TP2: Middleware y Coordinación de Procesos

Facultad de Ingeniería, Universidad de Buenos Aires
[75.74] Sistemas Distribuidos I

Federico del Mazo - 100029

26 de mayo de 2022

Reddit Memes Analyzer – FdelMazo/7574-Distribuidos

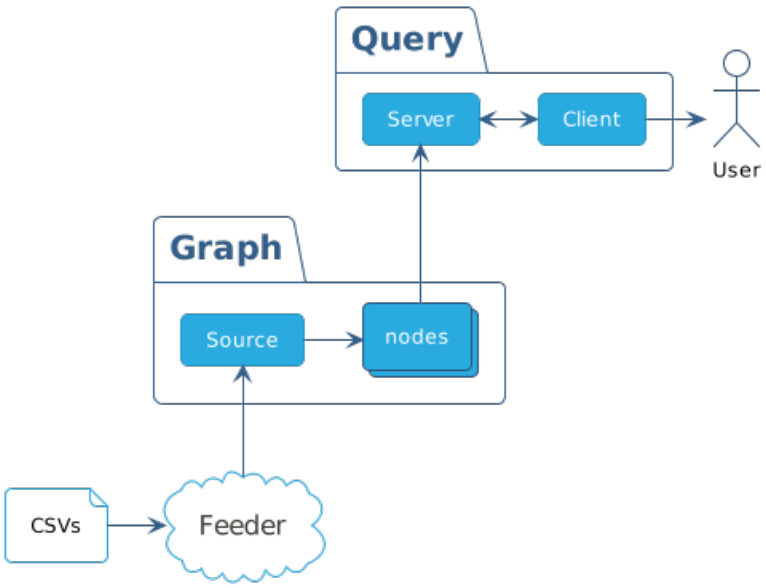
Introducción

El *Reddit Memes Analyzer* es un sistema distribuido preparado para analizar datos que provengan de Reddit. El resultado del análisis es en formato de **métricas**, las cuales son provistas al cliente del sistema en formato JSON.

Para simular la entrada de datos (posts y comentarios) y poder controlar el resultado, la entrada del sistema es en forma de un archivo `.csv` fijo para los posts y otro para los comentarios. Estos archivos son leídos y cada fila en ellos es enviada al sistema por un `Feeder`.

Los datos son luego procesados por un conjunto de nodos conectados entre sí, en forma de grafo acíclico. El grafo tiene un punto de entrada (el nodo `Source`) y un punto de salida (el `Collector`, el cual es un hilo dentro del servidor) donde se van depositando las métricas analizadas, y varios nodos en el medio que son los encargados de filtrar, reducir, agregar y operar sobre los datos que van entrando al sistema.

En cualquier momento de la ejecución, un cliente (`Client`) puede conectarse y hablar con el servidor (`Server`), el cual provee las métricas recolectadas hasta el momento del pedido.



Paquetes del sistema

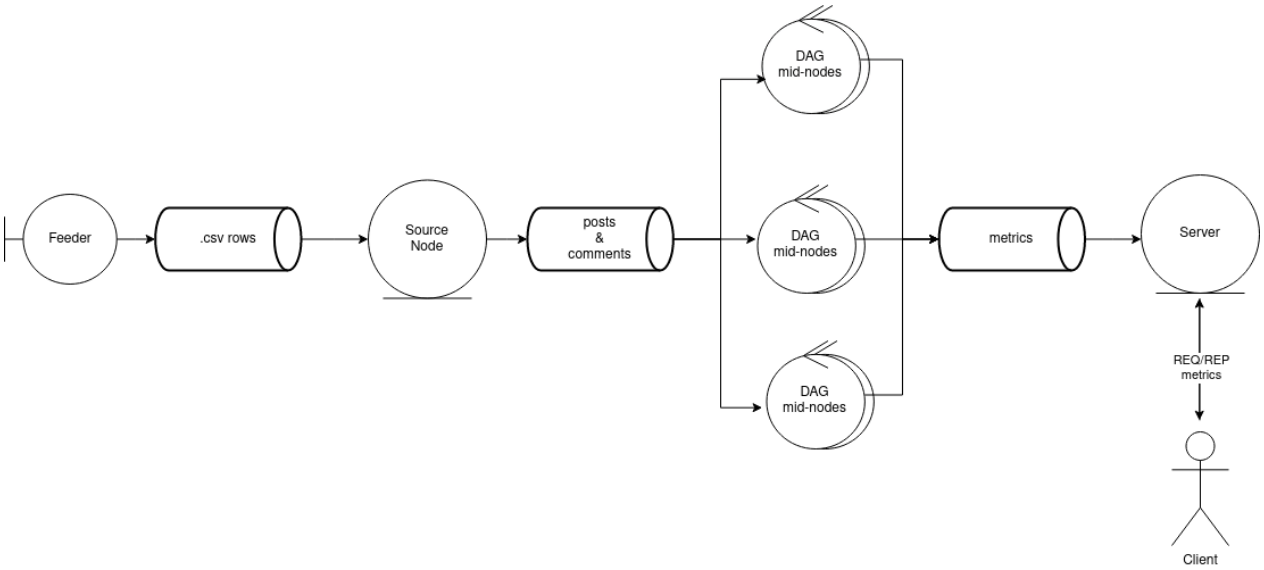
A nivel práctico, la manera más sencilla de ver esta disposición es ver el archivo `docker-compose-dev.yaml`, el cual contiene los nombres y perfiles de cada *container* de nuestro sistema, donde cada perfil puede ser levantado por separado. Ahí podemos ver que tenemos tres perfiles distintos:

- `graph`: El grafo acíclico que procesa los datos.
- `feeder`: El ingreso de los datos.
- `query`: La pregunta y respuesta de los datos.

