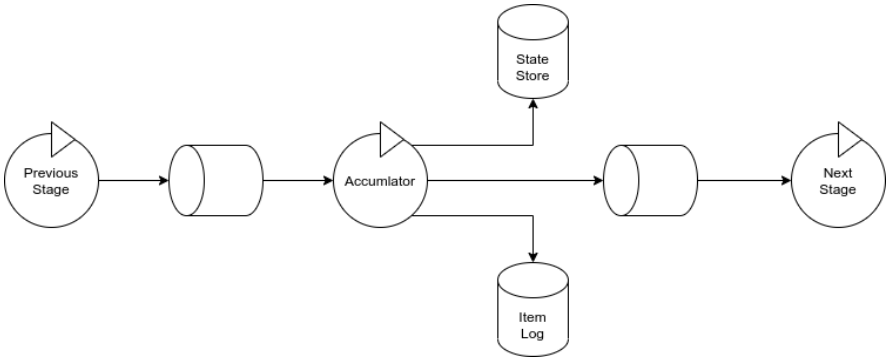
## Etapa de Ruteo

La etapa de ruteo tiene un flujo muy similar, con la salvedad de que procesa 2 streams de datos de manera secuencial: los negocios y las reseñas. Ante una falla procesando el segundo stream, al recuperarse deberá reconocer como completado el primer stream de lo contrario el sistema quedaría bloqueado tratando de consumir negocios que nunca llegarán. Para solucionar este escenario decidimos persistir los identificadores de batch completados por stream en la etapa de ruteo

