TP Final Sistemas Operativos 1

Servidor Memcached

Decisiones de diseño

Manuel Spreutels

Augusto Rabbia

Septiembre, 2023

Decisiones de diseño

Estructura de datos para el almacenamiento de claves.

La estructura de datos utilizada es una tabla hash, con el objetivo de proveer el acceso más eficiente posible a las claves almacenadas y de esta manera minimizar el tiempo de respuesta del servidor. El tamaño de la tabla está hardcodeado a un millón de entradas. Las colisiones se manejan a través de un área de rebalse, por lo que la tabla es en realidad un array de listas doblemente enlazadas DList.
Al ser una estructura concurrente, las casillas están protegidas por locks. Para intentar alcanzar un equilibrio entre el uso de memoria y la concurrencia de la que es capaz esta estructura, se decidió establecer un lock cada 10000 casillas (también hardcodeado). Esto es, dado el array de locks, locks[i] protege las casillas i*10000 hasta (i+1)*10000 - 1.

Política e Implementación del desalojo de memoria.

El desalojo de memoria fue manejado utilizando una estructura que llamamos “history queue”
Es simplemente una cola, que utiliza los mismos nodos de las listas de la tabla hash, (esta decisión de diseño se explicará a continuación) y los ordena según la antigüedad de la última acción que se realizó sobre cada uno de estos.
La estructura de los nodos tiene dos valores adicionales, que guardan cuál es el nodo inmediatamente más “antiguo” o “nuevo”. A su vez, la history queue guardará cuál es el nodo más recientemente modificado, y cuál es el más antiguo. Finalmente, la estructura de datos contiene un lock que la protege de condiciones de carrera y asegura su correcto funcionamiento.
El haber utilizado los mismos nodos de la tabla hash hace de la tarea de eliminar, insertar y actualizar nodos en la cola más sencillo:

- Al insertarse o buscarse un nodo en la tabla hash, por el lado de la history queue, se actualiza su posición, colocándolo al “inicio” de la misma (pasa a ser el más nuevo), y se modifican los valores en los campos “newer” y “older” de los nodos afectados (los adyacentes a este en la cola).
- Cuando la memoria se encuentra agotada, se selecciona el nodo más antiguo de la history queue y se elimina. Para realizar la eliminación, es necesario tomar su lock correspondiente en la tabla hash, de esta forma, nos aseguramos de que no exista una condición de carrera a la hora de removerlo. Sin embargo, durante este tiempo puede ocurrir que este nodo deje de ser el más antiguo, o haya sido eliminado. En este caso, no se eliminará.

Se puede observar que no está asegurado que la history queue elimine un valor cuando se llama a eliminar_menos_util(), ya que para asegurarlo, se tendría que tomar en primer lugar el lock de la history queue y luego el de la tabla, pero esto puede llevar a un deadlock. Sin embargo, a la hora de llamar a esta función, siempre se debe utilizar un while loop, puesto que, de todas maneras, no se puede estar seguro de que eliminando un único nodo se liberará suficiente memoria como para alojar la memoria que se necesita.

Este loop solucionaría el problema de que no se elimine un nodo en alguna llamada a esta función, pues además, suponemos que eventualmente siempre ocurrirá una eliminación de algún nodo, y que sucederá en pocos intentos, ya que se trata de una ocurrencia relativamente rara, principalmente cuando hayan muchos mensajes y una tabla hash de tamaño grande.

Epoll y recurrencia

Respecto de este tema tuvimos gran cantidad de dificultades. Muchas de las fuentes de información eran contradictorias entre sí, no trataban el epoll desde el punto de vista de multithreading o no eran lo suficientemente exhaustivas respecto de algunos comportamientos. En un inicio contábamos con un array de epoll_event global, y sólo permitíamos que un único thread accediera a ejecutar la función epoll_wait(), de la siguiente manera:

```
void* server_threads(void* nprocArg) {
  int eventFd;
    while (1) {
      pthread_mutex_lock(&pedidos);
      if (eventIndex >= nEvents) {
        pthread_mutex_lock(&eventsMutex);
        nEvents = epoll_wait(epollfd, events, sizeSockTypeArray, -1);
        pthread_mutex_unlock(&eventsMutex);
        if (nEvents == -1) {...} // error-checking
          eventIndex = 0;
      }
      eventFd = events[eventIndex].data.fd;
      eventIndex++;
      pthread_mutex_unlock(&pedidos);
      if (eventFd == text_sock || eventFd == bin_sock) {
        // Nuevo cliente se intenta conectar
        // . . .
      } else {
         // Handle_connection
      }
   }
   return NULL;
}
```

(sockTypeArray ya no existe).

Así, un thread accedía solo al wait si todos los eventos se habían manejado o se estaban manejando, y cada thread manejaba un evento distinto. Sin embargo, luego de asistir a una consulta, supimos que este código no era el estándar para realizar servidores multithread con epoll, por lo que finalmente decidimos evitar las posibles condiciones de carrera haciendo el array de eventos una estructura local para cada thread, y añadiendo la bandera EPOLLONESHOT a los files descriptors de interés. En teoría, estas race condition se solucionan sólo a través de la bandera EPOLLEXCLUSIVE, pero a nosotros nos generaba

errores, por ejemplo, en algunos casos epoll_ctl() devolvía -1, o bien varios threads “entraban” a manejar un mismo evento, en contradicción con el comportamiento esperado de la bandera, además, en algunos casos, existían clientes que al intentar conectarse no generaban un evento en el kernel de epoll.
En nuestra implementación, cuando llega un evento, gracias a EPOLLONESHOT, se retira al respectivo file descriptor de la lista de interés, se gestiona el evento, y una vez hecho esto se lo vuelve a insertar en la lista de interés del kernel. Esto lo hacemos para evitar race conditions (thundering herd).