La manera más sencilla que tenemos de levantar el sistema es utilizando `make up` seguido de `make logs`.

```
# Ejemplo donde el servidor nos devuelve una métrica: el score promedio de los posts
↪  procesados al momento
$ make up
$ make logs
client_1              | Got some stuff from the server!
client_1              | * post_score_average: 817.75

# De querer envíar una cantidad fija de datos, se puede utilizar la env var TEST_LINES
$ make run TEST_LINES=500 # make run es simplemente un alias de make up + make logs
feeder_1              | Finished sending 500 lines from
↪  ./data/the-reddit-irl-dataset-posts.csv and
↪  ./data/the-reddit-irl-dataset-comments.csv
client_1              | Got some stuff from the server!
client_1              | * post_score_average: 1551.19
```



Robustez

Feeder

El `Feeder` simplemente se encarga de leer los archivos `.csv` y enviar los datos fila a fila. Para simular un ambiente caótico, donde no sabemos qué tipo de dato llega antes, el `Feeder` lee tanto del archivo de posts como del de comentarios y le envía eso al nodo `Source`. Esto se hace a un *endpoint* único, en vez de a uno por cada tipo distinto de dato para que luego el grafo en sí sea el encargado de discriminar y organizar lo que le llegó, sin pre-procesamiento previo.

Luego de enviar los datos, el `Feeder` tiene que notificar que terminó la simulación de streaming, y envía el mensaje especial de end of file: `{'type': 'EOF'}`. Al recibir este mensaje, el sistema sabe que no tiene que

procesar cualquier dato nuevo que llegue, y que las métricas que contiene actualmente son las últimas.

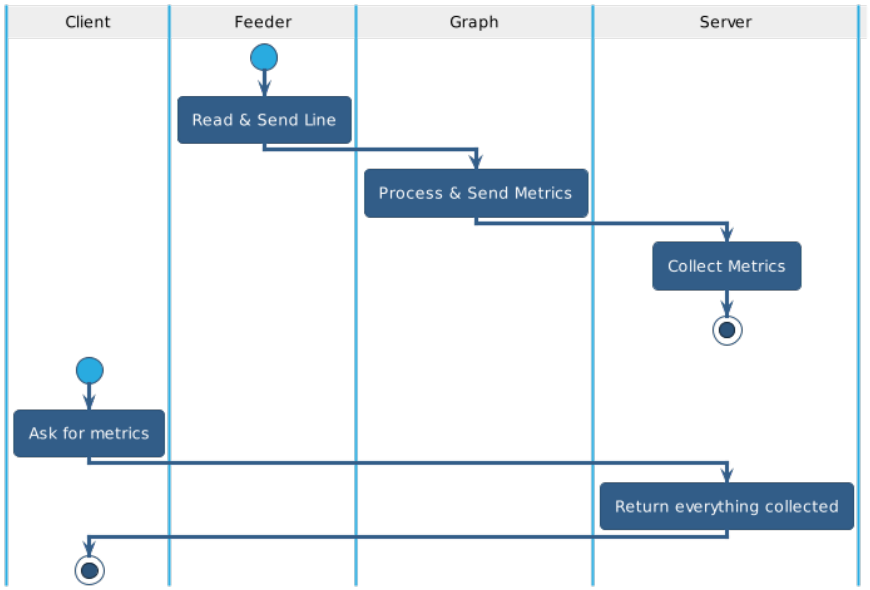
Cliente y Servidor

Del otro lado del sistema, alejándonos de como entraron y fueron procesados los datos, nos interesa saber como saldrán. Lo que tenemos es, en una disposición `REQ` y `REP` bastante inocente, un cliente que cada unos segundos le preguntará a un servidor las métricas que tiene hasta el momento. Es decir, nunca esperamos a asegurarnos de que los archivos `.csv` fueron enviados en su totalidad, ya que en cualquier momento del trayecto el sistema ya tiene datos que nos interesan (que más tarde serán pisados por datos actualizados).

El servidor por lo tanto tiene dos funcionalidades:

- Escucha *requests* del cliente, para saber cuando debe retornar métricas
- Recibe las métricas finales del grafo, para saber que enviar al cliente cuando se lo solicite

Es decir, en términos prácticos, el servidor también puede considerarse como parte del grafo, ya que actúa del nodo `Collector` (o `Sink`) de este.



