Trabajo Práctico 1

Ventura Julián
Padrón: 102391

Sistemas Distribuidos I
1ºC 2022
Fecha de Entrega:
28/04/2022

Contenido

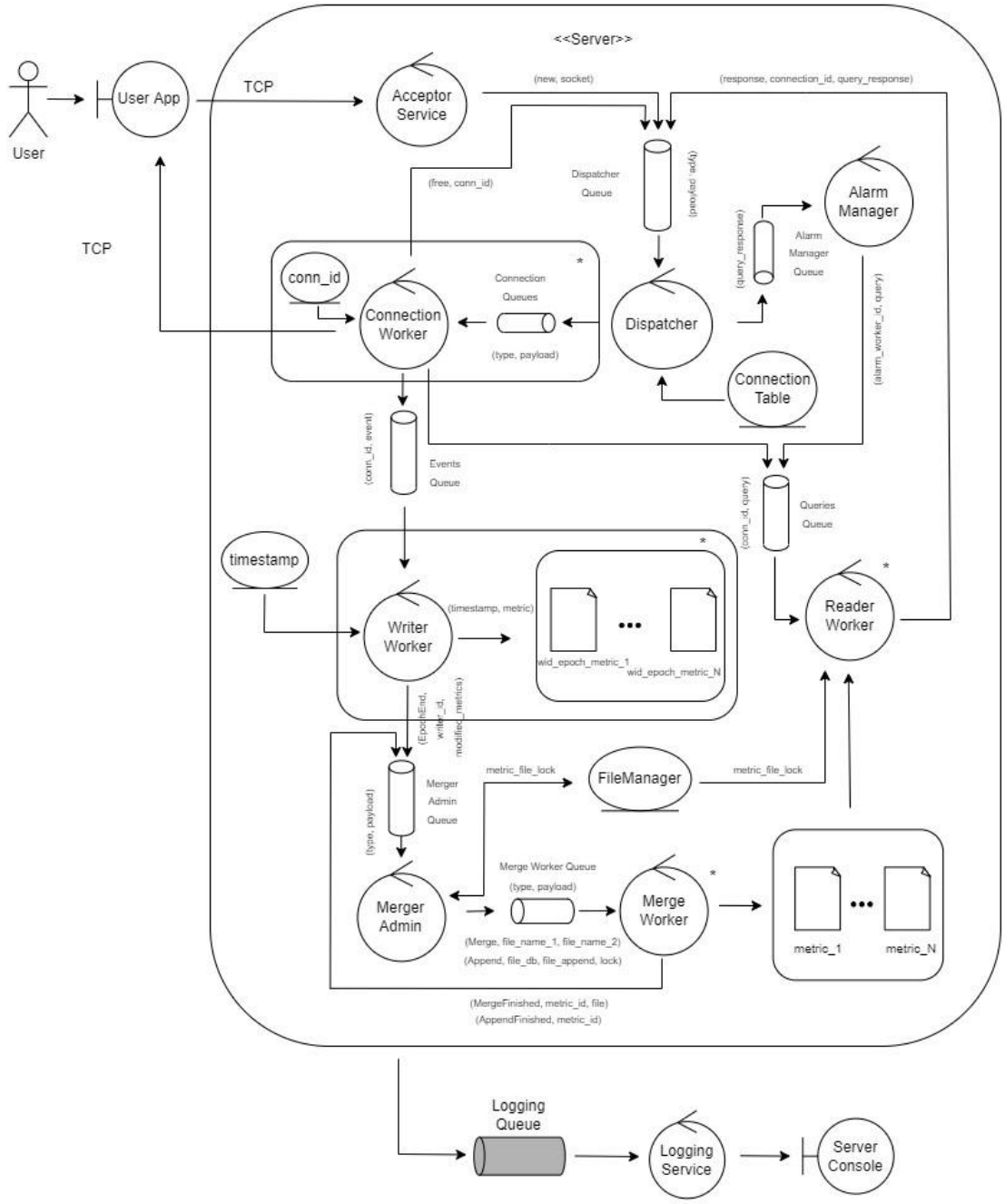
Decisiones Tomadas

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

- El servidor permite mantener conexiones abiertas con las aplicaciones de los clientes. A lo largo de la vida de esta, la aplicación podrá enviar múltiples métricas y queries de forma secuencial, obteniendo una respuesta del servidor en cada una de ellas.
- Las conexiones con las aplicaciones finalizarán automáticamente si no se detecta un evento por cierto tiempo configurable (timeout).
- Se permitirá un máximo configurable de N conexiones simultáneas. Un intento de conexión N+1 recibirá un mensaje de denegación por parte del servidor.
- El protocolo se implementará en binario, con el objetivo de minimizar el transporte de información entre cliente y servidor.
- Se asume que la cantidad de eventos de escritura sobre la base de datos es mucho mayor a la de eventos de lectura, por ende, se buscará maximizar la paralelización de los primeros frente a los segundos.
- El sistema provee un modelo de consistencia eventual sobre los datos, en el cuál se pone una cota inferior al tiempo demorado entre la escritura de una métrica y la posible lectura de esta en una query. Este tiempo, denominado epoch, es configurable.
- El timestamp asociado a cada métrica será creado justo antes de ser escrita en la base de datos, con el objetivo de garantizar el ordenamiento de estas en los archivos.
- No se provee tolerancia a fallos, sin embargo, en condiciones normales se realiza un cierre ordenado de los recursos sin pérdida de datos.
- Las alarmas serán configuradas al inicio del servidor desde un archivo de configuraciones. Son implementadas realizando queries periódicas a la base de datos, análogas a las queries de los clientes. No se diseñó un sistema de procesamiento paulatino de los datos.
- Debido a la naturaleza del lenguaje utilizado Golang, se trabajó con punteros a estructuras en la mayoría del lenguaje, buscando meramente (en la mayoría de los casos) una mejora de performance.

Arquitectura Propuesta

La arquitectura implementada en el sistema se muestra con el siguiente diagrama de robustez



Los puntos de entrada al Servidor se producen a través de la interacción del usuario con la aplicación Cliente. En el entregable se propusieron dos implementaciones de dichas aplicaciones, la primera orientada a la generación de métricas y la segunda al envío de queries.

Ambas aplicaciones se comunican de la misma forma con el servidor, siguiendo un protocolo TCP que será definido en una sección siguiente.

A continuación, se detallará brevemente cada proceso de la arquitectura, con el objetivo de acompañar el diagrama de robustez presentado.

