Reddit Memes Analyzer HA

Manuel del Carril, *Padrón 100772*
mdelcarril@fi.uba.ar

Sebastian Ripari, *Padrón 96453*
sebastiandanielripari@hotmail.com

Matías Lafroce, *Padrón 91378*
mlafroce@fi.uba.ar

1er. Cuatrimestre de 2022
75.74 Sistemas Distribuidos I
Facultad de Ingeniería, Universidad de Buenos Aires

Índice

1. Alcance

El proyecto tiene como alcance el procesamiento de los sets de prueba encontrados en https://www.kaggle.com/datasets/pavellexyr/the-reddit-irl-dataset en un tiempo razonable. Estos sets serán ingresado al sistema a través de una conexión TCP, ingresando primero todo el set de posts y a continuación el de comments. Luego, el sistema devolverá el resultado de las siguientes consultas:

- Promedio de score de todos los posts
- URLs de memes que gustan a los estudiantes (con comments sobre university, college, student, teacher, professor y con score mayor al promedio)
- Url del meme con mejor sentiment promedio

Además el sistema logra un "graceful quit" frente a señales SIGTERM.

2. Arquitectura del Software

Como se dijo anteriormente, los sets de datos serán ingresados a través de un puerto expuesto.

Una vez se encuentren dentro del sistema, el procesamiento de los datos será repartido a través una red de controladores que siguen el modelo "MapReduce". Cada uno de estos controladores será responsable de completar una pequeña subtarea del pipeline de tareas que representa cada una de las tres consultas implementadas.

Para la comunicación y coordinación de los distintos controladores, se usó un "Message Oriented Middleware" (RabbitMQ).

Por último, para el monitoreo y manejo de caídas de los distintos servicios, se implementó un TaskManagement explicado más abajo.

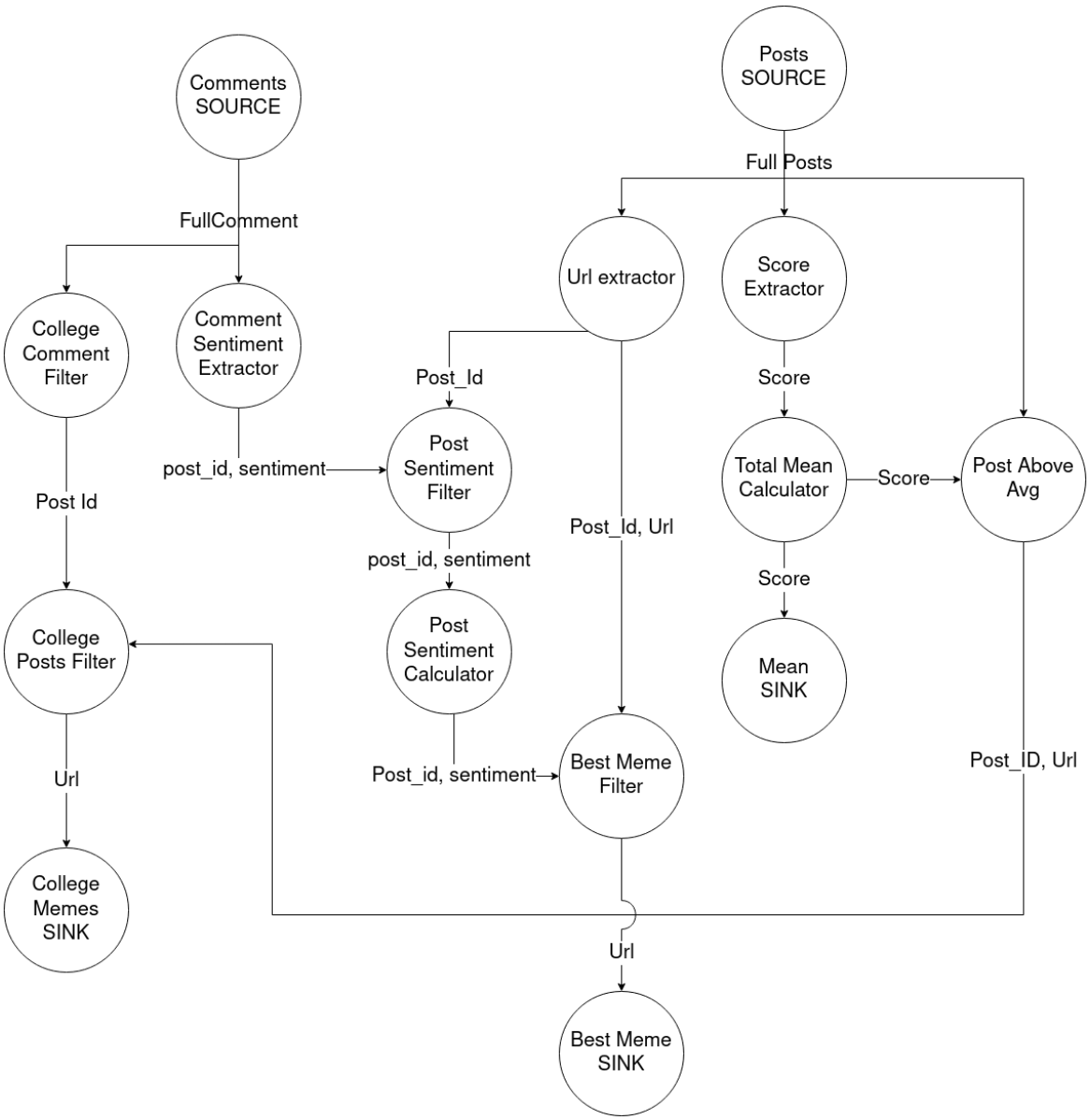
3. Objetivos y limitaciones arquitectónicas

