Trabajo Práctico N. 2
Reddit Memes Analyzer

Fecha:
9 de Junio
2022

Docentes:
- Pablo D. Roca
- Ezequiel Torres Feyuk
- Ana Czarnitzki
- Cristian Raña

Alumno:
- Sebastian Ripari (96453)

Introduccion

El trabajo práctico fue desarrollado en **Rust.** Se llega a la solución mediante procesamiento coordinado de varios procesos. Para la comunicación de los mismos se utilizó queues, provistos por **RabbitMQ.**

Procesos

Se cuenta con procesos que poseen diferentes responsabilidades. Los que se encargan de tareas más costosas y tienen la ventaja de que eran sin estados, fueron replicados.

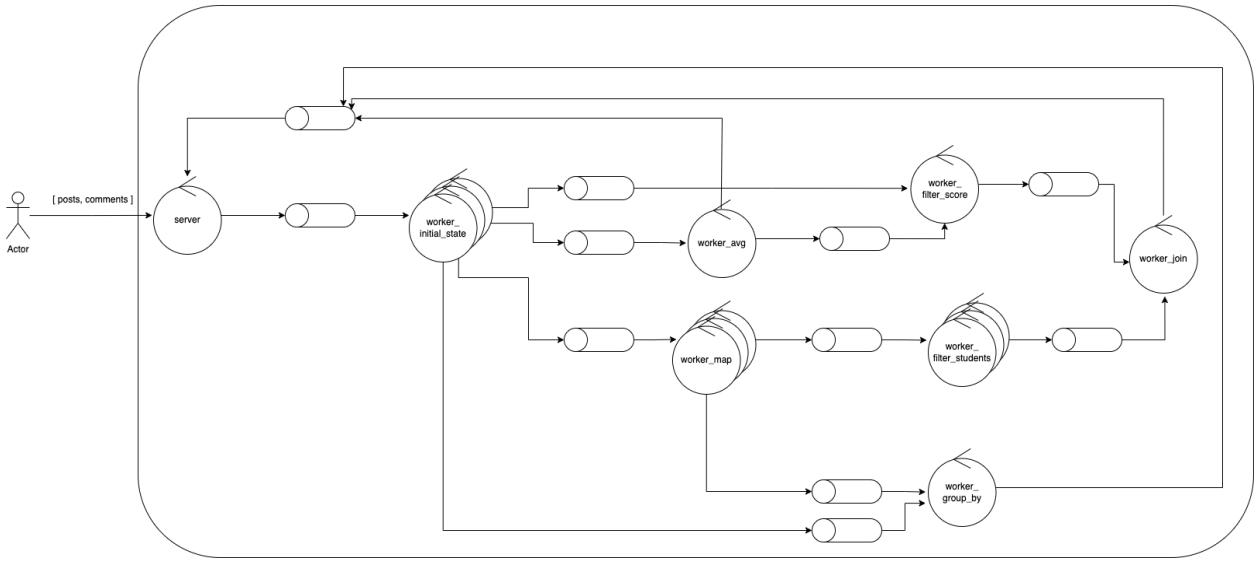


Diagrama de Robustez

Se enuncia a continuación la principal actividad de cada proceso:

server: Recibe y envía mensajes al cliente por socket.

worker_initial_state: consume posts y comments, para dárselos al proceso correspondiente.

worker_avg: consume todos los score, calcula el average.