- Acceptor: Encargado de realizar la aceptación de nuevos clientes al servidor y posterior comunicación al Dispatcher, a través de su cola de mensajes.
- Dispatcher: Encargado de realizar la repartición de mensajes entre el Alarm Manager y los Connection Workers. Entre los mensajes aceptados encontramos la generación de una nueva conexión, la finalización de una conexión y la respuesta de una query. Además, implementa la funcionalidad de rate limiting.
- Alarm Worker: Encargado de realizar chequeos periódicos sobre las métricas almacenadas en la base de datos, en base a alarmas prefijadas por configuración. Su trabajo será realizado a partir de la generación de mensajes Query a los Read Worker de la Base de Datos. Las respuestas a estas Querys serán recibidas en su cola de mensajes, gracias al trabajo del Dispatcher.
- Connection Worker: Encargado de manejar una conexión con un cliente, desde su inicio a fin utilizando un protocolo en común. Entre sus tareas encontramos la recepción de métricas y queries, su correspondiente validación en base a las reglas del negocio y finalmente su envío a las colas de Write Worker y Read Worker según corresponda. Además, en el caso de las queries se encargará de realizar la agregación de los resultados tras recibir la correspondiente respuesta del Read Worker. Tras la finalización de la conexión, indica al Dispatcher que se encuentra disponible para aceptar una nueva conexión.
- Write Worker: Encargado de recibir y procesar eventos de escritura de métricas a la base de datos. Cada uno de estos procesos dispondrá de una serie de archivos privados, uno por cada métrica, sobre los cuales tendrá acceso a escritura. Tras la culminación de cada epoch, notificará al Merger Admin la cantidad de métricas modificadas y generará una serie de nuevos archivos privados.
- Merger Admin: Encargado de administrar el mergeo de las métricas escritas por los Write Worker en cada epoch junto al resto de la base de datos para que puedan ser leídas por los Read Worker. Para esto contiene estructuras internas y delega las operaciones de mergeo sobre los Merge Worker. Entre los mensajes que recibe destacamos: Evento de finalización de epoch por parte de un Writer; Evento de finalización de mergeo por parte de un Merger; Evento de finalización de append por parte de un Merger.
- Merger: Encargado de realizar las operaciones de merge y append sobre dos archivos provistos por el MergerAdmin.
- Read Worker: Encargado de recibir y procesar parte de las querys a la base de datos. Con el objetivo de minimizar su trabajo, únicamente se encargará de realizar la recolección de las métricas que cumplan el rango de fechas establecido por los parámetros From y To. Tiene acceso de lectura únicamente a los archivos de métricas de la base de datos, uno por id de métrica, sin incluir a los que restan mergear.