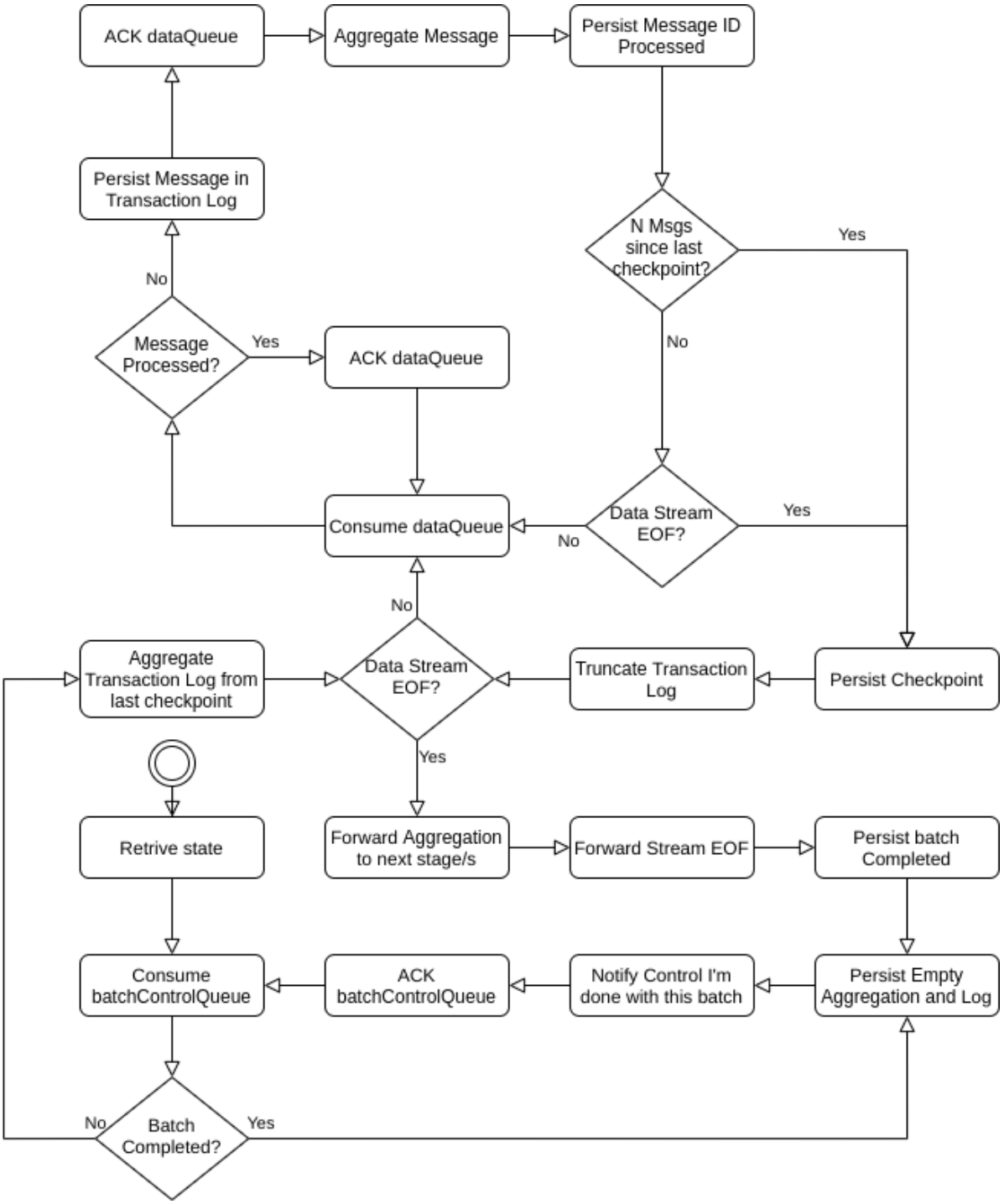
# Etapas de Agregación

Luego de que suceda una falla el sistema debe poder recuperarse y continuar procesando los datos, como si la falla no hubiera ocurrido.

Para ello la estrategia utilizada es persistir los elementos ya procesados en un transaction log y cada cierta cantidad de elementos tomar un snapshot del estado acumulado descartando los datos del transaction log previos al snapshot.
La idea es que frente a una falla tengamos todos los datos entre el checkpoint previo y la falla. De forma tal de reconstruir el estado al momento de la falla.
Haciendo foco en esta parte del sistema podemos observar la interacción entre el proceso acumulador y su contexto de operación:




El algoritmo de procesamiento transita el siguiente flujo:




El flujo de control en esta etapa es similar al de las que analizamos anteriormente, por ello pongamos el foco en la parte novedosa: la utilización de snapshot y transaction log.
Al recuperarse el nodo tras una falla lo que hará es recuperar el estado antes de la falla a partir del estado del checkpoint/snapshot y los elementos que haya en el Transaction log. Hasta no detectar la señal de fin de stream consumirá mensajes de la cola, filtrando duplicados, y los irá guardando en el Transaction log, agregando al estado y marcando como procesados. Cada cierta cantidad N de mensajes acumulados en el Transaction log actualizará el checkpoint y recortará los elementos antiguos del log.

Luego envía el resultado y fin de stream a la siguiente etapa y persiste el batch como completado.
En el caso que la falla se produzca al finalizar el stream de procesamiento, no es un problema ya que se puede utilizar el identificador de batch en la siguiente etapa para descartar el duplicado.
Finalmente se resetea el estado, se notifica a la etapa de control el fin del procesamiento (reintentando si no responde) y por último se da ack al batch de la cola de control. Ante una caída después de persistir el batch como completado se identificará el batch desencolado como duplicado y se repetirán los últimos 3 pasos para asegurar que se ejecuten.