Entrada de datos y pedido de métricas

Funcionalidades

Las funcionalidades del sistema son sencillamente las métricas que soporta, con la idea de que sea simple agregar métricas en un futuro. Con este objetivo en mente, en vez de tener un *endpoint* por cada métrica, simplemente tenemos un *endpoint* único, dedicado a traer todo lo que exista en el servidor.

Las métricas (actuales) son:

- El promedio de *score* de todos los posts procesados
- La imagen más popular de acuerdo al sentimiento de los comentarios
- Las URLs de posts relacionados a contenido de estudiantes y universidades, basándonos en el contenido de sus comentarios.

Como podemos ver, se pueden dividir en dos tipos distintos de métricas: las que son cadenas planas (el promedio y las URLs), y las que son contenido en bytes (la imagen).

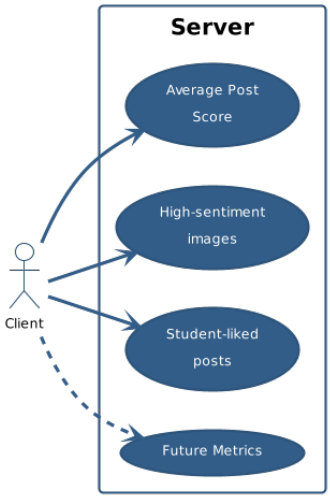
Es por esto que hay que armar un (muy) pequeño protocolo de cómo el cliente recibe las métricas, de manera tal que puedan ser genéricas y no haga falta agregar código específico cada vez que introduzcamos métricas

nuevas.

```
{
    [nombre] <string>: {
        'metric_value': [valor] <string>
        'metric_encoded': [indicador] <booleano opcional>
        'metric_final': [indicador] <booleano opcional>
    }
}
```

La clave `'metric_encoded'` es opcional e indica que `'metric_value'` contiene una cadena codificada en base 64. De suceder esto, el cliente recibe esa cadena, la decodifica y la guarda en un archivo. En caso de la métrica sea una cadena plana, simplemente se imprime por pantalla.

La clave `'metric_final'` indica que el servidor termino de procesar las métricas y no aceptará nuevos datos, avisando que el cliente ya no va a recibir nuevos valores (y puede cerrarse, ya que no tiene propósito de seguir corriendo).

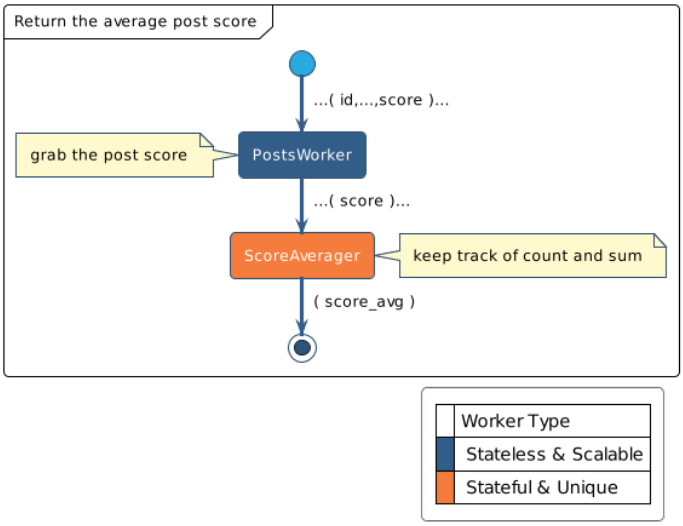


Posibles casos de uso

Las métricas son las que darán forma a nuestro grafo. Por separado, cada métrica tiene su propia manera de ser procesada, pero lo que nos interesa es encontrar puntos en común y lugares de optimización, para que el sistema pueda trabajar en conjunto en vez de ser solamente N sistemas aislados que procesan métricas por su cuenta.

Promedio de score de posts

La métrica más sencilla de procesar es el promedio de *score* de todos los posts procesados. Consiste solamente de recibir un post, tomar su *score* y promediarlo con el resto. Buscando que cada nodo trabaje lo menos posible y pase al siguiente trabajo, usamos la manera más liviana de contar un promedio: mantener una cantidad y una suma. Es decir, con la cantidad de datos que estamos manejando, nos queremos ahorrar llamados como `sum(<list>)` que son más costosos.



