Trabajo Práctico 2

Ventura Julián
Padrón: 102391

Sistemas Distribuidos I
1ºC 2022
Fecha de Entrega:
26/05/2022

Contenido

Decisiones Tomadas

A continuación, se listan una serie de decisiones tomadas a la hora de diseñar e implementar la arquitectura.

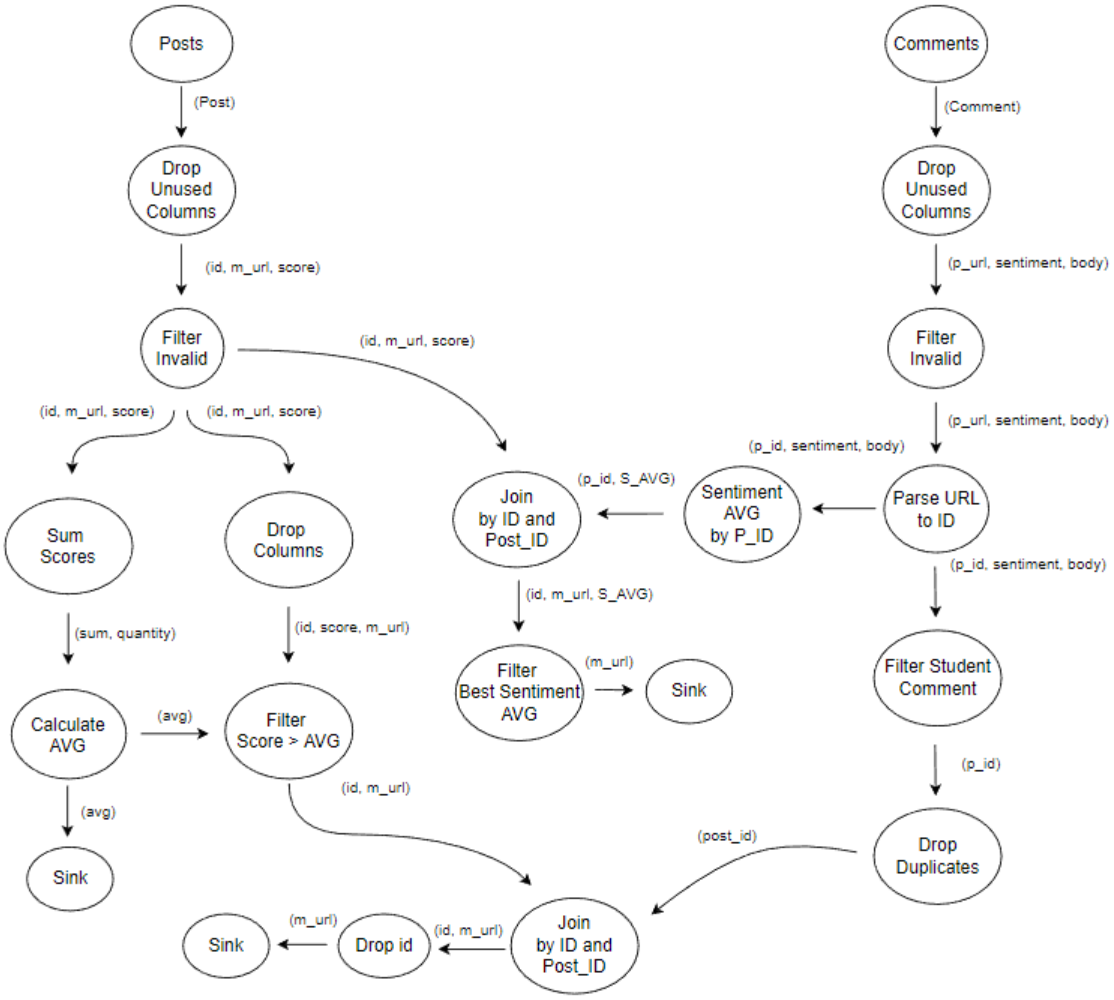
- El protocolo se implementará en binario, con el objetivo de minimizar el transporte de información entre cliente y servidor.
- El servidor deberá soportar una conexión con un único cliente, la cual disparará la ejecución de tres operaciones sobre el stream de datos recibido. Tras la culminación de estas, el servidor finalizará.
- El empleo de la señal SIGTERM para la finalización de los procesos será necesaria en toda ejecución del sistema.
- Al recibir una señal SIGTERM, todo proceso del sistema buscará realizar una salida graceful, interrumpiendo las operaciones y abriendo la posibilidad a perdidas de datos.
- El stream de datos, tanto de posts como de comentarios se recibe de forma paulatina e intercalada por parte del cliente.
- El servidor conoce el formato de los datos a recibir, pero no posee datos precargados.
- La conexión con el cliente se realiza por TCP, a partir de un protocolo binario hecho a medida para esta problemática.
- La ejecución de las operaciones y almacenamiento temporal de los datos se realizará en memoria.
- Los datos de entrada, tanto de posts como de comentarios, pueden tener datos incompatibles que deberán ser filtrados.
- La obtención meme de sentimiento promedio máximo será realizada luego de ejecutar una operación de join entre comentarios y posts, para garantizar la obtención de un resultado, siempre y cuando el join no sea vacío. Esto debido a que puede no tenerse información del post cuyo sentiment promedio resultó ser el mayor, sea porque fue filtrado o porque no existía en el set de datos.