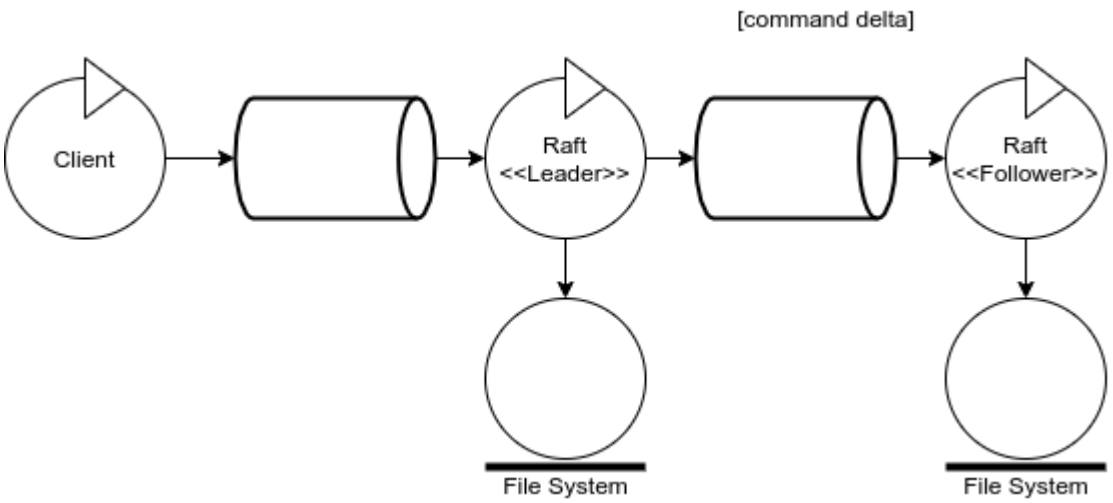
# Persistencia

Para que el recupero pueda funcionar correctamente requerimos un almacenamiento estable. El mismo está implementado basado en el algoritmo  Raft, que permite replicar comandos sobre máquinas virtuales distribuidas. En nuestro caso particular la máquina virtual es simplemente un diccionario en memoria y los comandos son las operaciones de agregar, sacar y recuperar datos.

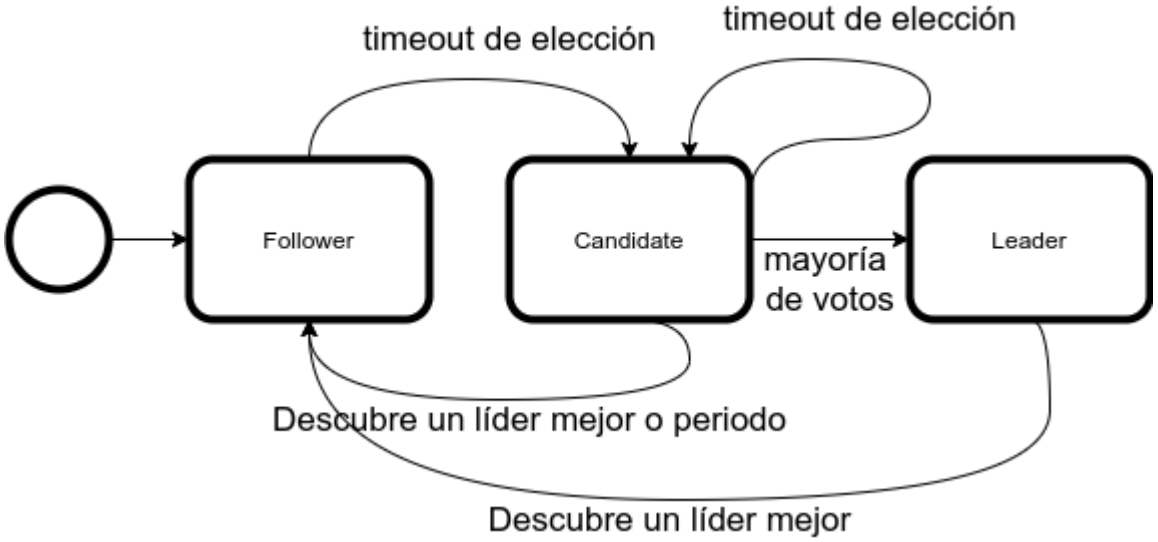
El algoritmo Raft tiene las siguiente características:

- Orientado a Leader y elecciones.
- Para un cluster de 2n+1 instancias tolera n fallas.
- Bien estudiado y utilizado en la industria.