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.

Solución de la Concurrencia

En el diseño de la arquitectura se buscó maximizar la paralelización disminuyendo todo lo posible el acceso compartido a un mismo recurso.

A excepción de la base de datos, se consiguió centralizar los recursos en cada proceso de forma tal que estos no debiesen tener algún tipo de mecanismo de sincronización, como puede ser un mutex. Ejemplos de esto lo vemos en el Dispatcher, quien tiene información de todas las conexiones abiertas y el id de cada una de ellas, o en las Connection Worker que mantienen el estado de la conexión con una aplicación de cliente en particular.

Siguiendo el principio de “Don’t communicate by sharing memory, share memory by communicating” se buscó una arquitectura orientada al pasaje de mensajes utilizando colas con mensajes bien definidos para cada proceso, implementadas como canales en el lenguaje Golang. Esto trae el beneficio de la no necesidad de realizar monitores para recursos compartidos.

Como contra, la utilización de colas de mensaje implica ciertas complejidades a la hora de realizar el envío de una respuesta a un mensaje emitido previamente por un proceso desconocido, como es el caso de una respuesta a una Query por parte de un Read Worker. Esto se resolvió añadiendo información identificatoria a cada query realizada que, reaprovechando la figura administradora de un único dispatcher, sirviese para encontrar el destinatario de forma unánime, en este caso pudiendo ser cualquiera de las N conexiones abiertas o el propio Alarm Worker.

Si se necesitó cierto grado de sincronización en cada proceso para evitar un posible deadlock que ocurriese al momento de realizar el cierre del servidor. El proceso en cuestión podría quedar bloqueado esperando la recepción de un nuevo mensaje que nunca llegase. Esto se evitó utilizando, para cada proceso, un canal secundario donde se pudiese informar la intención de finalización del servidor, tras la recepción de una señal de SIGTERM.

De esta forma, tras la recepción de dicho mensaje, cada proceso tiene una lógica determinada que le permite finalizar sin perdida de datos ni bloqueos.

Hasta este punto no hubo mucha complejidad, pues no existía necesidad de compartir estado entre los procesos. Sin embargo, esto no fue posible de lograr del todo en la base de datos.

Como se introdujo anteriormente, se orientó a la arquitectura a soportar un número de escrituras relativamente mucho más alto que el de lecturas. Para esto, y apoyándose en la premisa de consistencia eventual del sistema, se armó una división de los archivos de la base de datos de la siguiente forma.

Cada proceso escritor tiene acceso a un grupo de archivos privados a él, uno por cada métrica de la cuál recibió un mensaje de escritura, donde escribirá cada nueva métrica generada en orden ascendente por timestamp. Esto funciona genial, pero si los procesos lectores no tienen acceso a estos archivos, entonces las queries no verán reflejados los datos.