Arquitectura Propuesta

A continuación, se explicará la arquitectura utilizada en el sistema, con la ayuda de diagramas.

Diagrama de Flujo de Datos

En el siguiente diagrama DAG se puede observar cómo es el flujo de datos desde la recepción de los posts y comentarios, hasta la obtención de los resultados finales del cómputo en los nodos marcados como sink.



Las tuplas que acompañan a las aristas representan las tuplas de datos que viajan por el pipeline de una etapa a la otra.

Como se ha comentado en las decisiones tomadas, se incluyen etapas de filtrado de datos para ambos streams. Adicionalmente, la etapa de join de sentimiento promedio antecede al cálculo del mejor sentimiento promedio, para garantizar un resultado ante la posible falta de posts.

Diagrama de Robustez

En los siguientes diagrama observaremos una vista de la interacción entre los controladores, actores y entidades del sistema. En este caso, cada controlador fue implementado como un proceso corriendo en una máquina aislada, con conexión a una red común con los demás procesos del sistema.

Las aristas representan con su sentido, cómo fluyen los datos entre los distintos controladores. Los datos en cuestión son representados, al igual que en el DAG, como tuplas.

Se añade una vista lógica de la conexión entre los distintos controladores, a partir de colas de mensajes que podrán ser de tres tipos distintos:

- Worker Queue: Refiere a la cola de mensajes productor consumidor habitual, la cual permite múltiples consumidores y productores. En el diagrama es referenciada símplemente como “Queue”.
- Topic Queue: Refiere a una cola de mensajes en la cual es posible introducir mensajes por tópico. Admite múltiples productores y múltiples suscriptores. La clave es que los suscriptores podrán suscribirse a la cola para escuchar mensajes con algún tópico en particular.
- Fanout Queue: Idéntica a la cola de tópicos, sólo que no admite filtrado por tópico por parte de los consumidores.

Debido a la complejidad del diagrama, se decidió fraccionarlo en dos partes, cada una de las cuales hace foco en los procesos involucrados en las operaciones que se pretenden detallar.

Adicionalmente, se omitió al proceso administrador de mensajes “middleware_admin”, debido a que no aportaba a la comprensión de los diagramas.

Diagrama de Robustez Parte A

El siguiente diagrama muestra a los procesos involucrados en los cálculos de puntaje promedio de posts y posts con algún comentario relacionado al ámbito escolar, que superen al promedio en puntaje.

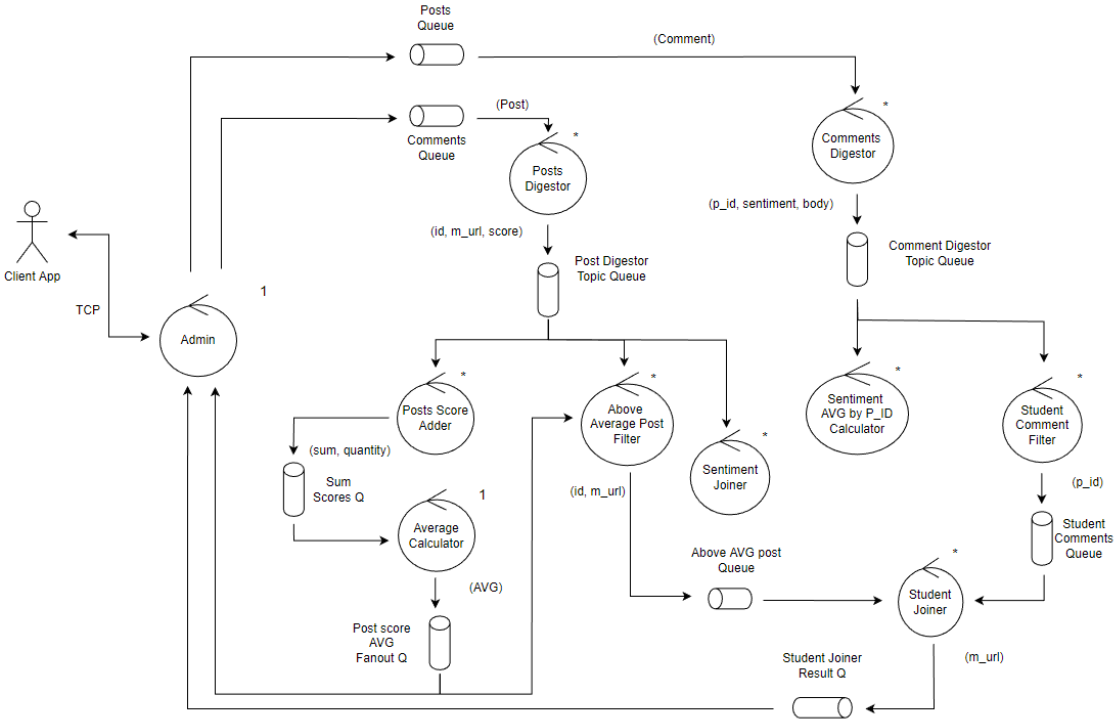
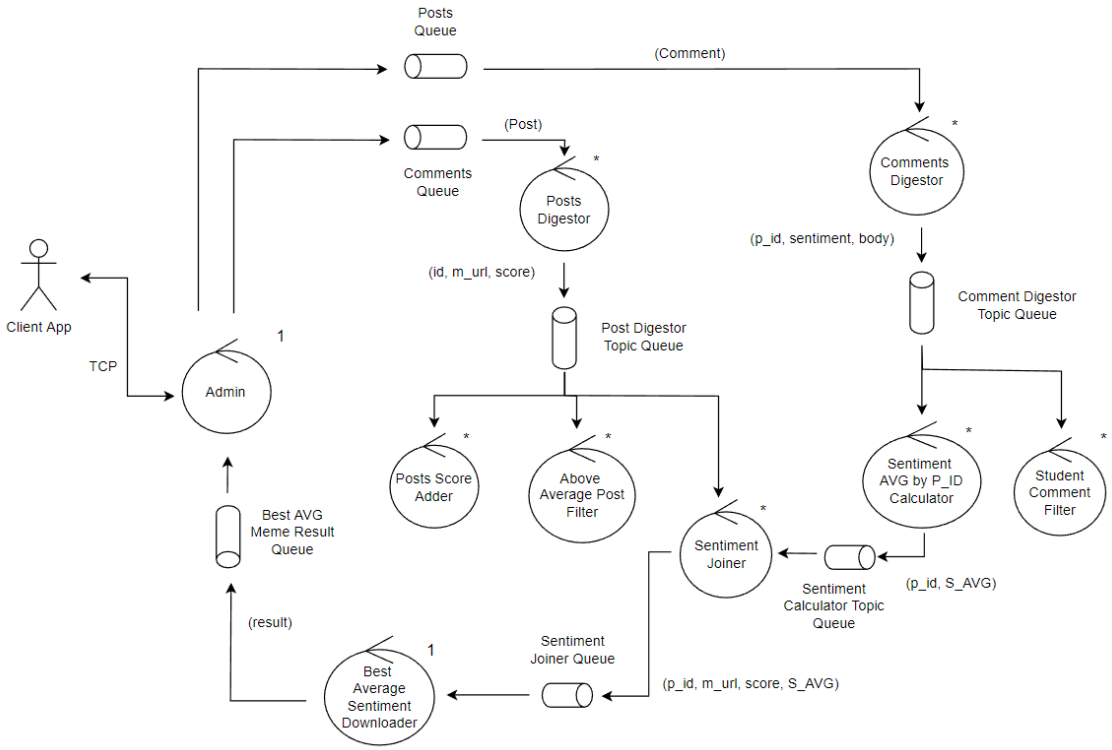