Para el manejo o la discriminación entre clientes en protocolo binario o en protocolo texto, se crea en el heap, para cada cliente conectado, una estructura que mantiene el fd que se le asignó, así como el tipo de protocolo de la conexión, que es toda la información que un thread necesita antes de comenzar a manejar una conexión. El campo event.data.ptr apunta a esta estructura, y luego llamamos a la función epoll_ctl() sobre event para añadirlo al kernel de epoll, de manera que esta información sea visible a todos los threads a través de epoll.

Parseo de entrada

El parseo de la entrada en el modo texto se realizó utilizando la función proporcionada, por lo que no hay comentarios adicionales. Por otro lado, el parseo del modo binario tiene ciertas complicaciones.
Para realizarlo, utilizamos una secuencia de reads parciales, ya que no hay ninguna forma de saber en un inicio el tamaño del mensaje a leer.
Primero leemos el primer bit, que nos dirá qué acción se ha de tomar, y, si es STATS, verificamos que el mensaje está bien formado (que no haya más bytes por leer), terminamos de parsear y pasamos a responder el mensaje. Sino, verificamos que sea cualquier otro comando válido, y si lo es, continuamos.
Luego, leemos los próximos 4 bytes y determinamos el tamaño de la key. Con esta información, leemos exactamente la cantidad de bytes especificada en este campo. Luego, si el comando era GET o DEL, dejamos de parsear y pasamos a responder el mensaje. Finalmente, el caso PUT es análogo, leemos el tamaño del valor y leemos tantos bytes como es necesario.
Ahora bien, a la hora de leer en este modo, es necesario utilizar un while loop. Esto es porque, en caso de que el mensaje fuera demasiado largo, es posible que este no quepa en el buffer interno del kernel de epoll. En estos casos, entra una cantidad menor de bytes. Entonces, seguiremos intentando leer del file descriptor hasta que la cantidad de bytes leídos sea igual al tamaño especificado en el mensaje.
Por último, cabe recalcar que si se envían múltiples mensajes al mismo tiempo, pero uno de ellos no se ajusta al protocolo, descartamos todo el contenido posterior, ya que no podemos asegurar que esos mensajes serán leídos de manera correcta, pues pueden arrastrar errores del mensaje incorrecto.

Secuencia general del recibimiento de un mensaje

Al llegar un mensaje al servidor, se realizan entonces la siguiente secuencia de operaciones para generar una respuesta:

1. Un proceso loopeando en la función **server_threads()** realiza un epoll_wait(), y recibe el mensaje.
    a. Si era un nuevo cliente, simplemente se acepta y continúa loopeando.
    b. Si era un cliente ya reconocido, continúa en el punto 2.
2. Se entra a **handle_conn_*X*()**, en este caso, supongamos que es binario, y luego, se entrará a **bin_consume()**. (Donde *X* representa *bin* o *text*)
3. Se parsea el mensaje, colocando los valores de interés en las variables *action, sizeKey, key, sizeVal,* y *val*, y se llama a **bin_handle()** con estos argumentos (siempre que el mensaje no contenga errores).
4. Se determina el tipo de acción que se desea realizar. Si era STATS, se responde. En otro caso, luego de incrementar el campo de las estadísticas correspondiente a la acción que se realizó (*stats[i]*++), se llama a la función de la tabla hash correspondiente a la acción (**tablahash_insertar()** / **tablahash_buscar_y_retornar()** / **tablahash_eliminar()**).
5. Desde estas funciones, actualizamos la tabla hash, pero a su vez, actualizamos la history queue para colocar la última acción al principio de la cola. Una vez hecho esto, devolvemos el resultado de la operación (si es un get, el valor, si es un put, si se pudo insertar en la tabla (puede ser que no haya suficiente espacio en la memoria y falle)).
6. Escribimos el resultado en el socket con la función **WRITE()**.
7. Finalmente, volvemos hasta el loop de la función **server_threads()**, y esperamos a un nuevo cliente.

Cliente en Erlang

Creamos dos clientes en Erlang: uno de texto y otro binario. Sin embargo, ambos fueron implementados de forma similar:
La función start únicamente se encarga de hacer un spawn, creando un nuevo proceso que se quedará a la espera de órdenes para enviar al servidor, y devolviendo su ID de proceso. Esta ID será utilizada para llamar a las demás funciones del cliente y estas simplemente enviarán un mensaje al proceso loopeando, especificando qué debe mandarse, y este se encargará de realizar el envío.

Testeos

Para la realización de testeos se utilizó una versión modificada del cliente de Erlang, de manera que se pudieran realizar miles de conexiones y peticiones en pocos milisegundos. Se limitó la memoria a 70 MB para comprobar el correcto funcionamiento del proceso de desalojo. También se redujo el tamaño de la tabla hash a 100 casillas y con 1 y más lock para el testeo de

deadlocks. Las respuestas eran satisfactorias incluso con 10000 clientes a la vez, en términos de consistencia y rapidez.