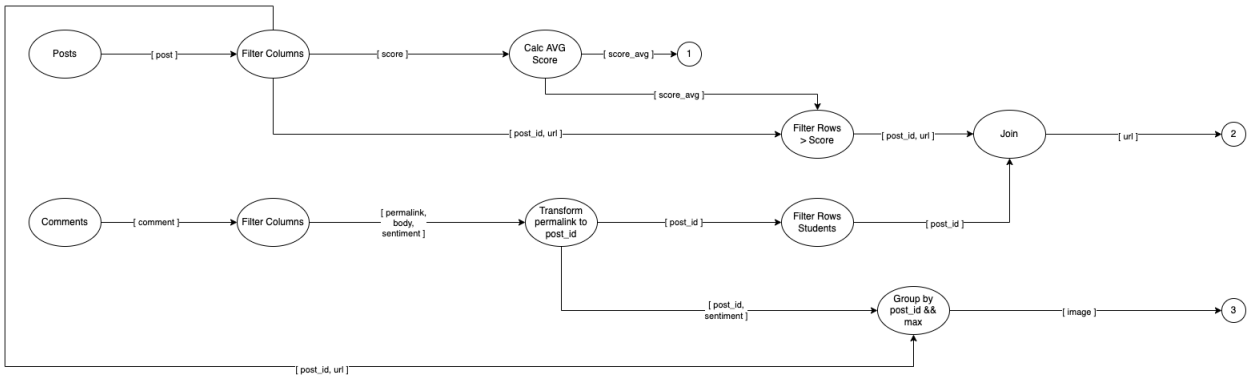
worker_map: consume todos los permalink de los comments, los parsea para encontrar el post_id.

worker_filter_students: consume todos los body de los comments, indicando cuales matchean con un estudiante.

worker_filter_score: consume todos los post y score average filtrando aquellos post que tienen un score mayor al average.

worker_join: consume los post que tengan un score superior al average y los comments de estudiantes. Hace un join de estos por post_id, encontrando cuáles post cumplen ambas condiciones.

worker_group_by: consume los post y los comments, hace un group_by post_id de los comments y realiza la función de agregación average sobre el sentiment. Luego se queda con el que posee el máximo.



DAG

Queues

Se cuenta con varias queue, estas son:

AVG_TO_FILTER_SCORE
QUEUE_INITIAL_STATE
QUEUE_COMMENTS_TO_FILTER_STUDENTS
QUEUE_COMMENTS_TO_GROUP_BY
QUEUE_COMMENTS_TO_JOIN
QUEUE_COMMENTS_TO_MAP
QUEUE_POSTS_TO_AVG
QUEUE_POSTS_TO_FILTER_SCORE
QUEUE_POSTS_TO_GROUP_BY
QUEUE_POSTS_TO_JOIN

Cada una de ellas fue pensada como punto de entrada de cada uno de los diferentes workers, por ejemplo `QUEUE_COMMENTS_TO_FILTER_STUDENTS` es aquella que es utilizada por el worker_filter_students.

Middleware

Toda la comunicación de un proceso con otro, es encapsulada por medio de un middleware. Este realiza toda la funcionalidad, desde la creación y destrucción del canal y de los recursos, hasta el envío de mensajes. Estos mensajes pueden ser de dos tipos, el mensaje normal que contiene información, o el mensaje end, que indica que ya no habrá más mensajes para consumir.

Funcionalidad:

middleware_connect
middleware_create_channel
middleware_declare_queue
middleware_create_consumer
middleware_create_exchange
middleware_send_msg_end
middleware_send_msg
middleware_end_reached
middleware_consumer_end

Hubiese sido ideal juntar connect, create_channel y create_exchange en una sola, pero por dificultades del lenguaje no fue posible. Otro deseable hubiese sido poder crear una clase que tenga como atributos, el exchange, el channel y la conexión, pero por lo mismo no fue posible. Así que la interfaz es al estilo C, donde por parámetro se pasa todo lo que se necesita.

Detección de parar de consumir

Se utilizó la técnica de mandar mensajes indicando el end. Debido a que algunos procesos están replicados, es necesario que los productores manden tanto mensajes de end como consumidores allá. Toda lógica se encuentra dentro del middleware, más precisamente en middleware_consumer_end. Comienza usando la función middleware_end_reached detecta cuando le llegó la cantidad de ends necesaria para parar de consumir. Y en este punto middleware_send_msg_end sabe como enviar los ends a los consumidores, cuántos de ellos enviar.