Diagrama de Robustez Parte B

El siguiente diagrama muestra a los procesos involucrados en la obtención del meme cuyo post tuvo el mejor sentimiento promedio entre sus comentarios.



Finalmente, puede apreciarse que se incluyó a un servicio de logging externo al sistema. Este servicio fue incluido de forma externa al diagrama por dos motivos, el primero es que no fue implementado para el trabajo sino que se utilizó una biblioteca ya existente de Golang denominada logrus; el segundo es que todos los procesos del sistema tienen acceso a él para emitir logs.

Explicación de la Arquitectura

A continuación, se brindará una breve explicación de los procesos involucrados en la arquitectura.

Es importante notar que se hablará de “Al finalizar, …”. La señal de finalización de entrada de datos se detecta a partir del cierre de la cola de entrada, tarea que es responsabilidad del middleware de mensajes implementado y se detallará en una sección siguiente.

- Admin: Este proceso sirve de interfaz del sistema hacia el exterior. Abrirá una conexión en un host y puerto determinado y esperará a la llegada de un cliente. Luego recibirá el stream de posts y comentarios proveniente de este y los colocará en las colas de posts y comentarios, respectivamente. Tras la finalización del stream, quedará escuchando en las tres colas de respuesta de puntaje promedio de post, mejor meme por sentimiento, posts escolares por encima de la media. Finalmente, reportará los resultados al cliente y finalizará su ejecución.