Por lo tanto es necesario que periódicamente se realice un merge de todos los archivos escritos por cada proceso escritor con los archivos que pueden ser accedidos por los procesos lectores. Aquí entra el concepto de época (epoch).

Cada escritor mantendrá una referencia de la epoch actual y escribirá sus archivos en un file system común con el resto de procesos de la base de datos, añadiendo un código

identificatorio asociado a su id de proceso, la métrica que está escribiendo y el número de epoch en el cuál se encuentra.

Cada epoch tiene una duración finita y configurable, por lo tanto cada escritor deberá ser capaz de reconocer la culminación de una epoch y de esta forma dar comienzo a una nueva, dejando libres de acceso a los archivos generados en la epoch anterior.

Además, cuando un escritor detecta la finalización de una epoch informa al MergerAdmin que sus archivos de la epoch anterior están libres para ser mergeados, indicando un listado de las métricas generadas.

El MergerAdmin mantiene un estado de la epoch global de la base de datos y recibe los mensajes EpochEnd de los procesos escritores. A medida que reciba los mensajes EpochEnd, irá almacenando las métricas que han sido modificadas en la epoch actual en una estructura local.

Además, en la medida que sea posible el MergerAdmin emitirá mensajes Merge a los procesos Merger indicando los archivos sobre los cuales se debe realizar una combinación ordenada por timestamp.

Alguno de los procesos merger tomará la tarea y realizará la combinación, sin necesidad de realizar algún bloqueo sobre los archivos. Tras su culminación, emitirá un mensaje MergeFinished al MergerAdmin, indicando el nombre del nuevo archivo resultado.

A medida que el MergerAdmin reciba los mensajes de MergerFinished de los mergers, irá enviando nuevos mensajes de Merge, acumulando los resultados hasta llegar a tener un único archivo disponible por procesar.

Hasta este punto todos los procesamientos pudieron realizarse en paralelo, sin condiciones de carrera. Lo que falta es realizar la combinación de las nuevas métricas generadas (resultado acumulado de todos los merge) junto al archivo de métricas de la base de datos. Aquí es donde se necesita utilizar un lock de archivos.

Se implementó una estructura FileManager compartida por el MergerAdmin y los ReadWorker que provee de locks para la lectura y escritura de archivos de la base de datos.

De esta toma, yras recibir el último mensaje EpochEnd, el MergerAdmin emitirá mensajes tipo Append a los procesos Merger, en los cuales indicará el nombre del archivo de la base de datos sobre el cuál realizar la combinación, el archivo resultado de las etapas de mergeo previas y un lock del archivo de la base de datos.

Cuando un Merger reciba un mensaje de append, tomará el lock en modo escritura y realizará la copia del archivo resultado dentro del archivo de la base de datos, para le métrica indicada. Tras lo cual liberará el lock del archivo y enviará un mensaje AppendFinished al MergerAdmin.

Cuando el MergerAdmin reciba respuesta de todos los mensajes Append generados, realizará un cambio de epoch para repetir el proceso.

Siguiendo este proceso se consigue, como se comentó antes, una paralelización de la escritura máxima, donde se puede tener a los N procesos escritores escribiendo una misma métrica de forma paralela sin pérdidas de rendimiento por sincronización.

Además, los procesos ReadWorker son capaces de leer de la base de datos de forma paralela, pues se utiliza un RWMutex que tiene esa funcionalidad.

Siendo que se posterga lo máximo posible el momento de realizar la combinación entre las nuevas métricas y las métricas ya existentes en la base de datos, al final de cada epoch, el tiempo que cada Merger tiene tomado el lock sobre un archivo es mínimo.

Por último, este particionamiento de los archivos únicamente por métrica, y ordenados en base al timestamp facilitan el algoritmo de búsqueda para la solución de queries.