Los objetivos y limitaciones impuestos fueron los siguientes:

- Definición de un middleware para la comunicación basada en grupos
- Múltiples ejecuciones subsecuentes del procesamiento
- El sistema debe mostrar alta disponibilidad hacia los clientes
- El sistema debe ser tolerante a fallos como caída de procesos
- El sistema debe permitir ser re-ejecutado una vez finalizado el procesamiento sin necesidad de reiniciar el servidor

4. Vista lógica

El siguiente DAG muestra la funcionalidad a implementar por el sistema:



Este gráfico representa la lógica detrás del procesamiento de las 3 queries pedidas representadas como los SINK. Para la obtención del Promedio de score de todos los posts, sólo es necesario consumir de Posts para extraer la puntuación de los mismos y posteriormente realizar el cálculo.

Para los memes estudiantiles, se filtran aquellos comentarios de estudiantes y se realiza un join con aquellos posts que superen el promedio de puntuación.

Por último, para calcular el mejor meme, se obtiene a partir del join entre los posts originales y aquellos a los que se les pudo calcular un sentiment. De todos ellos, se selecciona el mejor.

4.1. Concurrencia

El sistema fue implementado usando multiprocessing en un contexto de multicomputing. Para la sincronización se utilizaron colas bloqueantes que almacenan las tareas pendientes (es decir, los datos a ser procesados).

Además se utilizaron locks booleanos para manejar el proceso de cierre ordenado.

4.2. Protocolo de mensajería

Para el pasaje de valores entre los distintos nodos a través de las colas se definieron varios tipos de mensajes con sus respectivo campos. Los mismos son capaces de serializarse y deserializarse. Previo a su envío, los mensajes son agrupados en un BulkMessage con el propósito de reducir la cantidad de mensajes y la cantidad de lecturas.

Para la comunicación entre cliente y servidor, se implementó un protocolo donde se envía codificado en binario un string de longitud variable, especificando previamente la longitud en bytes del mensaje:

LEN_MSG (8 bytes) | VALUE (string)

4.3. Unicidad de mensajes entre procesos

Para soportar la recuperación frente a la caida de un proceso, tenemos que que recuperar el estado anterior a la caida haciendo uso de un log de transacciones.

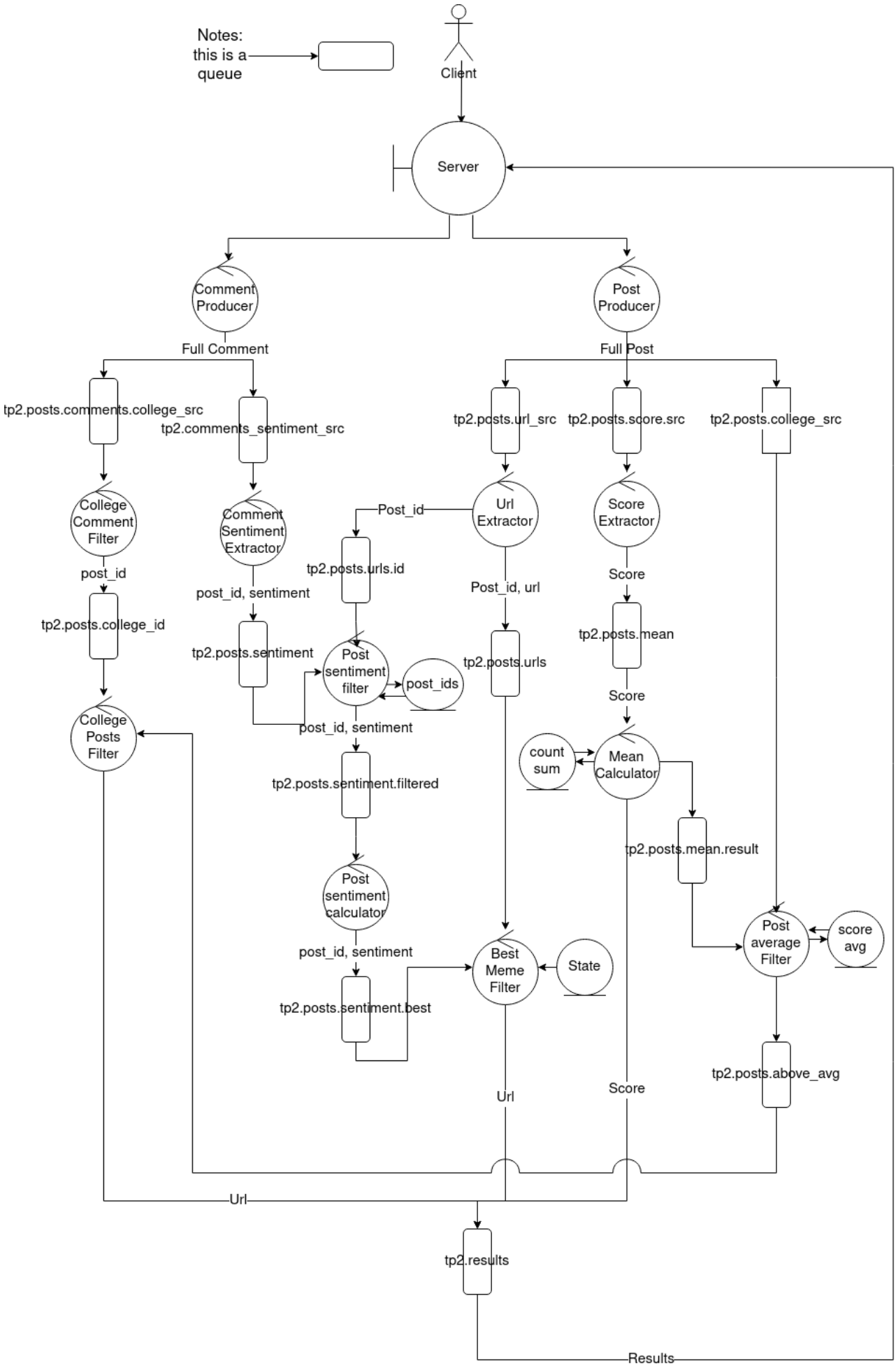
El log nos permite saber si estamos procesando un bulk de mensajes desde 0, si el mensaje fue procesado, guardando el estado del servicio, si envié la respuesta, y si esta respuesta fue "confirmada"