Métrica 1: Score Promedio

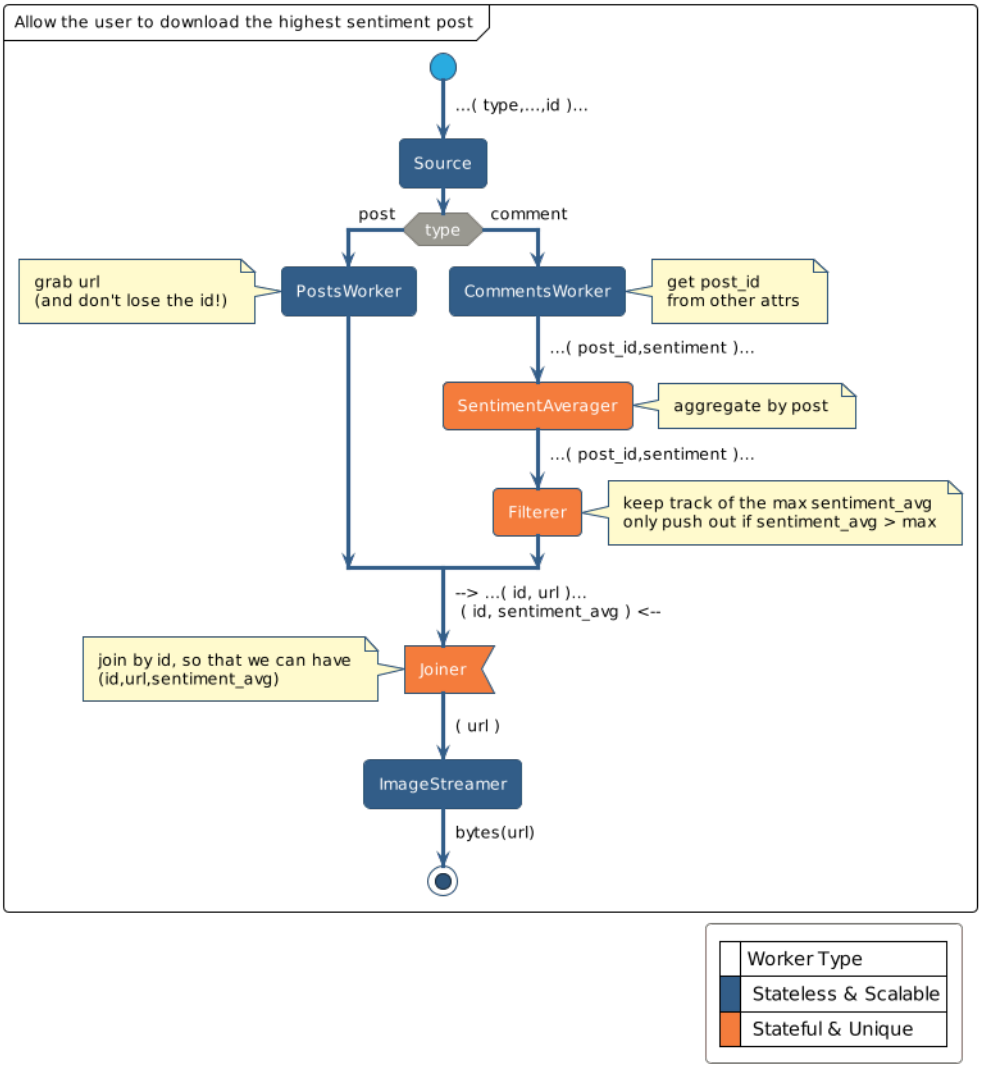
Lo que podemos ver con este grafo de solo dos nodos es que tenemos dos tipos distintos de nodos: los que deben guardar estado y los que no.

Los nodos que no deban llevar la cuenta de nada y sean *stateless*, son los que podremos escalar en nuestro sistema distribuido. Si yo me diese cuenta que tengo que procesar 5000 posts por cada 1 comentario que ingresa en mi sistema final, entonces dedicaría más recursos a `PostsWorker` que al nodo que se encargue de trabajar en comentarios. Para lograr esto tenemos que tener una disposición de la red que nos permita que las N instancias de mi nodo escalable trabajen en paralelo, agarrando trabajos disponibles desde la misma cola de tareas.

Por otro lado, los nodos que sí tengan estado (llevar cuenta de un promedio, mantener en memoria algún atributo en particular, etc) son los que deben ser únicos, y así ahorrarnos los problemas de sincronización, usando la filosofía de ZeroMQ de "Share Nothing".

Meme con mayor sentimiento de comentarios promedio

La segunda métrica a devolver es una imagen de un post, basado en metadata que proviene de sus comentarios. Es acá donde nos cruzamos con nodos más complejos, en particular el `Joiner`. Lo que precisamos es un nodo que pueda recibir información de distintos lugares y unirla antes de seguir con el resto del grafo.

