Trabajo Práctico 1
Metrics && Alerts Server

Sistemas Distribuidos I (75.74)

1er Cuatrimestre 2022
Facultad de Ingeniería
Universidad de Buenos Aires

Mauro Parafati
102749
mparafati at fi.uba.ar

Monorepositorio de TPs

Índice

1. Introducción

El presente trabajo práctico consiste en el desarrollo de un **sistema distribuido** que brinda servicios para el **reporte de eventos** de aplicación, **consulta de métricas** y **disparo de alertas**. Este sistema estará pensado para ser instalado dentro de una empresa que posee un gran ecosistema de aplicaciones y usuarios que monitorean los servicios.

Los temas abarcados por el mismo son el manejo de la **concurrencia** (se debe construir un sistema que sea escalable, pero siempre cuidando la consistencia y evitando las condiciones de carrera en los puntos de datos compartidos, como en este caso sería la base de datos) y las **comunicaciones** (construir protocolos sencillos pero eficientes para comunicar los distintos componentes).

2. Diseño de la arquitectura

El primer punto sobre el que se trabajó, fue en el diseño de la arquitectura del sistema. Esta misma se formaría mediante numerosas iteraciones de diseño evaluación y mejora, que se detallan a grandes rasgos en las siguientes secciones.

2.1. Identificación de componentes

Para comenzar, se intentó identificar componentes individuales encapsulando comportamiento. En esta etapa, se transformaron los requerimientos funcionales en entidades que puedan cumplir con los mismos. Se identificaron los siguientes componentes:

- **Event Writer:** entidad responsable de recibir eventos de las distintas aplicaciones internas de la empresa, y de su almacenamiento coordinado en la base de datos compartida.
- **Aggregator:** entidad que atiende consultas de los usuarios, leyendo la información de la base de datos y agregándola según el criterio requerido.
- **Alert Service:** entidad encargada de realizar chequeos automáticos cada cierto tiempo, disparando alertas de ser necesario.
- **Database:** estructura compartida donde se almacenarán los datos. Su diseño tendrá un impacto arquitectural muy importante sobre nuestro sistema, debido a que es un componente accedido por distintos procesos en todo momento.

Con estos componentes pre-armados, se prosiguió a realizar una primera vista lógica de nuestra arquitectura.

2.2. Primera vista lógica

En esta primera vista lógica, se buscó conectar a los mencionados componentes entre sí, dando lugar a un sistema que cumpla con los flujos esperados. Se obtuvo el siguiente diagrama:

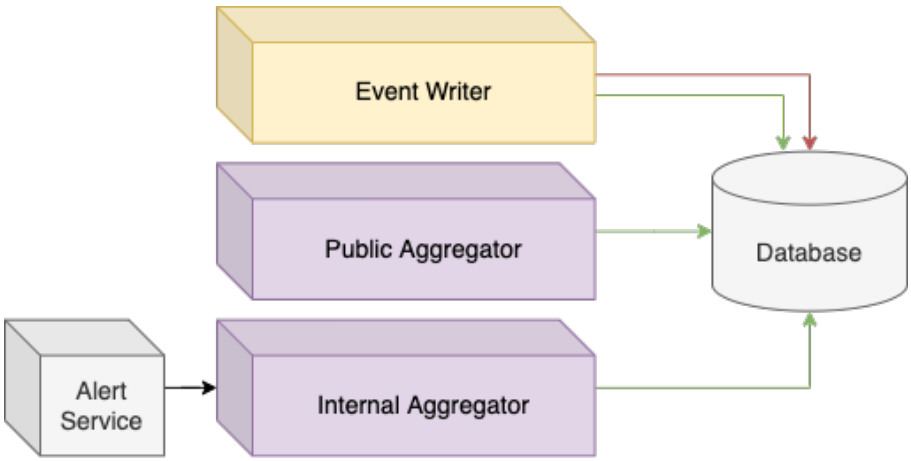

Figura 1: Vista lógica de entidades

En el que ya podemos identificar en qué puntos deberemos diseñar protocolos y para comuni-

car a qué entidades. También podemos verificar que todos los requerimientos funcionales pueden cumplirse con una arquitectura de este estilo.