Por configuración de docker-compose especificamos en cada proceso cuantos producers y consumers tiene. En el caso del proceso map (worker_map) tiene dos tipos de consumer

(worker_filter_student y worker_group_by), así que desde el docker-compose-yml se le indican dos valores separados por coma, indicando la cantidad.

```
worker_avg:
   container_name: worker_avg
   image: worker_avg:latest
   entrypoint: /tp2/target/release/worker_avg
   environment:
     - LOG_LEVEL=info
     - RABBITMQ_USER=root
     - RABBITMQ_PASSWORD=seba1234
     - N_PRODUCERS=${N_WORKER_INITIAL_STATE}
     - N_CONSUMERS=1 # 1 worker_filter_score
   depends_on:
     - server
   networks:
     - tp2_net
```

```
worker_map:
   image: worker_map:latest
   entrypoint: /tp2/target/release/worker_map
   deploy:
     mode: replicated
     replicas: 6
   environment:
     - RABBITMQ_USER=root
     - RABBITMQ_PASSWORD=seba1234
     - LOG_LEVEL=info
     - N_PRODUCERS=${N_WORKER_INITIAL_STATE}
     - N_CONSUMERS=${N_WORKER_FILTER_STUDENTS},1   # 1 worker_group_by
   depends_on:
     - server
   networks:
     - tp2_net
```

URL de memes de estudiantes con score mayor al promedio

En el siguiente diagrama se muestra como se lleva a cabo la búsqueda de los memes que le gustan a los estudiantes con un score superior al promedio. De los 3, es el que más procesos involucra. Vemos entonces las actividades que cada uno realiza y con quien interactúa.