- Post Digestor: Este proceso descarta todas las columnas de posts que no son utilizadas a lo largo de la ejecución del sistema, además de filtrar las filas que contienen columnas necesarias inválidas. Leerá de la cola de posts y escribirá sus resultados en una cola por tópico haciendo balanceo de carga según se le haya configurado. El topico resultante será “post_digestor_result.{number}” donde “number” es el resultado de aplicar una función de dispersión sobre el id del post, teniendo en cuenta el número máximo de procesos “Sentiment Joiner”.
- Post Score Adder: Recibe los scores de los posts por entrada, filtrando por el tópico “#” y acumulándolos en una cola tipo worker compartida entre todos los procesos del grupo. Cada proceso consume de esta cola, acumulando los scores y llevando una cuenta del número de posts que ha recibido. Al finalizar, escribe una tupla con la suma total y el número de posts recibidos a la cola tipo worker del Avg Calculator.
- Post Score Average Calculator: Recibe los resultados de todos los adders y calcula el promedio de todos ellos. Al finalizar, escribe el resultado en una cola tipo fanout “post_score_avg_result”.
- Above Avg Post Filter: Tendrá dos colas de entrada. La primera será una cola de worker compartida entre todos los procesos del grupo, la cual escuchará mensajes por tópico “#” de la cola de resultados del post digestor. En la segunda cola escuchará el resultado del cálculo de score promedio. Comenzará a procesar una vez haya recibido el score promedio. Su operación consistirá en filtrar los posts que superen en score, al promedio total. Escribirá este resultado en una cola por tópico haciendo load balancing de forma análoga al post digestor, teniendo en cuenta el número de procesos “Student Joiner”.
- Comment Digestor: Este proceso descarta todos los comentarios que tengan un valor de sentiment, body o permalink inválido. De las filas válidas, extrae el post id a partir del permalink y escribe como resultado una tupla compuesta del post_id, sentiment y body. Los resultados serán escritos realizando load balancing como ya fue explicado, teniendo en cuenta el número de procesos “Sentiment AVG by P_ID”.
- Sentiment AVG by P_ID: Este proceso agrega el sentiment de cada tupla que recibe por id de post. Leerá de una cola tipo worker anónima (y única a ese proceso) que se suscribirá al topico “comment_digestor_result.{worker_id}” donde “worker_id” es el id numérico, a partir de cero, del proceso lector. Al finalizar escribirá los resultados obtenidos en una cola por tópico, haciendo load balancing a partir del número de procesos “sentiment_joiner”
- Student Comment Filter: Este proceso leerá de una cola de entrada tipo worker compartida por todos los procesos del grupo, suscripta al tópico “#” de la cola resultado de “comment_digestor”. Filtrará todas las filas que correspondan a comentarios del ámbito académico, analizando el campo body. De estas filas, extraerá el campo p_id y lo enviará como resultado, filtrando duplicados. La cola de resultados corresponderá a una cola por tópico “student_comment_filter_result”, sobre la cual realizará load balancing teniendo en cuenta el número de procesos “student_joiner”
- Student Joiner: Este proceso tendrá dos colas de entrada. La primera corresponderá a la salida de “above_avg_post_filter”, la segunda corresponderá a la salida de “student_comment_filter”. Ambas colas serán de tipo worker anónima (únicamente para este proceso) y suscriptas a los tópicos “above_avg_post_filter_result.{id}” y “student_comment_filter_result.{id}” respectivamente, donde “id” representa el id del proceso actual. Primero comenzará leyendo de la cola de posts generando una tabla interna (side join) con cada entrada, hasta que la misma haya finalizado. Luego comenzará a leer de la cola de comentarios, realizando el join con la tabla interna. Si el join pudo ser realizado, escribirá por una cola tipo worker “student_joiner_result”

- Sentiment Joiner: Funciona de forma análoga al “Student Joiner”, solo que leerá de las colas resultado de “post_digestor” y “sentiment avg by p_id”. Escribirá el resultado del join en una cola tipo worker “sentiment_joiner_result”.
- Best Avg Sentiment Downloader: Leerá de la cola tipo worker “sentiment_joiner_result” hasta que esta haya finalizado. Filtrará el post que mejor sentiment promedio tenga, escribiendo el url del meme de este por una cola tipo worker “best_avg_sentiment_downloader”.

Coordinación de Servicios y Middleware

La coordinación de servicios fue realizada a partir de un middleware de mensajería centralizado. Este middleware se encarga abstraer la utilización de las colas de mensaje, para que las mismas sean análogas a los canales nativos del lenguaje utilizado, Golang.

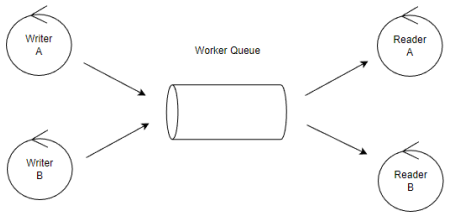
Para esto, el middleware basa su funcionamiento en un Message Oriented Middleware como es RabbitMQ.

Abstracción

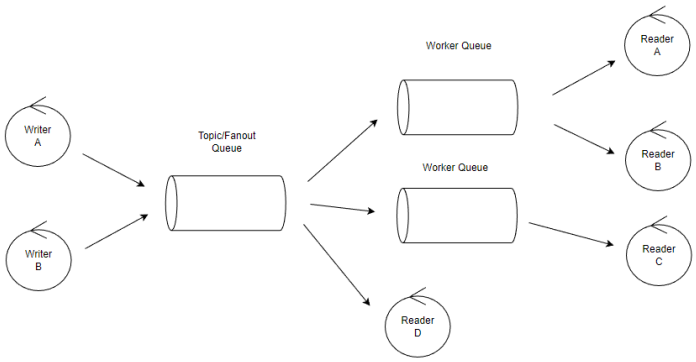
Previamente se introdujo el modelado de colas utilizado por los procesos del sistema, el cual fue basado en la siguiente abstracción:

Worker Queue

Se trata de una cola de mensajes productor consumidor habitual, la cual permite múltiples consumidores y productores, como puede observarse en el siguiente diagrama.



Sin embargo, este tipo de cola se extendió para permitir la posibilidad de defiir un “source” de los datos a leer, de forma tal de poder crear composiciones como la siguiente:



Como se puede observar, un grupo de procesos lectores podrán realizar trabajo repartiendo la carga creando una cola productor consumidor, que tenga como productor a otra cola de tipo “topic” o “fanout”, teniendo la posibilidad de indicar un tópico en el primer caso. En el diagrama, los lectores A y B crearon este tipo de patrón.

Por otro lado, como muestra el diagrama, esto no impide que otros procesos se suscriban a la cola source de la forma que deseen, ya sea a través de una cola worker intermedia o directamente.

Topic Queue

Las colas de tipo tópico tienen una funcionalidad idéntica a sus homónimos en RabbitMQ

Fanout Queue

Las colas de tipo fanout tienen una funcionalidad idéntica a sus homónimos en RabbitMQ

Composición

La composición del middleware puede dividirse en dos partes, la biblioteca que provee una iterfaz al usuario y el administrador que se ejecuta en un proceso distinto.