Esto es debido a que, cada vez que enviamos un Bulk, luego enviamos un mensaje "Confirmado". Esto nos permite ignorar mensajes no confirmados e ignorar confirmaciones redundantes.

5. Vista física

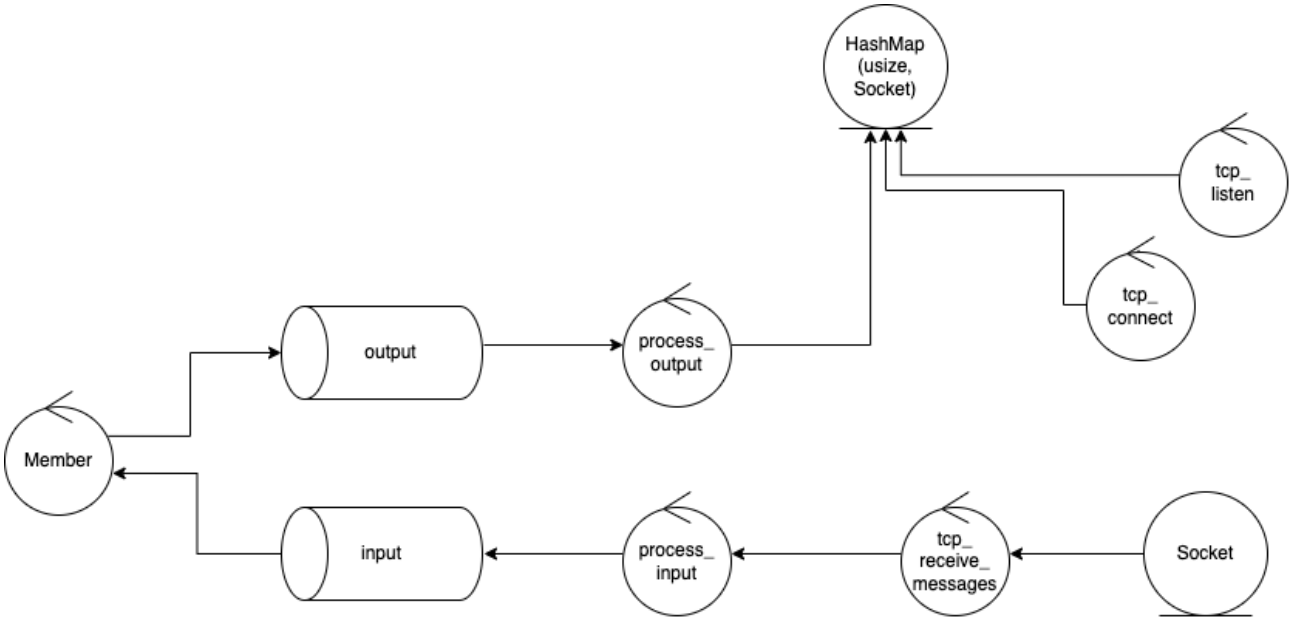
5.1. Diagramas de robustez

Se presenta el diagrama con la información de todos los nodos (excluyendo lo relacionado al monitoreo). Se muestran los distintos controller (que mapean 1:1 con los nodos del DAG) y los estados que almacenan algunos de ellos, además de la comunicación de colas que tienen entre ellos. Notar que el punto de entrada que es "Server" no utiliza colas (en Deudas Técnicas se encuentra explicado el motivo) sino que se comunica directamente por una conexión TCP con los producers. Tambien notar que los servicios son todos de instancia única (explicado en Deudas Técnicas)