2.3. Arquitectura de cada componente

Una vez que se tuvo la idea general de la arquitectura, se diseñó cada componente, siguiendo el paradigma de concurrencia orientado a mensajes, en el que es muy común el patrón de uso de un "Thread Pool" compuesto por distintos workers que pueden atender consultas. Se detallará este paradigma utilizado en una posterior sección.

Los componentes diseñados resultaron de la siguiente forma:

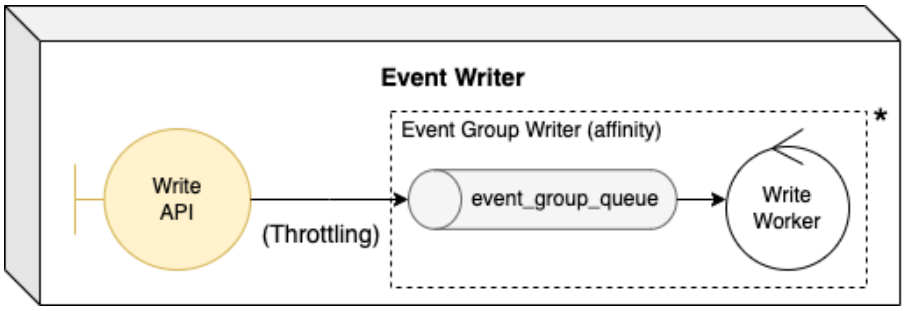

Figura 2: Vista lógica del componente **Event Writer**

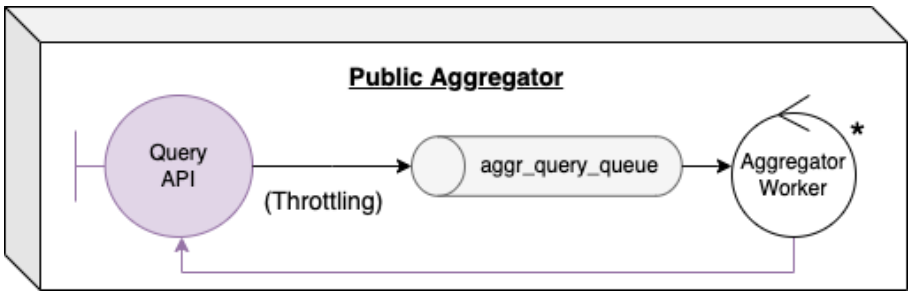

Figura 3: Vista lógica del componente (genérico) **Aggregator**

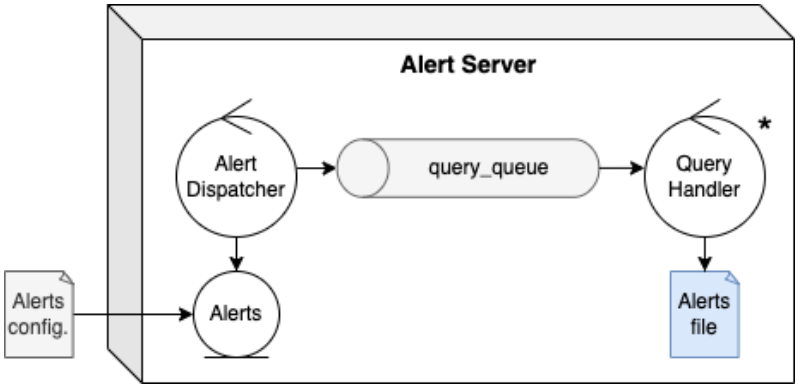

Figura 4: Vista lógica del **Alert Service**