Interfaz de Cliente del Middleware

La interfaz del cliente, correspondiente a la primera parte de la división, es la que tiene la mayor lógica. La interfaz que provee al usuario es la siguiente:

- Crear nueva cola de tipo lectura, en base a configuración.
- Crear nueva cola de tipo escritura, en base a configuración.
- Crear nueva cola, tipo a determinar en base a configuración.

La configuración mencionada para cada cola es la siguiente:

- Name: Indica el nombre de la cola que será creada, tomando vacío como una cola anónima.
- Class: Indica el tipo de cola a crear, entre Worker, Fanout y Topic
- Topic: Indica el topico al cuál la cola deberá suscribirse.
- Source: Indica a qué cola deberá suscribirse la cola actual a crear, para tomar información.
- Direction: Indica la dirección de la cola a crear, entre Read y Write.

Con esta interfaz es muy sencillo crear colas a partir de un archivo de configuraciones. De esta forma, el proceso que las utilice puede quedar no solo abstraído de su funcionamiento, sino que también de su creación.

Internamente, la interfaz de cliente utiliza la interfaz de RabbitMQ para crear las abstracciones antes mencionadas, declarando colas y exchanges y bindeandolos, valiéndose de la propiedad de idempotencia de declaración que provee el MOM de Rabbit.

Por cada cola creada, el middleware creará un worker en una gorutina encargado de hacer la traducción entre la información provista por rabbit al usuario y viceversa. Cuando la cola es de lectura, el worker traducirá los mensajes provenientes de rabbit en mensajes que respeten la interfaz provista por el mom implementado. Cuando la cola sea de escritura, hará el camino contrario, tomará los mensajes que respetan la interfaz y los enviará por rabbit en formato string.

El problema de la implementación surge cuando se desea notificar el cierre de un canal de comunicación, por ejemplo, la finalización del stream de posts.

Sería ideal que el proceso escritor pudiese cerrar el canal de golang y que esa acción fuese transmitida de alguna forma al proceso lector, que verá su canal (también de golang) cerrado. Esta idea, aunque muy sencilla, llevó varios dolores de cabeza que se detallarán en una sección siguiente.

Para implementar este comportamiento, se realizó lo siguiente:

- Un proceso escritor cierra el canal de golang
- Eventualmente, el worker traductor del mom recibirá esta señal, por lo que enviará un mensaje “finish” por la cola de rabbit.
- El worker lector del otro extremo, al recibir el paquete “finish” cerrará el canal de golang
- Eventualmente, el proceso recibirá la señal de cierre del canal.

A simple vista esto resulta muy sencillo, sin embargo, tiene dos grandes problemáticas. La primera es que no se puede simplemente cerrar un canal cuando hay múltiples escritores, porque, aunque uno haya finalizado, los demás pueden seguir escribiendo. La segunda es que si se tiene lectores compitiendo por los mensajes en una cola (una cola worker), solo uno de ellos recibirá el mensaje de finalización, por lo que el otro se quedará en deadlock.

Comenzaremos detallando la solución del segundo problema.

La misma consiste en no utilizar los ACKs automáticos que provee la biblioteca de rabbit, sino utilizarlos de forma manual. Cuando un worker lector del mom recibe un mensaje que no es una señal de finalización, enviará un ACK a rabbit, para indicarle que dicho mensaje ha sido recibido satisfactoriamente. En cambio, cuando detecte que el mensaje recibido corresponde a una señal de finalización, enviará un NACK a rabbit. Esta señal enviada a rabbit hará que el mensaje sea despachado a otro consumidor de la cola, el cuál repetirá la acción. Eventualmente, el mensaje llegará a todos los lectores de la cola.

En la sección “Deuda Técnica” se detalla otra posible solución y problemáticas experimentadas en relación con este tema.

Para la solución del primer problema, que consistía en definir una forma de cierre de canal cuando se tienen múltiples escritores, se decidió crear un administrador.

Administrador del Middleware

El administrador del middleware es un proceso que corre de forma independiente y cumple una simple pero crucial tarea. Deberá controlar el envío de la señal finalización de una cola. El mismo tiene comunicación con los procesos cliente de la mom a través de una cola de control tipo worker.

Como se detalló anteriormente, cuando un usuario crea una cola de escritura, se creará un worker traductor a nivel del mom. Al mismo tiempo que este worker es creado, el cliente del mom emitirá un mensaje “new,<queue_config>” al administrador.

El administrador mantendrá una estructura interna con todas las colas de escritura creadas y el número de escritores que cada una de ellas tiene. Entonces, al recibir el mensaje antes mencionado, creará una nueva entrada en dicha estructura o aumentará en uno el contador de escritores.

Cuando un worker traductor del cliente recibe la señal de finalización, enviará un mensaje “finish,<queue_name>” al administrador.

El administrador, al recibir este mensaje reducirá el contador de escritores de dicha entrada. Si el contador llega a cero, emitirá un mensaje por dicha cola (que recordemos tiene acceso porque se le ha enviado la configuración de inicio), con el mensaje “finish”.

Protocolo Implementado

Para la comunicación entre los procesos del sistema se utilizaron mensajes en formato texto. A excepción de la keyword reservada “finish”, todos los mensajes son recibidos y enviados sin modificación a los procesos usuarios de la mom.