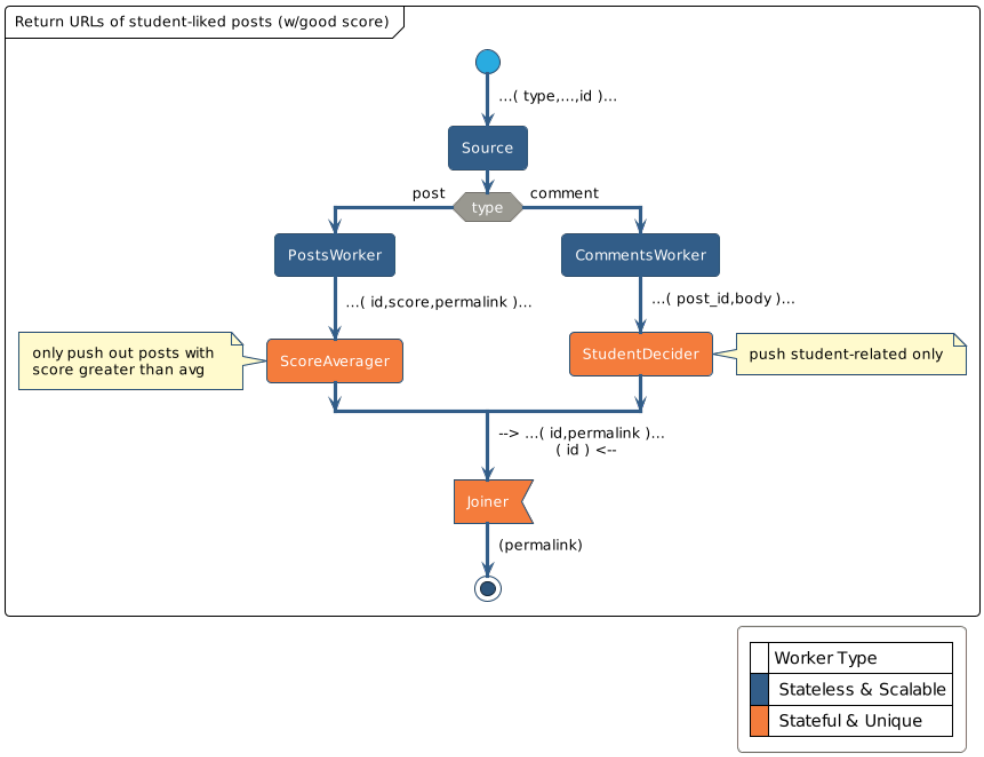


Métrica 2: Meme con mayor sentimiento de comentarios promedio

Al introducir el nodo `Joiner` y el nodo `Filterer`, empezamos a encontrarnos con decisiones a tomar en cuanto a la disposición de los nodos. ¿Primero filtro, y luego uno, o primero uno y luego filtro? Esta pregunta nos la podemos hacer porque al tener juntos todos los datos por los que tengo que filtrar, no me veo forzado a hacer ninguna operación antes que la otra. Ya que podemos elegir, lo ideal sería que el `Joiner`, que hasta ahora es el nodo que más almacena en memoria, reciba la menor cantidad de datos posibles, por lo que tiene sentido que el `Filterer` venga antes.

Posts que más gustan a estudiantes

La última métrica a devolver es bastante similar a la anterior, juntando datos tanto de los posts como de los comentarios. Lo que incluimos acá es que tenemos dos filtros. Por un lado, queremos filtrar los posts que incluyen contenido de estudiantes, y por el otro, queremos filtrar los posts con score mayor al promedio.



Métrica 3: Posts que más gustan a estudiantes

En este caso, para no tener una lista larga de posts a devolver, en cada iteración que me pidan la métrica devolvemos un *sample* al azar de N posts, en vez de la lista entera.