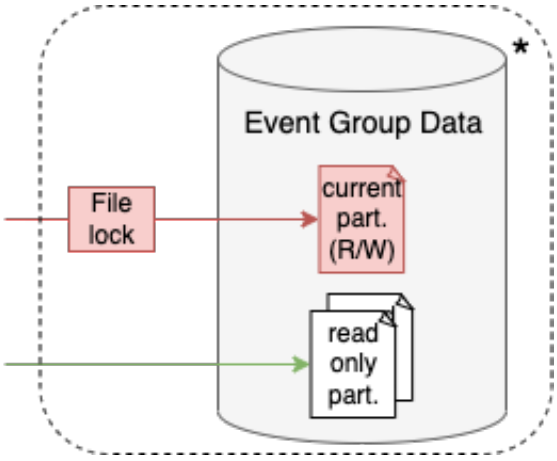

Figura 5: Vista lógica de la **Database**

2.4. Arquitectura final

Combinando la primer vista lógica que conectaba nuestros componentes, con el desarrollo de cada uno de ellos, podemos realizar un diagrama de robustez final que nos de un pantallazo general de todo el sistema:

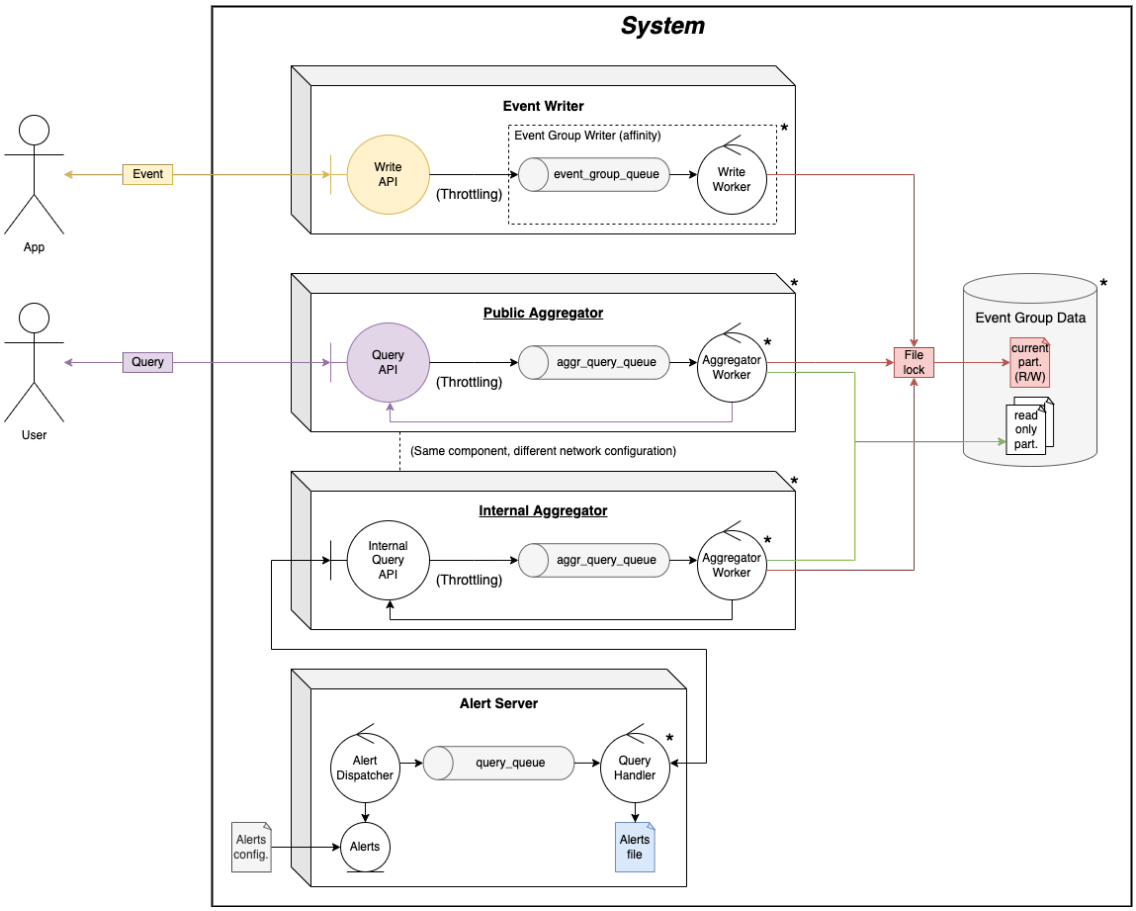

Figura 6: Diagrama de Robustez

2.5. Análisis de la arquitectura resultado

La construcción de una arquitectura no debe ser el paso final de la etapa de diseño, ya que es importante realizar un análisis posterior de la misma con el objetivo de identificar ventajas y desventajas de lo que se propuso, así como determinar si es necesario realizar alguna modificación.