Para la comunicación entre el cliente y el servidor se utilizó un protocolo binario similar al utilizado en el trabajo práctico 1.

Se detallarán brevemente las características:

- Cada mensaje enviado tiene un header de 4 bytes que indican el largo, en bytes, del cuerpo del mensaje.
- El cuerpo del mensaje tiene como primer byte un opcode que permite determinar el tipo de mensaje, de entre los cuales se encuentran:
    - Post
    - Comment
    - PostFinished
    - CommentFinished
    - Response
    - Error
- Cada mensaje tiene su codificacion particular, dependiendo de su composición. En forma general, cada mensaje se codifica como la composición de la codificación de sus tipos, los cuales son:
    - El OPCode antes mencionador, que se codifica como un byte sin necesidad de conversiones.
    - Un número se codifica simplemente como su representación bianria en BigEndian
    - Una string se codifica como su representación binaria en UTF8, anteponiendo un entero de 4 bytes que indica el largo del string.

Diagrama de Actividad

El siguiente diagrama de actividad muestra el flujo de la aplicación cuando se realiza el cómputo de puntaje promedio de los posts.

Es importante destacar que este diagrama no es completamente fiel a una ejecución verdadera, ya que falta el procesamiento de las demás tareas, correspondientes a el mejor meme por sentimiento promedio y las urls de memes de posts escolares. Se decidió simplificar el diagrama para que sea más sencillo de entender.

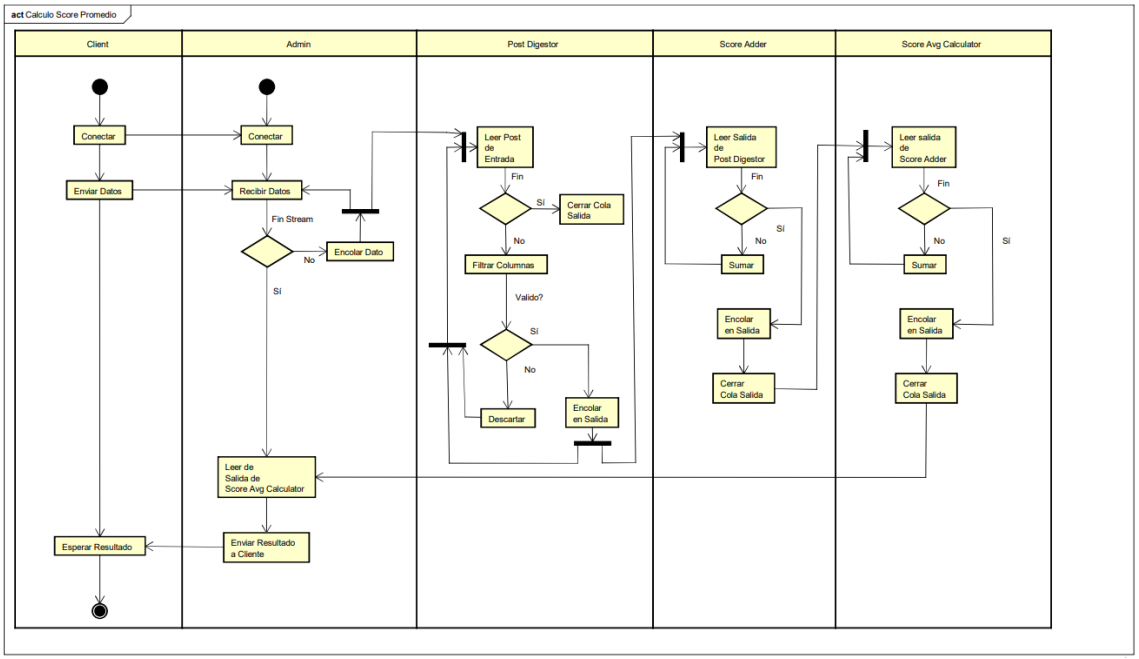


Diagrama de Paquetes

El siguiente diagrama muestra como se encuentra organizado el código del proyecto, en relación con sus módulos.

Es importante destacar que el paquete “worker_process” no existe como tal, sino que es una representación de todos los paquetes que representan a los procesos que realizan cómputo en el sistema, los cuales tienen las mismas dependencias entre sí.

Estos paquetes son:

- Post Score Adder
- Post Score Avg Calculator
- Post Sentiment Avg Calculator
- Joiner
- Comment Digestor
- Best Sentiment Downloader
- Post Above Avg Filter
- Post Digestor