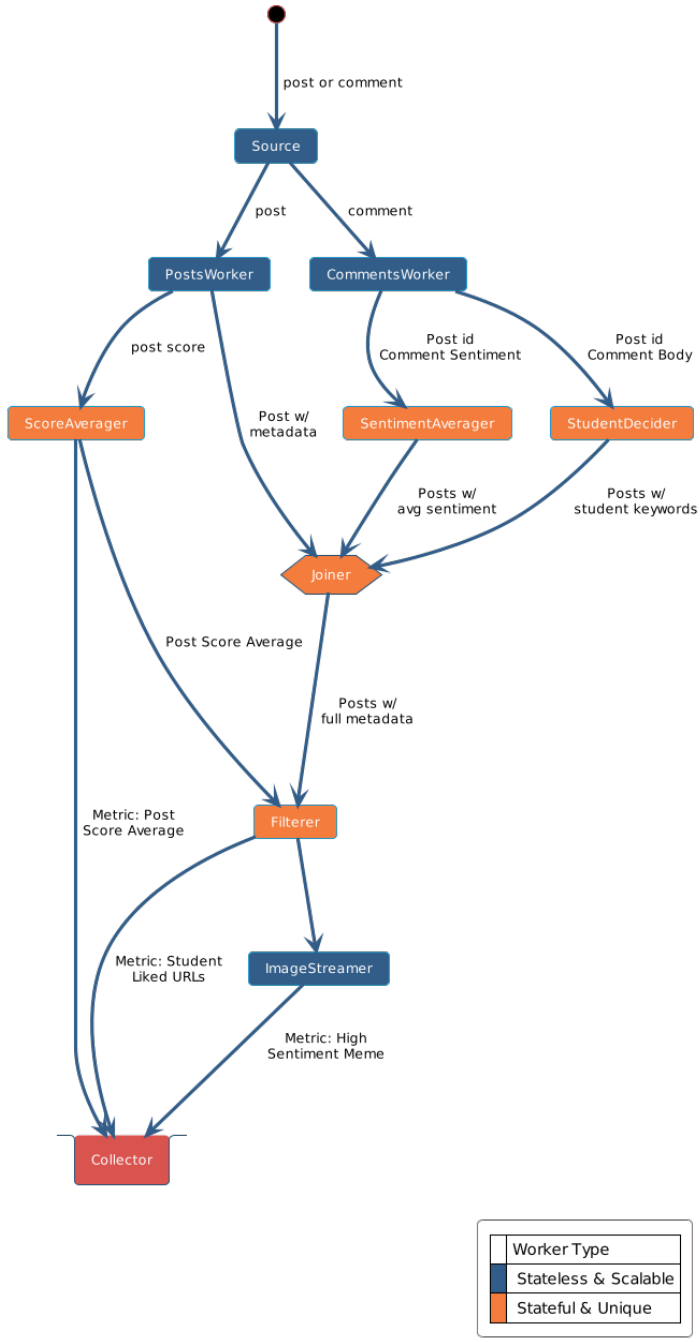
Grafo

Habiendo visto cada métrica por separado, es ahora que podemos diseñar un grafo que contenga las tres funcionalidades trabajando en conjunto, ya que datos de una le sirven a otra.

La suma de estas métricas separadas no es trivial, ya que lo que en una es una ventaja, en la siguiente puede ser una desventaja. Si bien se podrían tener un nodo por cada una de todas las tareas que vimos hasta ahora, todo por separado, también queremos lograr una unicidad semántica y funcional, para lograr que agregar métricas en el futuro no sea extremadamente complejo.

Las cosas que podemos notar de los diseños son:

- Necesitamos un colector/sumidero que este todo el tiempo disponible para entregarle una métrica (o, alternativamente, tener un sumidero por cada métrica).
- Podemos intentar unir nodos acorde a los datos que procesan (ej: un trabajador del atributo sentimiento) como a las operaciones que aplican (ej: un filtrador genérico, sin importar si recibe posts o comentarios).
- El diseño original de las métricas por separado puede cambiar cuando unamos todo, ya que habrá *trade-offs* involucrados en la estructura del grafo final.



Grafo Final

Los nodos finales son:

- `Source`: Recibe datos indiscriminados, y los envía al trabajador correspondiente de acuerdo al tipo de dato.

  Podría ser escalable ya que no tiene estado, aunque no es aconsejable, ya que tenemos un control más granular al escalar los workers dedicados a posts y comentarios por separado.

Tiene una conexión directa con el `Collector`, para avisarle que recibió un mensaje de `EOF` y que debe marcar a las métricas que tiene como finales.

- `PostsWorker` y `CommentsWorker`: Reciben un post o comentario y envían a los siguientes trabajadores exactamente los atributos que necesitan. También pueden aplicar algún pre-procesamiento necesario a los atributos que lo necesiten, o filtrar los datos que consideremos inválidos para todo el resto del sistema.

  Estos dos nodos funcionan como un despachador que minimiza el payload (al no enviar los atributos que nos sobran) con el que trabajará el resto del grafo, así logrando transportar la menor cantidad de información posible en la red.

  De necesitar más poder de cómputo para procesar posts o comentarios, estos nodos se pueden replicar ya que sus instancias reciben trabajo de la misma cola.

- `ScoreAverager` y `SentimentAverager`: Estos dos nodos mantienen registro de los distintos datos que se van recibiendo, y calculan el promedio según atributos. Si bien ambos calculan promedios y a simple vista podrían ser abstraídos al mismo nodo, el promediado de *scores* recibe posts, mientras que el de *sentiment* recibe comentarios, por lo que cumplen distintas funciones dentro del sistema.

- `ImageStreamer`: Este nodo simplemente recibe una URL, intenta descargar su contenido y envuelve sus bytes en un string para poder enviar la métrica al colector.

- `StudentDecider`: Este nodo recibe comentarios y determina si un post es relevante de acuerdo al contenido de cada comentario, y luego solo empuja los posts relevantes.