Como contra, a esta arquitectura le encontramos dos problemas. El primero es su mayor complejidad frente a una solución que simplemente realice un particionamiento de los archivos por métrica y tiempo. El segundo es la demora que existe entre la generación de un dato y su correspondiente posible lectura por parte de un lector, que depende de la configuración. Una duración de epoch muy alta implicará que los datos leídos serán “menos frescos”, pero disminuirá la carga de los procesos de mergeo. En cambio, una duración de epoch muy corta tenderá a mejorar la frescura de los datos aumentando la carga en los procesos de mergeo. Es importante aclarar en este último caso que existirá un punto de quiebre en el cual la frescura de los datos no mejorará al disminuir la duración de los epoch, debido a que la propia carga de los mergeos implica una perdida de tiempo muy grande.

En conclusión, se cree que esta arquitectura ajusta correctamente al problema a resolver.

Una mejor evaluación consistiría en realizar pruebas cargando el sistema y midiendo cómo reaccionan las distintas arquitecturas ante los posibles casos.

Protocolo Implementado

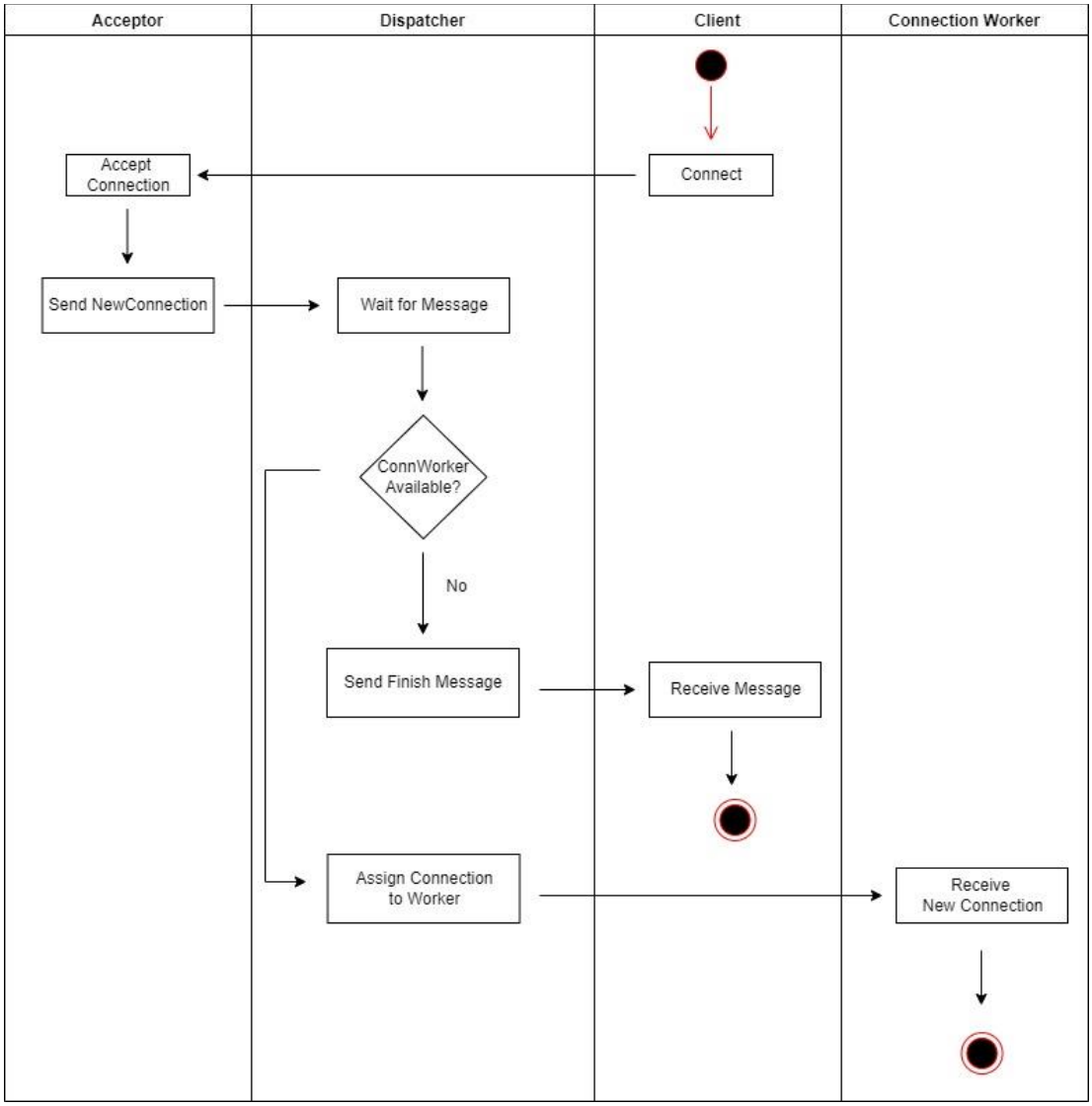
Se implementó un protocolo binario con las siguientes características:

- 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:
    - Metric
    - Query
    - QueryResponse
    - Error
    - Finish
    - Ok
- 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.
    - Un vector se codifica anteponiendo un entero de 4 bytes que indique su largo y concatenando las conversiones a BigEndian de cada uno de sus elementos, sin separadores.
    - Un enum se codifica como un byte, sin necesidad de hacer conversiones.

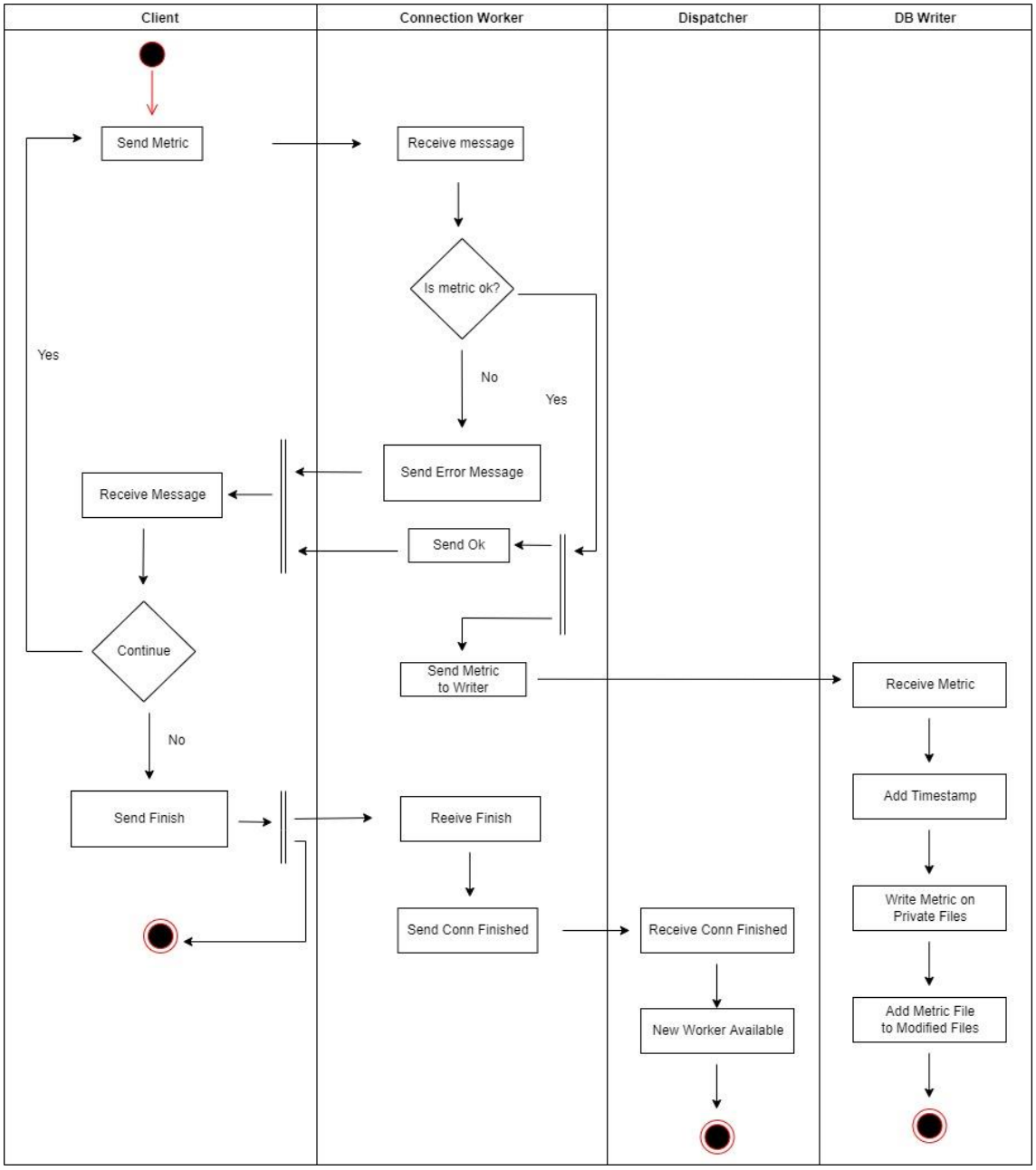
Diagramas de Secuencia

Se incluyen tres diagramas de secuencia que detallan, cada uno, una tarea distinta a realizar en el sistema