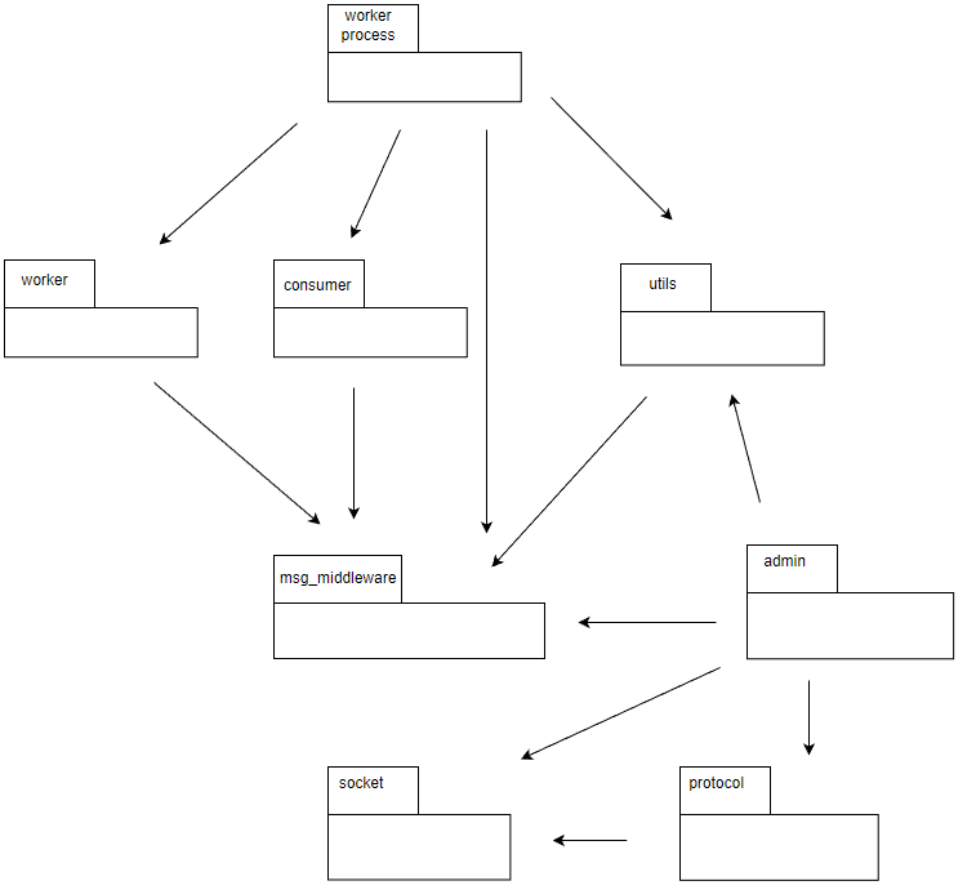


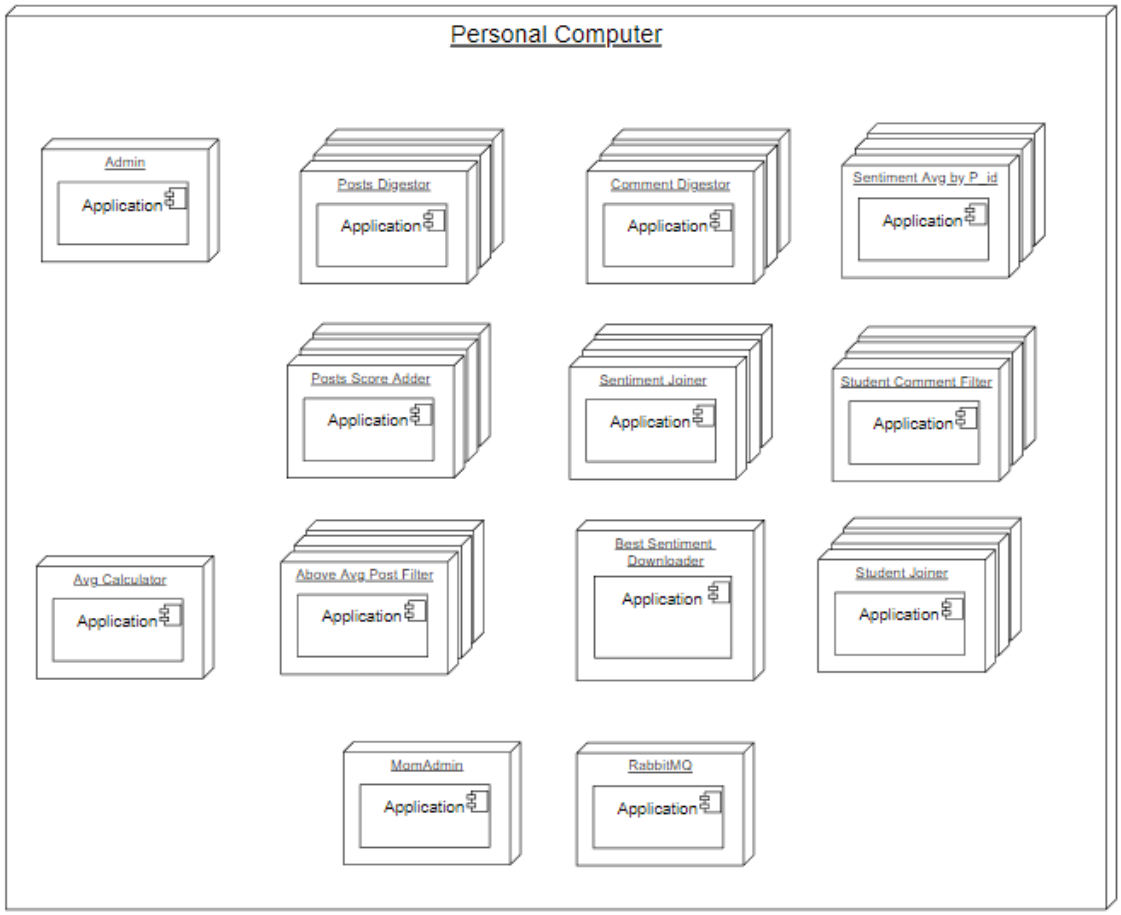
Diagrama de Despliegue

El siguiente diagrama muestra cómo es el despliegue de los procesos del sistema.