Notes:
this is a
queue
Client
Server
Comment
Producer
Post
Producer
Full Comment
Full Post
tp2.posts.comments.college_src
tp2.comments_sentiment_src
tp2.posts.url_src
tp2.posts.score.src
tp2.posts.college_src
College
Comment
Filter
Comment
Sentiment
Extractor
Post_id
Url
Extractor
Score
Extractor
post_id
post_id, sentiment
tp2.posts.urls.id
Post_id, url
Score
tp2.posts.college_id
tp2.posts.sentiment
Post
sentiment
filter
post_ids
tp2.posts.urls
tp2.posts.mean
Score
College
Posts
Filter
post_id, sentiment
count
sum
Mean
Calculator
tp2.posts.sentiment.filtered
tp2.posts.mean.result
Post
sentiment
calculator
Post
average
Filter
score
avg
post_id, sentiment
Best
Meme
Filter
State
tp2.posts.sentiment.best
tp2.posts.above_avg
Url
Score
Url
tp2.results
Results

5.2. Elección de Lider: Bully

Cada member de la eleccion de lider posee una arquitectura de hilos como se muestra en el siguiente diagrama de robustez. El main, denominado member en el diagrama, instancia una clase llamada LeaderElection, que posee un estado que es alterado por mensajes, estos entran y salen por medio de queues. Es interesante destacar que un principio se opto directamente por introducir los sockets adentro de esta clase, esto producia una logica compleja, debido a que se tenia logia de eleccion de lider y logica de comunicacion por sockets. En la solucion que se encuentra actualmente, con queues, tenemos separado la logica.



Cuando levanta un member, este arranca por un lado un hilo listen, para escuchar a que otros miembros se conecten a el. Y tambien levanta la conexion a otros miembros.

Nota (process id): los procesos, tienen id creciente. En la solucion entregada existen 4 procesos, 0, 1, 2, 3.

Observacion (queues): Para evitar confusiones cabe destacar que estas no son de RabbitMQ, sino que son simplemente queues bloquantes que comunican hilos.

Queues:

input: en ella se encontraran todos los mensajes destinados a este member provenientes del resto de los members.

output: en ella se encontraran todos los msgs creados por el member, destinados a ser enviados al resto.

Hilos:

tcp_listen: hilo que realiza socket listener. El socket resultante es ingresado en un mapa indexado por process id. Ademas lanza un nuevo hilo que va a ser el receiver de este socket, este es **tcp_receive_messages**.

tcp_connect: hilo para iniciar conexiones con otros miembros. Con el socket resultante se procede de igual manera que en el caso anterior.

process_input: lee msgs de la queue input y se los pasa al member.

process_output: lee msgs de la queue output y se los manda al member correspondiente.

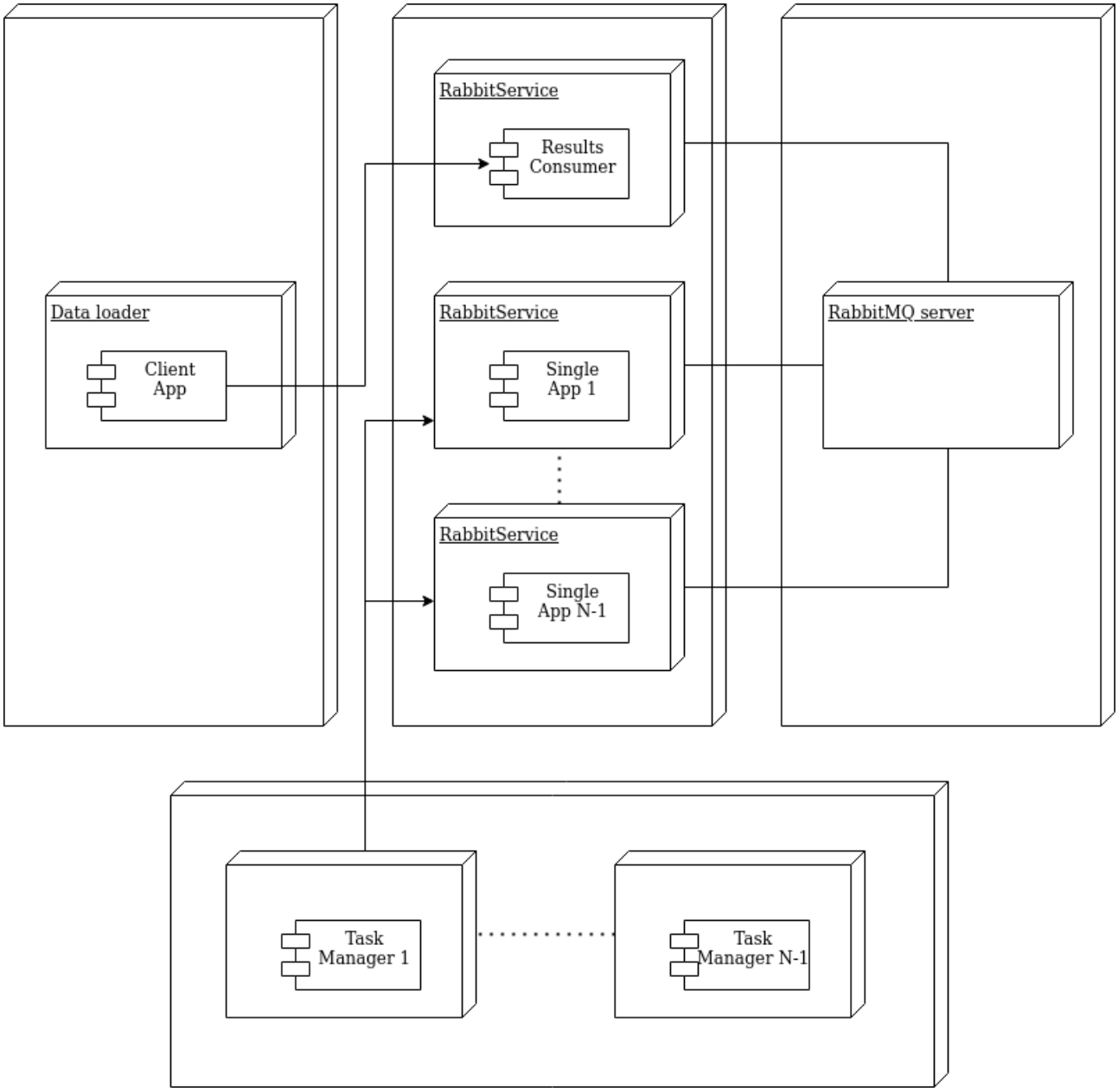
tcp_receive_messages: lee msgs de un socket y los coloca en la queue input.