2.5.1. Ventajas

- **Componentes simples de entender.** A primera vista puede entenderse qué hace cada componente de nuestra arquitectura.
- **Interfaces separadas para clientes, aplicaciones, y sistema interno de métricas.** Este punto es fundamental ya que evita que tres casos de uso distintos que tienen una

frecuencia muy diferente compartan el mismo canal de comunicación, pudiendo llevar a casos en que si tenemos muchos eventos y en comparación no tantas queries, tal vez ninguna query sería aceptada po el throttling.

- **Patrón cola-workers permite escalar horizontalmente sin límites.** Los puntos fuertes de procesamiento de nuestro sistema son los workers de los thread pools, y gracias al patrón implementado podemos instanciar tantos workers como queremos para adaptarnos a la demanda.
- **Arquitectura compatible con multi-computing.** El **Aggregator** y el **Alert Service** pueden desplegarse en tantas computadoras como se quiera. El caso del **Event Writer** es más delicado, ya que estaríamos **rompiendo con la affinity** definida para los workers.
- **Lock free** (salvo en la base de datos). Favorece la concurrencia, facilita el desarrollo de código.

2.5.2. Desventajas

- **Affinity entre workers y métricas** presenta una clara ventaja por la que fue implementado (evitar ciertos problemas de sincronización), pero significa una desventaja en cuanto a arquitectura, ya que si una métrica tiene muchos más eventos que las demás, tendremos a un worker desproporcionadamente saturado frente al resto, que estarán IDLE la gran parte del tiempo.
- **Soporte para hasta N consultas paralelas.** La arquitectura implementación de los aggregators es muy simple, generando como desventaja que no podamos manejar más de N consultas en paralelo, siendo N la cantidad de workers que tenemos.

3. Manejo de la concurrencia

En esta sección se discutirá brevemente el enfoque tomado para resolver la concurrencia de una manera sólida dentro del sistema.

3.1. Paradigma planteado

El paradigma planteado en líneas generales se trata del **pasaje de mensajes** y del uso de patrones conocidos como el ya mencionado "thread pool", en el que una serie de *workers* independientes entre sí toman tareas a realizar de una cola compartida.

Este paradigma presenta ventajas muy claras:

- **Escalabilidad.** Los mensajes, siempre y cuando estén bien diseñados, son livianos, y podemos tener miles y miles de estos siendo enviados entre threads (sean green threads o del sistema operativo), entre procesos, y hasta entre distintas computadoras. Un mensaje posee la ventaja de se trata de una simple estructura en memoria, contra lo que podría significar manejar cada solicitud con un hilo que se encargue exclusivamente de la misma.
- **Control.** Desde un principio podemos prefijar distintos valores que nos aseguran tener control sobre los recursos que utiliza nuestro sistema. Por ejemplo, podemos definir tamaños máximos para las colas de mensajes, cantidad de workers por cada servicio, etc.
- **Adaptabilidad.** Atado al punto anterior, podemos recolectar información y métricas sobre el consumo de recursos de cada componente, para luego hacer un trabajo iterativo de reorganización de los mismos buscando optimizar el desempeño del sistema en su conjunto. Esto podría hacerse o bien estática o bien dinámicamente, teniendo un componente responsable de tal tarea.
- **No hay estado compartido, no hay locks.** El único estado compartido que tenemos está presente en la implementación de las colas de mensajes para los thread pools, implementación que puede ser realizada una sola vez en forma de abstracción y reutilizada en distintos componentes. Gracias a esto, podemos diseñar una solución **lock-free** que maximice la concurrencia posible.

3.2. Identificando puntos de conflicto

Gracias al paradigma elegido, los puntos de conflicto en el trabajo son realmente pocos, y todos se centran en el mismo componente: **la base de datos**. Esto se debe a que la base de datos es el único elemento compartido accedido por varios otros, en la que algunos procesos intentarán escribir y otros leer. Para esto es necesario proteger esta estructura de alguna forma.