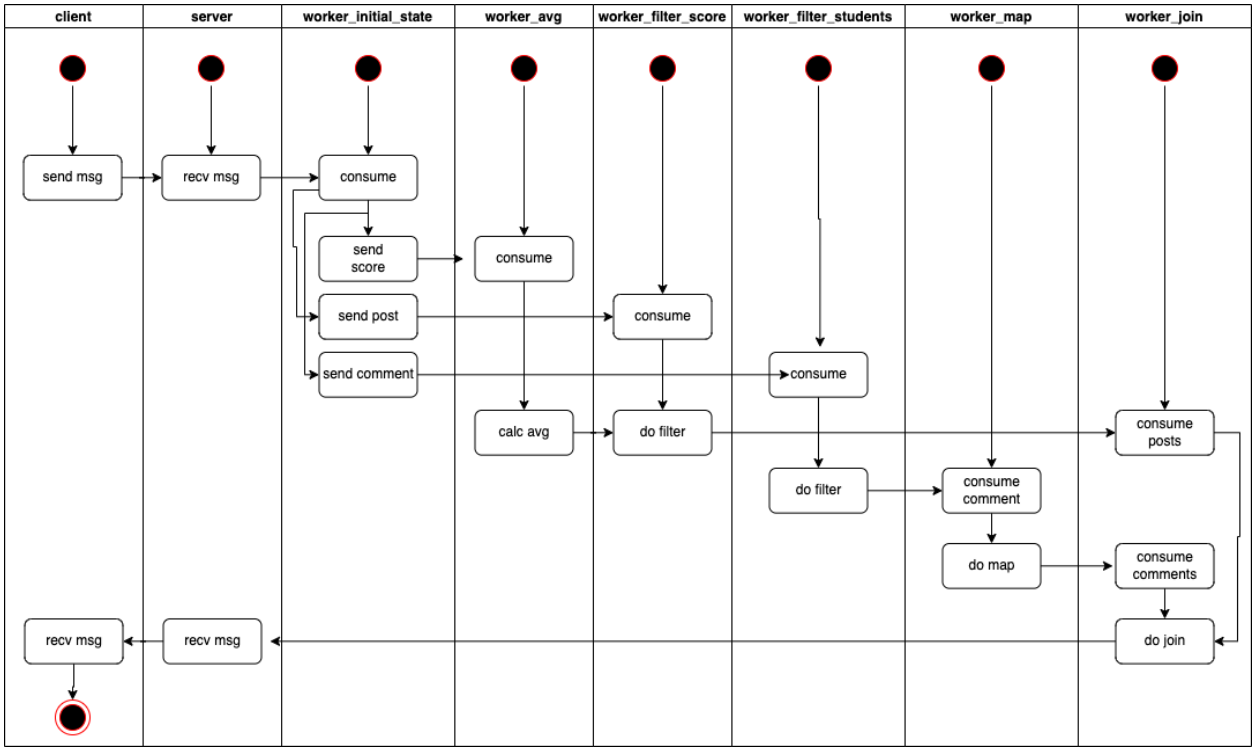


Diagrama de Actividades

Meme con mejor sentiment

Para encontrar el meme con mejor sentiment tenemos un proceso llamado worker_group_by
Se encarga agrupar por post_id todos los sentiments de los comments, luego calcula un avg de cada uno de ellos, y se queda con el max.

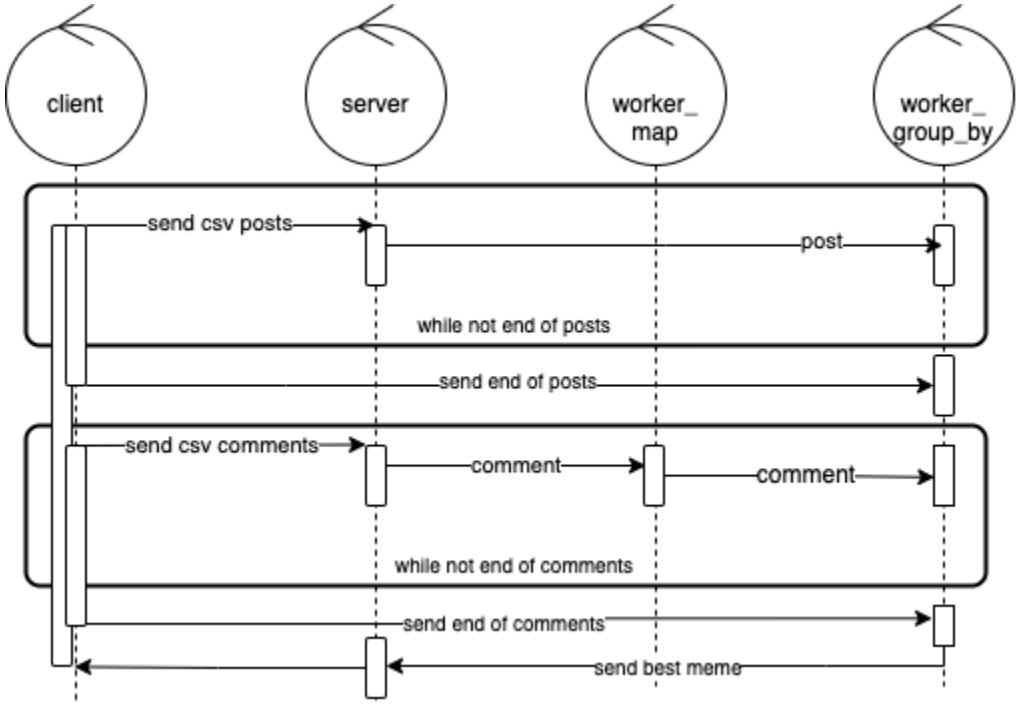


Diagrama de Secuencia

Conclusiones

El presente trabajo práctico fue la primera vez que me tope con RabbitMQ, y con la construcción de un middleware que facilite la comunicación entre procesos a traves de Rabbit. Por lo tanto fue de mucho aprendizaje.

Fue muy interesante manejar un gran volumen de datos, ver que el programa demore mucho tiempo en procesar todo el input, y sobre esto pensar cómo lograr que valla mas rapido mediante la repartición de tareas y envio batch.

En cuanto al middleware fue algo que arrancó de a poco, pasando la lógica de comunicación del proceso al mismo. Fue satisfactorio ver como el código de aplicación se iba haciendo más y más simple, si bien con algo más de tiempo podría haberlo mejorado aún más.

Me quedó el desafío de cómo lograr que en Rust pueda tener una carpeta *commons* con la entidad logger por ejemplo. Para no duplicarla en cada proceso.