5.3. Despliegue

El despliegue del Analyzer HA es similar al de su versión sin tolerancia a fallos. Podemos conectar un cliente al servicio Results Consumer, que se encarga de subirle los archivos a los post y comment producer.

Cada servicio es independiente de los demás, y se comunican entre sí utilizando un servidor de colas de mensajes RabbitMQ.

Para tolerancia a caidas del servicio se agregar instancias de TaskManager. Los Task Manager no son propiamente parte del procesamiento de mensajes, su función es levantar instancias de Rabbit Service caidas. Si bien utilizamos docker in docker para levantar procesos, utilizando una API HTTP de docker u otro servicio de containers que provea una API externa, podemos deployar los task managers en nodos separados.



6. Vista de procesos

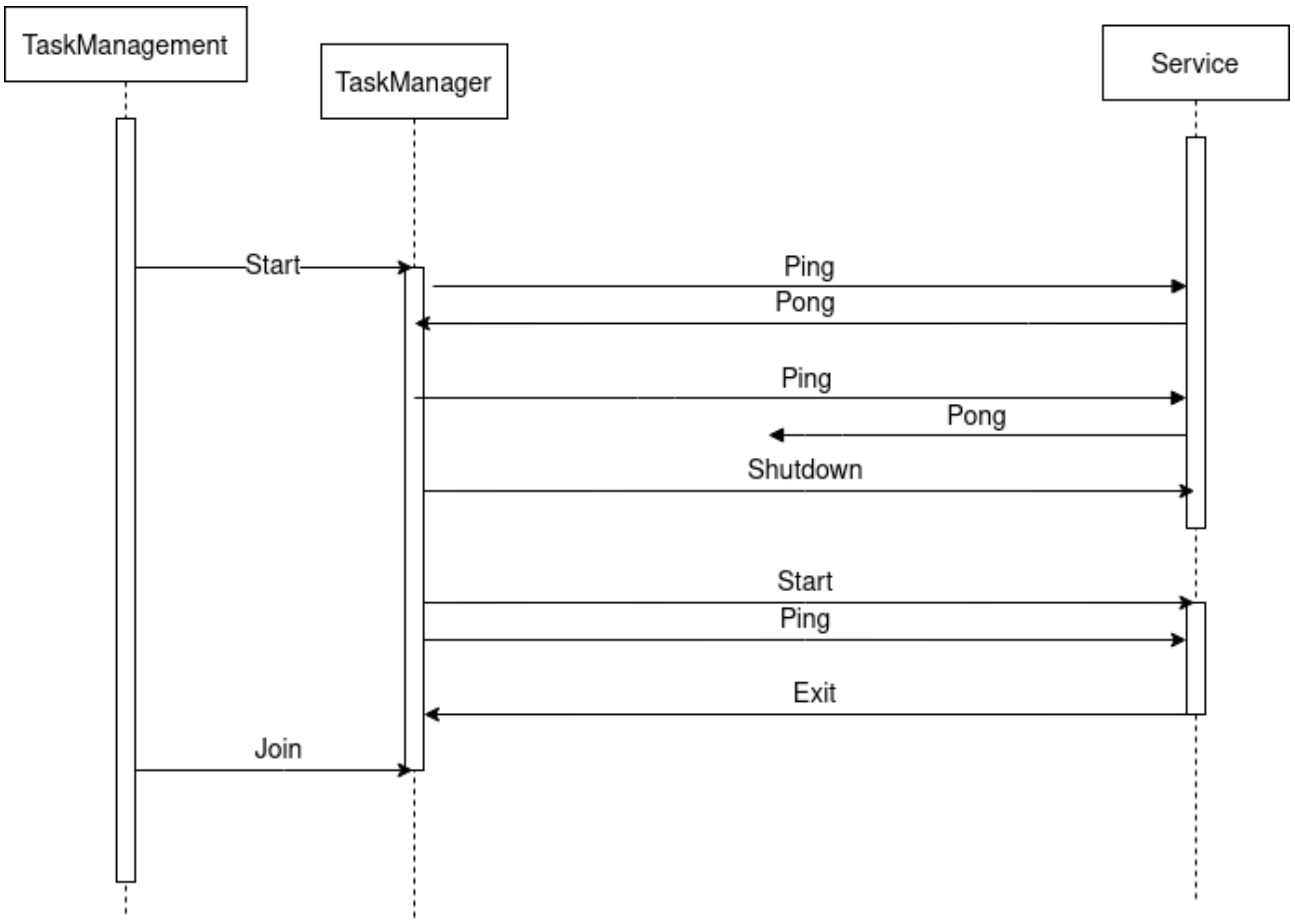
6.1. Task management