Aunque los procesos tienen capacidad de ser ejecutados en multicomputing, es decir, en diferentes computadoras, el despliegue utilizado a lo largo del desarrollo fue en una misma computadora personal.

En el diagrama se muestran entonces todos los procesos independientes que emulan correr en computadoras distintas, pero concentrados en una única computadora.

Se omitieron las conexiones entre componentes, pues sobrecargaban el diagrama y no aportaban información nueva a la que ya es visible en el diagrama de robustez.



Deuda Técnica

A continuación, se explicarán detalles a mejorar o faltantes de la entrega realizada. En su mayoría se considera que los mismos son debido a falta de experiencia en el uso de colas de mensajes y límites de tiempo.

Cálculo de Mejores Posts Escolares

Sin lugar a duda la deuda más importante es el hecho de que no se llegaron a cumplir las metas del trabajo. No se implementó el cálculo de los memes con mejor promedio, en ámbitos escolares (el tercer punto).

Sin embargo, se considera que, a menos que haya una falla grave no considerada en la arquitectura, la implementación de este cómputo debería ser sencilla, debido a que sólo sería necesario implementar el servicio “Student Comment Filter”, el cual es muy similar a filtros ya implementados. Adicionalmente, habría que realizar la recepción del resultado en el administrador y el retorno al cliente.

Detección de Cierre de Cola

El envío de una señal para indicar a los procesos la finalización de una etapa anterior, y particularmente el final de stream, resultó ser sumamente difícil.

En principio se considera que esto fue así debido, nuevamente, a la inexperiencia en el uso de este tipo de colas de mensaje.

Se comenzó implementando una arquitectura que al ser testeada tenía problemas de pérdidas de mensajes, ya que la base del algoritmo de detección era errónea.

En una sección anterior se comentó como se resolvió este problema, a partir del cierre de los canales de los usuarios del middleware.

Esta solución, sin embargo, puede que sea errónea. En la documentación de RabbitMQ para la función Nack se aclara que esta debería ser utilizada únicamente cuando el proceso lector no es capaz de procesar un nuevo mensaje. Por lo tanto, no debería ser utilizada para implementar una funcionalidad, como fue el caso.

Una implementación alternativa, que no pudo ser probada, consistía en la utilización de un canal adicional de control, a través del cual el administrador del middleware pudiese informar a un proceso lector (lector traductor, del middleware cliente) que la cola en cuestión había sido cerrada.

Luego, el proceso lector, para garantizar que todos los mensajes en camino (que pueden estar en red) hayan llegado, utilizaría la función “Cancel” de la biblioteca de RabbitMQ.

De lo poco que se leyó, se entiende que dicha función cierra un canal de lectura haciendo esperar al proceso lector a que todos los mensajes de la red emitidos hasta ese momento que tuviesen su destino llegasen. De esta forma se garantiza la no perdida de datos, y se evita el uso indebido de Nacks.

Desde ya, cualquier validación de alguna de estas alternativas, o el aporte de alguna nueva será bienvenido para trabajos futuros o la propia reentrega de este trabajo.

No se encontró en la biblioteca de RabbitMQ una forma sencilla de indicar, cual socket de TCP, que una cola era cerrada. Quizás esto es así debido al problema del cierre de una cola con múltiples escritores.

Mejorar Abstracciones

A pesar de haber logrado en cierta medida reducir el código necesario para implementar cada servicio, pues solo ha de implementarse la lógica de negocio y el setup inicial, igual se considera que hay lógica repetida en los mains que podría llegar a ser eliminada con una mejor abstracción.

Adicionalmente, los diagramas de paquetes muestran que las abstracciones realizadas en cuanto a dependencias dejan un poco que desear.

Prolijidad General

En relación a lo comentado en el ítem anterior, la prolijidad del código decayó en pos de buscar un trabajo funcional. Esto es claro cuando se ven partes de código comentadas, o con mensajes TODOs. Otro claro ejemplo es en el hecho de que el paquete “utils” y “worker” parecen cumplir funcionalidades similares, por lo que podrían llegar a unirse en uno solo.

Optimización en Mensajes

Se planeó realizar una optimización en el envío de mensajes desde el lado del middleware, como un cambio de baja prioridad. Actualmente se envían mensajes demasiado pequeños, los cuales pueden saturar el sistema de mensajes de Rabbit.

El plan era, desde el propio middleware, capturar mensajes y enviarlos en batch, teniendo cuidado de no mezclar mensajes con tópicos diferentes.

Escalabilidad del Sistema

El sistema, al menos en la computadora en la cual se desarrolló de 8gb de memoria, sólo es capaz de funcionar correctamente cuando la carga es baja.

Se realizaron pruebas utilizando únicamente la mitad del contenido de cada archivo y el consumo de memoria fue tal que rabbitmq comenzó a emitir alertas.

Se cree que la optimización en mensajes antes mencionada podría alivianar este problema.

Si el mismo persiste, habría que considerar optimizaciones más detallistas por parte de los procesos involucrados e incluso el uso de disco.

Filtrado de Comments

Como nota, se observó que la regex para extracción de post_id falla con los comentarios que tienen “me_irl” en lugar de “meirl” en su permalink. Se decidió no modificarla, aunque con más tiempo debería ser un cambio a realizar.