3.3. Solución propuesta (particionamiento de archivos)

Para solucionar los conflictos existentes en la lectura/escritura de la base de datos, se pueden tomar múltiples decisiones, cada una de ellas influyendo de forma directa en la performance final de nuestro sistema.

Un primer enfoque podría ser utilizar un **lock global en la base de datos**, para que solo un proceso pueda abrir archivos en un momento dado. Esto tiene la ventaja de que resuelve cualqueir tipo de problema de concurrencia, pero a costa de que **elimina la concurrencia por completo**.

No es una opción viable pero sirve para empezar a analizar el problema e iterar sobre esta idea. El objetivo es proteger el estado compartido, siendo lo más atómico que se pueda ser.

Se piensa ahora en que hay distintas métricas y que las operaciones se centran **en una sola métrica**. Este punto es clave para entender que se podrían **particionar los archivos por métrica**, pudiendo pasar de un gran lock global a un lock por métrica, permitiendo entonces ahora sí consultas y escrituras concurrentes siempre y cuando sean a distintas métricas. Este escenario ya es mucho más favorable, pero podemos seguir mejorándolo aun más.

Se decide entonces **particionar también por fecha**, teniendo múltiples archivos para una misma métrica, y lockeando cada archivo. Ahora no sólo soportamos lecturas y escrituras concurrentes entre distintas métricas, sino también dentro de la misma siempre y cuando sean en tiempos distintos.

Finalmente, se hace la pregunta: ¿es realmente necesario lockear todos los accesos? Si se considera que el único archivo que va a ser escrito y leído al mismo tiempo es el que representa a la partición más actual, rápidamente se puede pensar en que existen **dos tipos de particiones: de solo lectura, y de lectura/escritura**. Sólo existirá una partición de lectura/escritura por métrica en un momento dado, por lo que **sólo se necesita lockear un archivo por métrica**.

Esta es la solución final, que permite que tengamos **cualquier cantidad de lecturas en paralelo**, sólo bloqueando en el archivo de escritura actual.

4. Detalles de implementación

Se detallan brevemente a continuación algunos detalles de implementación que podrían resultar de interés.

4.1. Throttling

Para asegurarnos de poder estar siempre a la escucha de nuevas conexiones, debemos definir un límite desde el cuál dejaremos de procesar requests, pero siempre manteniendonos activos y con respuesta inmediata. Para esto se implementa **throttling** en las colas en las que los boundaries se encargan de depositar los mensajes (que representan trabajo para nuestros workers). Estas colas se crearon con un tamaño configurable por parámetro, y al llenarlas, el sistema devolverá **Server At Capacity** hasta que se haga lugar en las mismas para nuevas solicitudes.

4.2. Affinity entre workers y métricas

Para evitar problemas de órden de escritura del archivo de lectura-escritura de la base de datos, se decidió implementar **affinity** entre métrica y worker. Esto quiere decir que **una determinada métrica siempre será atendida por el mismo worker**, lo que en nuestro caso se traduce a que sólo un thread va a escribir el archivo de dicha métrica. Esto tiene como gran ventaja que el mismo siempre permanecerá ordenado en cuanto a los eventos, que podría no ser así si tuvieramos varios threads escribiendo sobre este.

4.3. Particionamiento de archivos

Para particionar los archivos y **no almacenar información innecesaria**, se decidió primero utilizar un directorio con el nombre de la métrica, luego se define el nombre del archivo de la partición como *timestamp/partition_length* (en segundos), para finalmente guardar el remanente en milisegundos (para permitir mayor precisión en las queries) en cada evento guardado en el archivo. Cuando se quiere hacer una consulta, **se construye el timestamp original haciendo el camino inverso**.

5. Puntos de mejora

Aprovechndo el análisis realizado en una sección anterior sobre la arquitectura resultante, detallo a continuación algunos puntos de mejora posibles para el sistema desarrollado.

Algunos de estos puntos fueron incluso diseñados con diagramas de robustez, pero por cuestiones de tiempo no pudieron ser implementados.