Para el monitoreo del estado de los procesos, se creó el TaskManagement. Este service es capaz de ser replicado en varias instancias entre las cuales se eligirá un líder. Este líder, lanzará hilos cada uno con un TaskManager que se encargará de monitorear un único servicio (la información de todos los servicios a monitorear se encuentra en un archivo de configuración).

Por otro lado, cada servicio en su arranque se queda escuchando a la espera de posibles monitores. Una vez acepta a alguno, se comienza una secuencia de Ping-Pong.

Llegado el caso en que un TaskManager detecte que la conexión se cayó o que el tiempo de respuesta exceda un timeout, el TaskManager se encargará de apagarlo ("docker stop") y después de levantarlo nuevamente ("docker start"). Luego, continuará monitoreándolo.

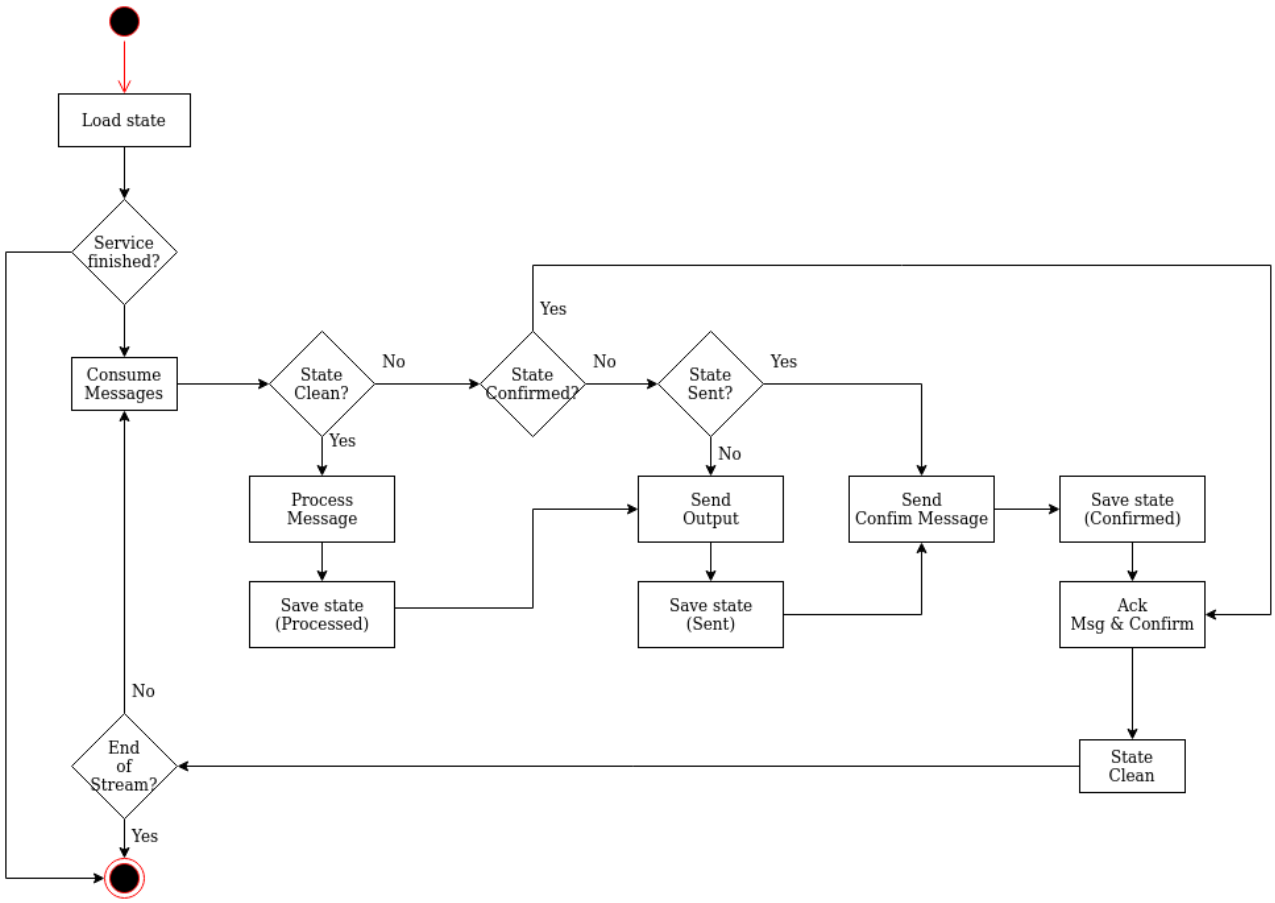
Cuando el servicio quiera salir ordenadamente, le enviará un mensaje de EXIT al TaskManager quien finalizará su ejecución.