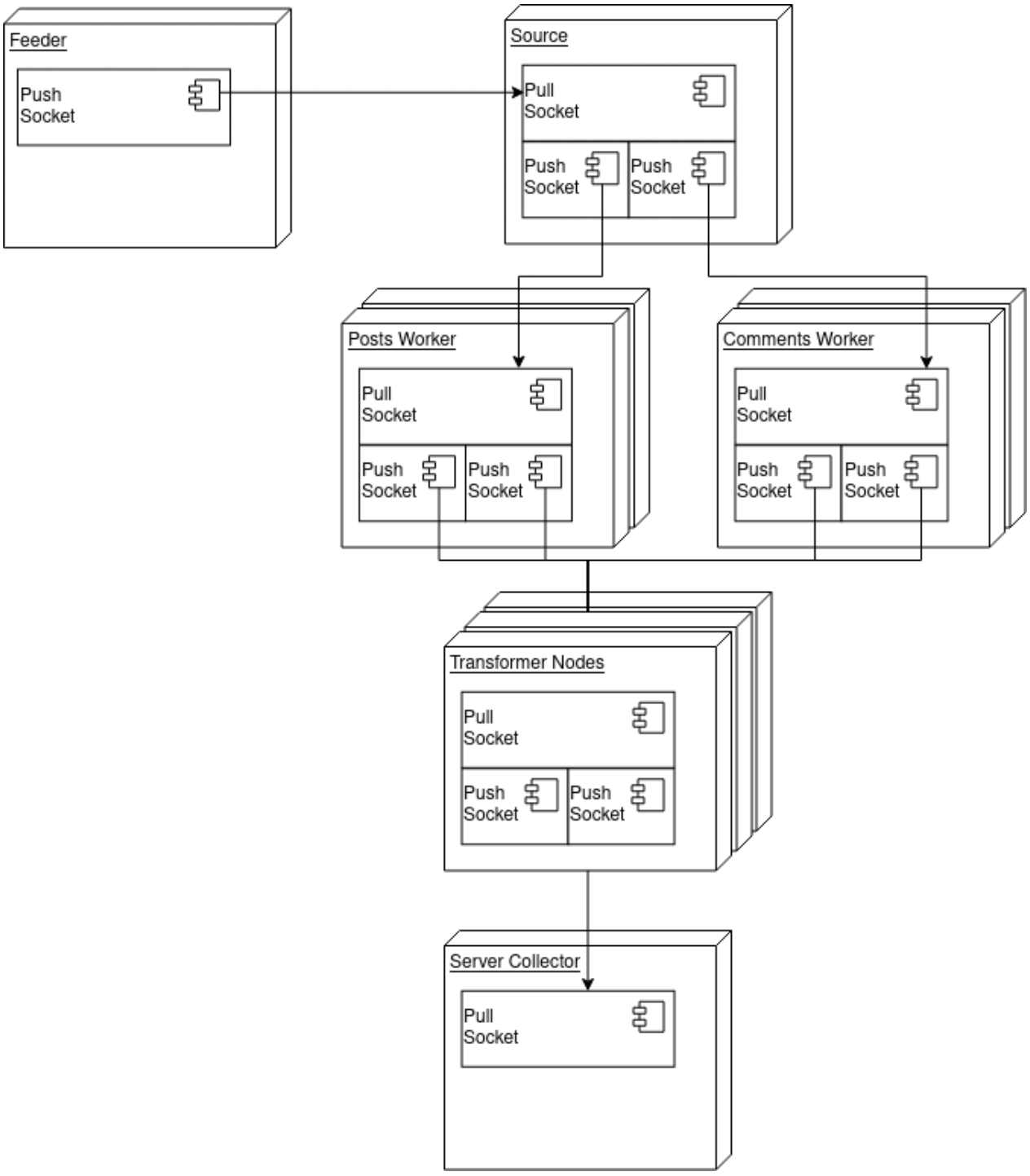
  Una decisión que se toma acá es la de hacer este nodo *stateful*. Si bien se podría recibir cualquier comentario y empujarlo solamente si contiene las palabras que nos interesan, preferimos reducir el tráfico de la red y solo empujar los posts que no nos hayamos cruzado previamente, evitando envíar duplicados. El trade-off es que este nodo no puede ser escalable. Alternativamente, aún siendo *stateful* se podría replicar y hacer que cada réplica mantenga su propia lista de posts, haciendo que se reduzca el envío de datos duplicados, sin mitigarse del todo, pero así perdemos nuestro principio de que solo queremos replicar nodos *stateless* (lo cual es una decisión, no necesariamente una restricción).

- `Joiner` y `Filterer`: Estos dos nodos son los más importantes del grafo, y son los que se encargan de hacer el unido y filtrado. Lo que se decide es que el `Joiner` puede a todo momento recibir un post con distintos atributos, desde cualquier nodo del sistema, y actualiza su estado interno para mantener registro de los nuevos atributos recibidos. Luego, se le pasa esto al filtrador, que está cargado de lógica para decidir en base a los distintos atributos cuales deben seguir su rumbo para proveer las distintas métricas.

El nodo final del sistema es el `Collector`, que se ejecuta en el servidor. Simplemente recibe métricas que le puede pasar cualquier nodo en cualquier momento de la ejecución, y las guarda para enviarlas al cliente cuando las solicite.

Siendo que cada nodo (sin contar el `Collector`) actúa de manera bastante parecida (levantar un dato de una cola, procesarlo, y empujarlo a la siguiente), se abstrae esta funcionalidad y se crea la clase `BaseNode()` de la cual los nodos heredarán su comportamiento, levantando datos de un socket `zmq.PULL` y empujando a un socket `zmq.PUSH`.