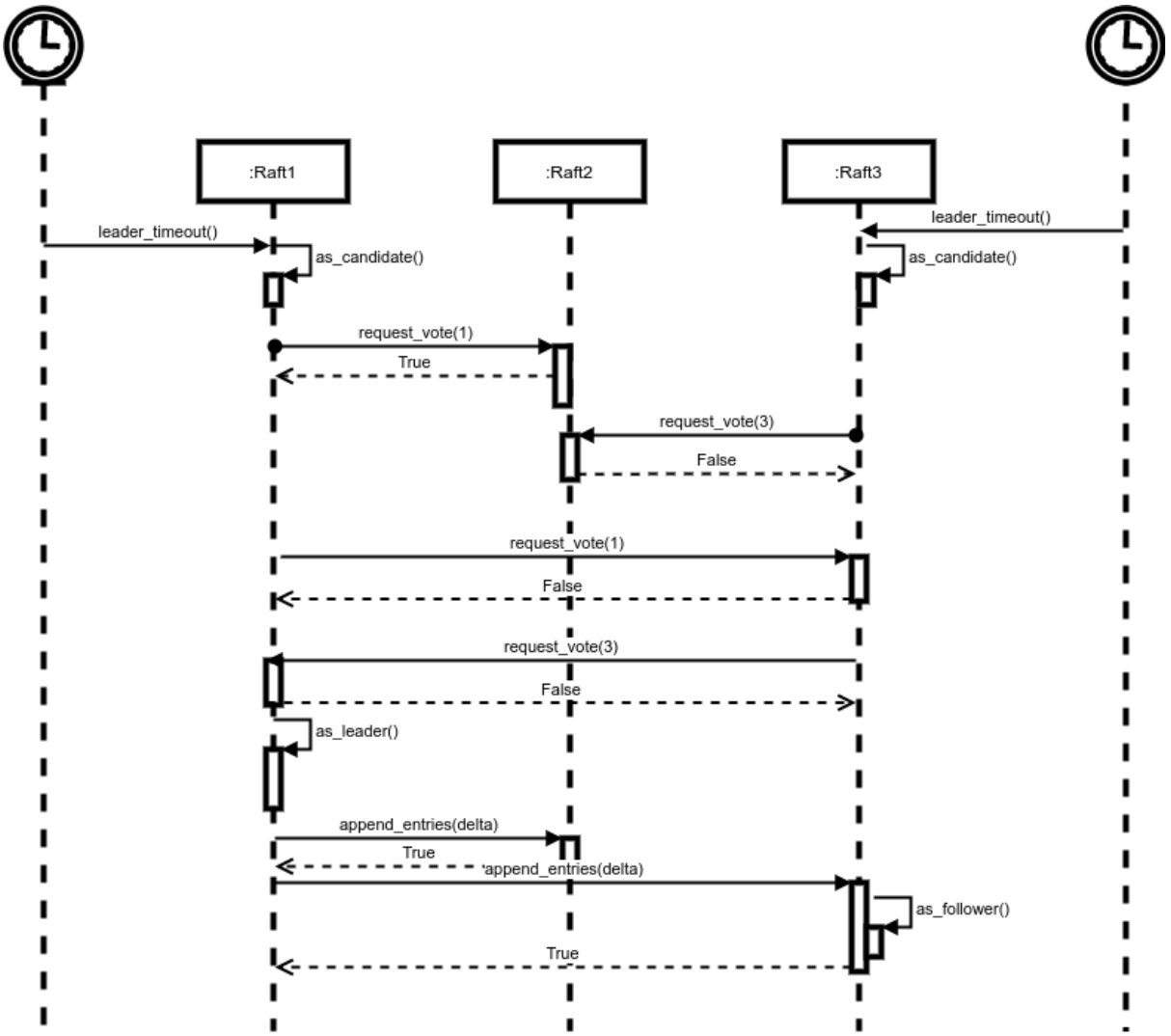
Los clientes siempre interactúan con el líder del cluster, es el encargado también de mantener las réplicas al día. El líder dará por commiteado/replicados correctamente los elementos que puede confirmar están en la mayoría de las instancias. Y dichos elementos son los que pueden aplicarse a la máquina virtual. Es decir, solo los comandos que estén replicados mayoritariamente, será aplicado a la máquina virtual.
Esto quiere decir, que el cliente debe reintentar los comandos hasta tener un confirmacion del leader, y por otro lado es necesario que los comando sean idempotentes, afortunadamente, los comandos de la keyvalue store los son.