6.2. Recuperación de procesos

Cada proceso (a veces llamado "nodo") está compuesto por 1 o 2 RabbitServices, que contienen adentro un MessageProcessor. Son los MessageProcessors los encargados de manejar la lógica de procesamiento de mensajes individuales, así como también definen un estado interno propio. Las instancias de RabbitService interactúan con este estado mediante los métodos *get_state* y *set_state*. Estos métodos se aplican sobre objetos serializables, ya que se persiste el estado del procesor por cada paquete (bulk) de mensajes procesado.

El procesamiento de los mensajes se explica en el siguiente gráfico

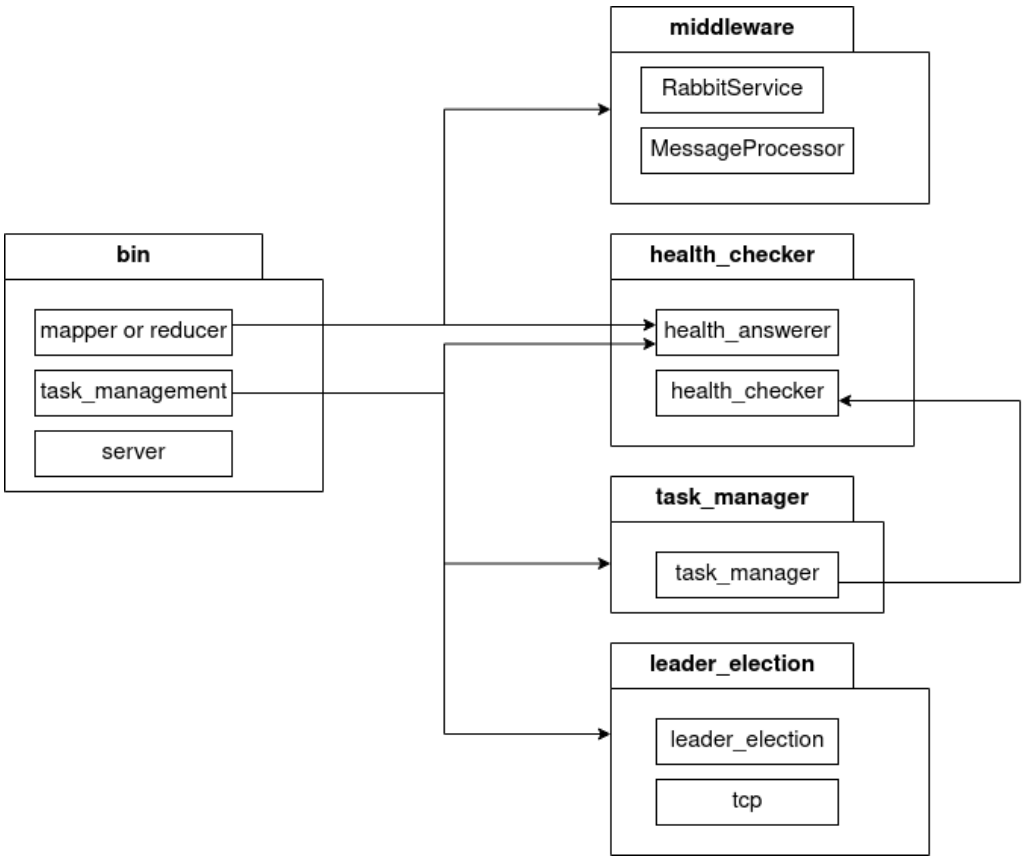


Como se puede ver, por cada mensaje enviado se envia otro paquete de confirmación. Esto nos ayuda a identificar paquetes duplicados sin depender de un id de paquete. Dado que las colas son únicas por proceso, no existen mensajes distintos sin que haya pasado una confirmación antes. En caso de que el proceso se caiga luego de enviar un mensaje pero antes de registrar el envío, al volver a levantarse, este último paquete es reprocesado y reenviado. El nodo consumidor de estos mensajes va a reconocer que son 2 mensajes sin un “confirm” en el medio, y los descarta, porque son iguales.

El receptor de estos mensajes debe tomar mensajes en una ventana de tamaño 2, y descartar todos los mensajes que no sean Mensaje seguido de Confirm.

7. Vista de desarrollo

A continuación se presenta el diagrama de paquetes, ilustrando la organización del código y las dependencias entre los mismos.



En la carpeta bin se encuentran todos los ejecutables. Estos son todos los servicios (ya sea un mapper o un reducer), el task management y por último server. Los servicios hacen uso del paquete Middleware que se encarga de encapsular la comunicación con RabbitMQ. Además utiliza health_answerer para contestar PINGs.

Por otro lado, Task Management hace uso de leader_election que encapsula el algoritmo de elección de líder. Además utiliza la librería de task_manager para delegar el monitoreo por servicio.

8. Escenarios

El principal caso de uso que tiene el sistema es aquel en el que el cliente envía los posts, luego envía los comments. El servidor a medida que obtiene resultados de las 3 queries pedidas, se las enviará al cliente. Una vez el servidor termine de procesar todos los datos, enviará un mensaje de “Finished” al cliente y se prepará para procesar el siguiente cliente en la cola de pedidos.