De querer ver todo el tráfico de mensajes que atraviesa nuestro grafo, podemos configurar el nivel de logueo en el archivo `config.ini` para que sea `DEBUG`



Despliegue del DAG

```
$ make run TEST_LINES=3
feeder_1              | Finished sending 3 lines from
↪  ./data/the-reddit-irl-dataset-posts.csv and
↪  ./data/the-reddit-irl-dataset-comments.csv
source_1              | {'type': 'post', 'id': 'tswh3j', 'subreddit.id': '2s5ti',
↪  'subreddit.name': 'meirl', 'subreddit.nsfw': 'false', 'created_utc': '1648771149',
↪  'permalink': 'https://old.reddit.com/r/meirl/comments/ttdab1/meirl/', 'domain':
↪  'i.imgur.com', 'url': 'https://i.imgur.com/ucZiw34.jpg', 'selftext': '', 'title':
↪  'meirl', 'score': '93'}
```

```
source_1              | {'type': 'comment', 'id': 'i2x2j0g', 'subreddit.id': '2s5ti',
↪  'subreddit.name': 'meirl', 'subreddit.nsfw': 'false', 'created_utc': '1648771186',
↪  'permalink': 'https://old.reddit.com/r/meirl/comments/tswh3j/meirl/i2x2j0g/',
↪  'body': 'Yes', 'sentiment': '', 'score': '1'}
source_1              | {'type': 'post', 'id': 'ttd0r4', 'subreddit.id': '2s5ti',
↪  'subreddit.name': 'meirl', 'subreddit.nsfw': 'false', 'created_utc': '1648770296',
↪  'permalink': 'https://old.reddit.com/r/meirl/comments/ttd0r4/me_irl/', 'domain':
↪  'i.redd.it', 'url': 'https://i.redd.it/x8sehlq23tq81.jpg', 'selftext': '', 'title':
↪  'Me_irl', 'score': '1'}
posts_worker_1        | {'type': 'post', 'id': 'tswh3j', 'subreddit.id': '2s5ti',
↪  'subreddit.name': 'meirl', 'subreddit.nsfw': 'false', 'created_utc': '1648771149',
↪  'permalink': 'https://old.reddit.com/r/meirl/comments/ttdab1/meirl/', 'domain':
↪  'i.imgur.com', 'url': 'https://i.imgur.com/ucZiw34.jpg', 'selftext': '', 'title':
↪  'meirl', 'score': '93'}
posts_worker_1        | {'type': 'post', 'id': 'ttd0r4', 'subreddit.id': '2s5ti',
↪  'subreddit.name': 'meirl', 'subreddit.nsfw': 'false', 'created_utc': '1648770296',
↪  'permalink': 'https://old.reddit.com/r/meirl/comments/ttd0r4/me_irl/', 'domain':
↪  'i.redd.it', 'url': 'https://i.redd.it/x8sehlq23tq81.jpg', 'selftext': '', 'title':
↪  'Me_irl', 'score': '1'}
comments_worker_1     | {'type': 'comment', 'id': 'i2x2j0g', 'subreddit.id': '2s5ti',
↪  'subreddit.name': 'meirl', 'subreddit.nsfw': 'false', 'created_utc': '1648771186',
↪  'permalink': 'https://old.reddit.com/r/meirl/comments/tswh3j/meirl/i2x2j0g/',
↪  'body': 'Yes', 'sentiment': '', 'score': '1'}
# ... etcetera
```

Finalización

El sistema de nodos no finaliza nunca per se, (obviando que se le envíe un `SIGTERM`). Esto es porque está originalmente pensado para procesar constantemente datos en streaming, y teniendo que mantener métricas constantemente en memoria para poder ir actualizándolas (o, en una mejora del programa, persistirlas de alguna forma).

Siguiendo esta linea, el servidor tampoco nunca finaliza, dejándolo prendido por si algún otro cliente se le quisiese conectar y pedir métricas (aunque habría que implementar una funcionalidad de descongelamiento de las métricas, ya que una vez que terminó de procesar todo el dataset, las métricas quedarán marcadas como finalizadas). De todas formas, finalizarlo es trivial, nada mas terminamos el *handshake* del cliente-servidor: una vez que el cliente solicita las métricas y recibe que están finalizadas, justo antes de cerrarse, le envía un comando de `/shutdown` al servidor avisándole que puede finalizar su ejecución.

Ahora bien... ¿cómo se podría implementar la finalización del grafo, una vez recibido el `EOF` del dataset?

Originalmente pensé que una vez que el nodo `Source` recibe un `EOF`, puede replicar el mensaje a todas sus colas de salida, para que los nodos que lean de ahí puedan tomar ese mensaje y replicarlo nuevamente y así ir lentamente propagando el mensaje de `EOF` para que todos los nodos del sistema vayan finalizando su ejecución.

El problema con esto es que con la implementación de `PUSH`/`PULL` actual, de tener algún nodo replicado, solo una de las N replicas recibirá el mensaje de finalización, dejando nodos levantados colgados, sin nunca finalizar.

La manera de mitigar este problema es utilizando un patrón de `PUB`/`SUB` específico para este tipo de mensajes, ya que el `PUB` si permite hacer un *broadcast* a todos sus lectores, a diferencia del `PUSH`.

Otra idea a considerar, pero se siente una implementación bastante frágil, es ponerle a cada nodo un *timeout* interno que diga que si en N segundos no recibieron un mensaje, pueden finalizar su ejecución. Esto no es conveniente porque no hay ninguna acción afirmativa que llame al finalizado, haciendo que esto no sea controlado (a diferencia de la idea anterior, donde se debe mandar un mensaje de `EOF` que comience la cascada de terminaciones).