Conectarse al Servidor

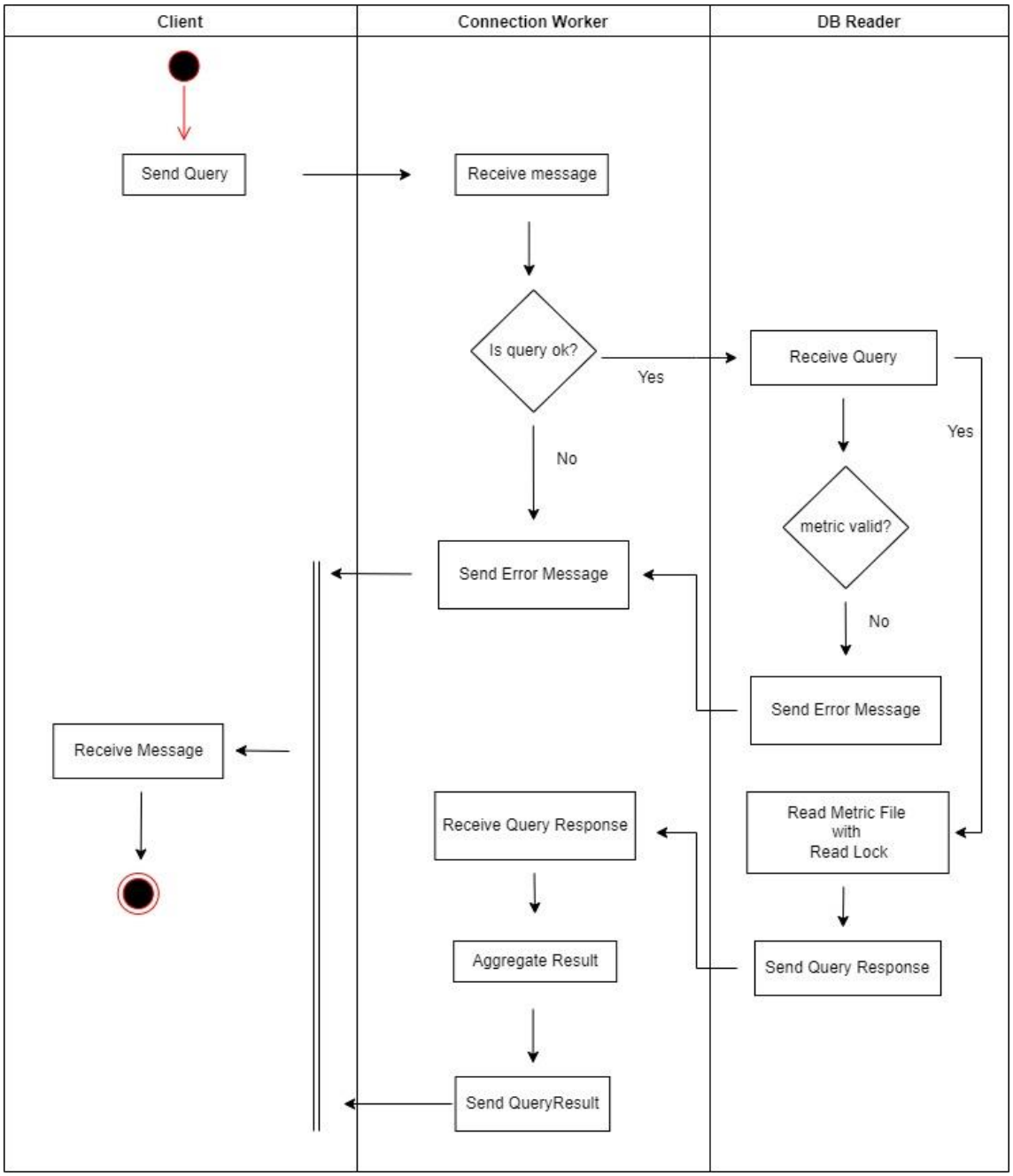


Enviar una Métrica



Enviar una Query

Nota: En este diagrama se elimina el recorrido que se realiza cuando el cliente desea desconectarse, pues es el mismo al detallado en el diagrama anterior.



Deuda Técnica

Manejo de archivos

Debido a cuestiones de tiempo, facilidad de debugging y de exposición, se decidió realizar el almacenamiento de las métricas en los archivos como cadenas de texto.

Principalmente debido a cuestiones de tiempo, se decidió priorizar temas de concurrencia, comunicación y arquitectura del sistema más allá del manejo de los archivos, por esa razón el sistema actual realiza una ineficiente y desprolija operación de estos. Cualquier tipo de lectura se realiza de forma completa, cargando todo el archivo en memoria, y luego se realizan los parseos necesarios dependiendo el caso, de forma lineal.

Una mejor implementación sería utilizar archivos en formato binario, permitiendo una mayor compresión de los datos.

Luego, los algoritmos de búsqueda podrían aprovechar el ordenamiento por timestamp del archivo, realizando un algoritmo similar a búsqueda binaria utilizando la función FSEEK, en lugar de leer todo el archivo a memoria.

Las posteriores lecturas podrían realizarse indicando de a porciones, indicando un tamaño de buffer determinado, para limitar el uso de memoria y la cantidad de syscalls invocadas.