Frente a un fallo del líder la primera réplica en detectar la ausencia del líder, transiciona a un estado candidato, en el cual inicia un proceso de elección de líder.
Frente a una competición entre candidatos, ganará el candidato que tenga mejor historial de comandos, para Raft esto significa que el tenga el log con más elementos y el periodo mayor.




El siguiente diagrama presenta un escenario de elección de líder, donde 2 instancia compiten el voto de la tercera. Si el primero en llegar gana los votos, la competencia se resuelve por mayoría.


:Raft1
:Raft2
:Raft3
leader_timeout()
as_candidate()
leader_timeout()
as_candidate()
request_vote(1)
True
request_vote(3)
False
request_vote(1)
False
request_vote(3)
False
as_leader()
append_entries(delta)
True
append_entries(delta)
as_follower()
True

# Status

La entrega actual contiene una implementación de los watchdogs completamente integrada y funcional.

La key value store también está completamente implementada y funcional, siguiendo lo mejor posible el pseudo código del paper de Raft, y pruebas preliminares indicarían que funciona correctamente. Pudimos comprobar que efectivamente el líder logra replicar su historial de eventos y que realiza snapshots cuando alcanza el máximo de archivo tolerado para el log de comandos.

Known Issues:

- Si bien en el análisis y el concepto del sistema existe un componente de summary, para coordinar el agregado de las partes del reporte. Por una cuestión de tiempos y simplicidad esta etapa fue implementada como parte del cliente, en lugar de un controller independiente, obviamente esta solución es un work in progress ya que la responsabilidad de recopilar los reportes y descartar los duplicados que pueda haber de cada reporte no corresponde al cliente.
- Tenemos un caso borde en el cual no se detecta el batch procesado para el caso en el que un nodo recibe data ya agregada para enriquecer en su procesamiento. En este caso no podemos verificarlo desde la capa superior por eso hace falta adaptar el modelo para verificarlo correctamente de forma que el sistema sea robusto ante el escenario en que se caen ambas etapas al mismo tiempo.
    - Al bajar los procesos de funny mapper y de business, se produce una inconsistencia entre el recupero del estado y el marcado de batch como procesado, esto provoca que funny mapper intenta mapear y enviar ítems cuando el batch ya finalizó.
    - Otro caso similar sucede en el join, no tan fácil de reproducir, sucede cuando se baja de forma reiterada kevasto, donde también se produce una descoordinación entre la reconstrucción del estado luego del eof y el marcado de batch procesado. La solución en este caso sería manejar una única marca de batch completado para ambas ramas del join y testear la reconstrucción del estado para asegurar que sea consistente con la marca de batch completado.
- En la implementación de raft que hicimos :
    - Cuando el timeout de elección es chico y el líder está realizando un snapshot, un candidato con el log desactualizado puede ganar la elección teniendo un log desactualizado, y llevando a la pérdida de información. Para mejorar esto es necesario escribir test de los escenarios posibles de elección de leader.
    - La generación del snapshot en el líder bloquea el envío de heartbeat, además sería bueno adaptar la implementación para que siga al pie de la letra las recomendaciones del paper.