5.1. Aggregator asincrónico

Esta mejora me parece la más importante y la primera que haría si tuviese que optimizar las consultas. En vez de mapear 1 a 1 los threads con un handler de consultas en el aggregator, podríamos diseñarlo de **forma asincrónica** de forma tal que pueda aceptar un número de conexiones considerablemente mayor, almacenando en memoria el estado necesario (resultados parciales, socket de la conexión TCP) para responder cuando sea posible.

Si además de esto, implementamos el patrón **fork-join** para favorecer la concurrencia de las lecturas a los archivos, tendríamos un aggregator súper escalable que podría responder a un enorme número de queries sin ningún problema y sin necesidad de agregar unidades de procesamiento.

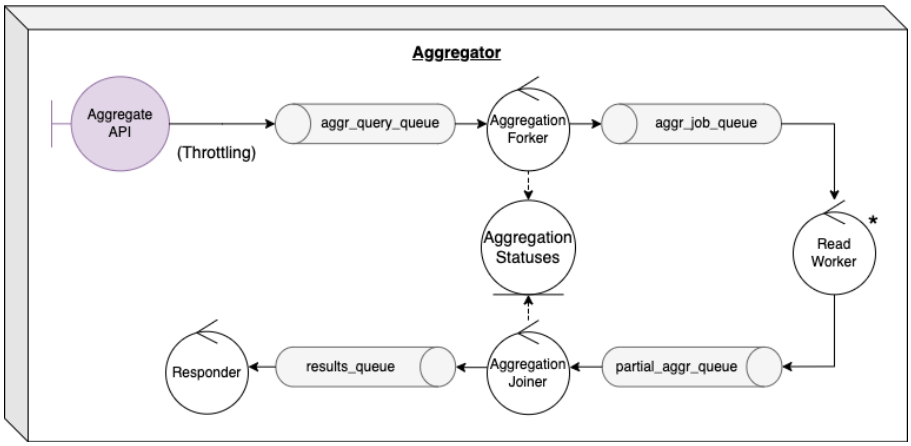

Figura 7: Posible implementación óptima del **Aggregator**

5.2. Cache entre la base de datos y el aggregator

Siguiendo con la optimización de consultas, podríamos implementar una **cache** entre el aggregator y la base de datos, que tendría mucho sentido en conjunto con el patrón **fork-join**, puesto que las agregaciones en este caso serían mucho más chicas, aumentando considerablemente la probabilidad de hits.

5.3. Elasticidad on-demand con métricas

Si nuestros workers reportasen métricas propias en un sistema aparte, que se encargue de recibirlas, analizarlas, y luego tomar acciones, podríamos ajustar la repartición de los recursos de forma dinámica obteniendo en régimen un funcionamiento prácticamente ideal, optimizando al máximo los recursos disponibles.

Para la recepción de métricas, se podría diseñar un sistema independiente de logging tan sencillo como este:

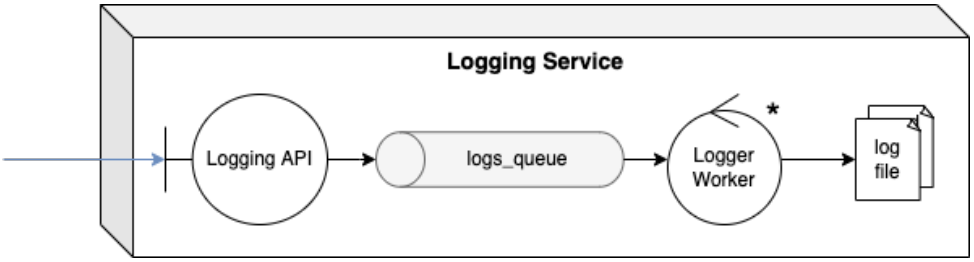

Figura 8: Posible implementación óptima del **Aggregator**

6. Conclusiones

Considero que el presente trabajo práctico resultó muy interesante para aprender a darle la importancia necesaria al diseño de la arquitectura de cualquier sistema antes de comenzar con su implementación, así como de la importancia de elegir el paradigma de concurrencia correcto para cada situación.