Como solución alternativa, si no se quiere sacrificar la legibilidad del archivo de texto, se podría almacenar cada métrica como una cadena de bytes que representen una string de un tamaño fijo. Esto debería realizarse imponiendo un tamaño máximo de string y utilizando padding. Con esto lo que se lograría sería tener un archivo legible y la posibilidad de utilizar FSEEK, a costa de una menor compresión de información.

Tests Unitarios y de Integración

Tanto a lo largo del desarrollo del sistema como para su futura evolución, una suit de tests tanto de integración como unitaria sería ideal, aunque debido a la complejidad del problema, esta sería uno de los cambios más difíciles de realizar.

Deadlock entre MergerAdmin y Mergers

Bajo la implementación actual existe un posible riesgo de deadlock si ocurre que las colas del MergerAdmin y los Mergers se completan, debido a la propia dependencia cíclica que hay entre estos procesos.

En el código se dejó un comentario detallando una posible solución, denominada “Deadlock Avoidance”

Mejor Interfaz de Usuario

Como se comentó anteriormente, por cuestiones de tiempo se sacrificó en cierta medida la usabilidad de los clientes propuestos para realizar pruebas contra el servidor. Se hubiera deseado implementar modelos más flexibles y que brindasen más información, al igual que

algún script que permitiese generar un Docker-compose con N clientes de forma automática, de forma análoga al tp0.

Actualmente los tests pueden realizarse, pero esto puede implicar modificar archivos de configuración, Docker-compose y/o levantar más de un tipo de cliente de forma simultánea.