El otro caso de uso es en caso de que ocurra algún error por le cual el servidor no sea capaz de procesar los resultados (ya sea porque el sistema se encuentra caído o en un estado inconsistente) enviará ”Servidor no disponible”.

9. Tamaño y performance

Para el set de pruebas provisto por Kaggle donde el set de posts contiene 3362747 líneas y pesa 508.7 MB y el set de comments contiene 12053027 y pesa aproximadamente 2.5GB el sistema tarda aproximadamente 3 minutos en procesar un cliente que envíe este dataset.

10. Deudas técnicas

10.1. Server (Entrypoint)

10.1.1. Comunicación a través de sockets

Una pequeña deuda técnica es la de implementar la conexión entre Server (el entrypoint) y sus consumidores (Post Producer y Comment Producer) a través de RabbitMQ. Actualmente se utiliza sockets y TCP por cuestiones de tiempo y por conseguir una implementación lo más rápido posible.

10.1.2. Server no se recupera

Por cuestiones de tiempo, no llegamos a implementar la recuperación después de una caída por parte del Server. Por como funciona actualmente el sistema, el servidor envía por socket primero los posts y a continuación, los comments que recibe del cliente a Post Producer y Comments Producer respectivamente, luego se queda esperando su resultado para enviar al cliente.

Detectamos 2 casos patólogicos:

- Enviar Posts y caerse después (o durante el envío de Posts) sin llegar a enviar Comments. Esto genera que cuando Server vuelva a levantarse y comience a procesar clientes, el sistema se encuentre cargado con Posts inválidos.
- Que Server se caiga antes de sacar de la cola de resultados el resultado asociado a un cliente. Esto genera problemas porque una vez vuelva a levantarse y procese nuevos clientes, va a estar devolviendo resultados de procesamientos previos y no de los pedidos actualmente.

La solución que discutimos fue asociar a los cliente con un ID incremental y firmar los paquetes con el mismo. De esta manera, en aquellos nodos que deban hacer joins, pueden ignorar los mensajes que no coincidan con el id actual (es decir, descartan el id más viejo) hasta que ambos flujos de los que consumen comparten el mismo id.

10.2. Cierre de Task Management

El cierre de Task Management no está pulido al 100 %. Como se explicó anteriormente, el Task Management deja de intentar levantar un servicio una vez dicho servicio le envía un EXIT, indicando que está cerrando en forma gracefully. Estas notificaciones de EXIT no se están guardado en ningún estado. Entonces podría ocurrir que al momento del cierre, justo se caiga el TaskManagement y se lance una elección de líder y el nuevo líder no sabría que algún servicio cerró ordenadamente e intentar levantarlo.

Aún así, es una fracción de tiempo ínfima porque:

- Todos los ejecutables hacen handle del SIGTERM y en particular, el TaskManager no se queda esperando a que todos los servicios le confirmen que están cerrando, simplemente cierra.
- La elección de líder lleva unos segundos, entonces seguramente las réplicas se cierren antes de completar el líder por alguna caída inesperada del TaskManager líder.
- Cuando recién inicia un nuevo líder, es más tolerable ante la imposibilidad de conectarse con algún servicio. Hace 3 intentos con un timeout de 30s de por medio. Esto es para que en el arranque del sistema, el TaskManager le de la posibilidad a todos los servicios de iniciar correctamente. Y cuando un TaskManager gana el liderazgo, se sigue este mismo flujo. Por este motivo, necesita que falle 3 veces este intento de conexión antes de levantarlo nuevamente y es muy poco probable que no le llegue un SIGTERM antes de levantarlo.

10.3. Mejoras en comunicación por Bulk messages

La comunicación se realiza en forma de paquetes. El empaquetado inicial se hace según el tamaño en bytes del payload, los servicios procesan de a 1 paquete y por cada paquete procesado envían un paquete transformado

(excepto aquellos que sólo consumen paquetes y emiten un resultado final). Esto genera que el payload de los paquetes se vaya achicando, pudiendo utilizarse un sistema de buffer que provea mejor comunicación. Se podría explotar más el protocolo de mensaje + confirm para facilitar esta mejora.

10.4. Paralelismo

Se podría hacer paralelismo utilizando sharding y colas para cada instancia del servicio. En la versión base del Analyzer, se utilizaba una cola para todas las instancias del servicio lo que dificultaba implementar la recuperaración del estado anterior del proceso.

10.5. Casos bordes de caida de servicios

Algunos servicios tienen casos borde que pueden llevar a errores del sistema: los servicios que cuentan de 2 procesadores de mensajes (por ejemplo, *best meme filter* toma primero el id del mejor meme y luego consume las url entrantes para saber cuál corresponde a ese id) pueden quedar en un estado inconsistente si no eliminan los transaction logs de los procesadores. Esto se soluciona utilizando un solo transaction log, con mejor soporte para multiples MessageProcessor

10.6. Mejoras en transaction log

El transaction log almacena poca información, no es incremental (guarda el estado completo en cada save), y cada linea es un json, por lo que la unica atomicidad de escritura que tenemos es que ese json se pueda decodificar o no. Debería tener soporte para estados incrementales (separando un save state parcial de uno total). Además el transaction log es local, en un escenario real debería ser un servicio externo.