Informe Memcached

Sistemas Operativos I

Juan Segundo Valero

21 de agosto de 2023

Funcionamiento general del programa

Almacenamiento de los datos:

Para almacenar los datos se utiliza una tabla hash, el funcionamiento de esta se basa en un arreglo de listas enlazadas, la estructura "Word"(Detallada en helpers.h) representa un nodo de una lista enlazada de la tabla hash. Para localizar o ingresar un elemento en la tabla lo que se hace es obtener un hash utilizando el algoritmo de MurmurHash2 (https://sites.google.com/site/murmurhash/) sobre la clave, luego se obtiene la posición en el arreglo haciendo hash % tamaño_de_la_tabla y se recorre la lista de esa posición hasta encontrar (o no) el elemento deseado.

Para solucionar el problema de condición de carrera en multithreading se implementó un sistema de locks sobre la tabla hash, se crea un array de locks que tiene 1/10 del tamaño que el arreglo de la tabla hash, por lo cual hay uno cada 10 listas enlazadas, el lock que le corresponde a la lista enlazada es obtenido haciendo posición % tamaño_arreglo_locks.

Manejo de memoria:

Para solucionar el problema del desalojo cuando la memoria esta llena se utiliza un protocolo de LRU (least recently used), la implementación esta dada por una lista enlazada anidada dentro de la tabla hash que contiene a los datos. Se guarda un puntero a la cabeza y a la cola de esta lista y cuando se necesita borrar un elemento se accede al último elemento, si este está siendo utilizado por otro thread se intenta de nuevo con el siguiente elemento de la lista hasta liberar alguno, en el caso de no poder liberar mas memoria se devuelve un error al cliente.
Nota: para indicar que no hay más memoria disponible para liberar se creó un nuevo código EMEM (116).

Networking:

El networking está manejado por epoll, se crean dos sockets de escucha, uno en el puerto 888 (modo texto) y uno en el 889 (modo binario) y se agregan a la lista de interés de epoll. Luego se generan X threads, X siendo la cantidad de cores que tiene el procesador, y en cada uno hay un epoll_wait, esperando requests a estos sockets. Los sockets de escucha están inicializados con EPOLLET que pone epoll en modo edge-triggered y los sockets de los clientes con EPOLLONESHOT que hace que solo un thread pueda utilizar ese socket al mismo tiempo, esto permite el uso del epoll en múltiples threads. Cada request que llega al servidor, es tomada por un thread individual, este recibe la información y ejecuta el comando mientras que los otros threads están libres para recibir mas llamadas.

Parseo de comandos

Modo texto:

Al ejecutar un comando en modo texto, la función thread_f, que es la que incluye el epoll_wait, detecta que la request viene al puerto 888 y llama a la función handle_conn, que se encarga de llamar a dos funciones, text_consume e input_handler, y manejar sus resultados. El propósito de text_consume es leer el file descriptor hasta encontrar un '\n', lo cual indicaría un comando completo, o hasta que no haya más caracteres en el buffer, lo cual indicaría que este comando está cortado. Si el comando no se pudo completar se guarda lo recibido en un buffer hasta que llegue la siguiente request que debería completar el comando, esto se hace por si hubo algún problema de ruido en la request. En caso contrario, se llama a la función parser, que divide el comando recibido en 'CODIGO-KEY(si la tiene)-VALUE(si la tiene)' y luego a la función input_handler, que se encarga de ver el resultado parseado y llamar a la función que lleve a cabo el comando solicitado, o de devolver EINVAL en caso de que este mal formado.

Modo binario:

El funcionamiento en modo binario es muy similar al del modo texto, thread_f detecta la request al puerto 889 y llama a handle_conn_bin, que contiene a las funciones text_consume_bin y parse_text_bin. La principal diferencia entre estas y las del modo texto es la manera en la que se detecta un comando incompleto. La función text_consume_bin se encarga de leer todo lo que haya en el buffer del file descriptor y luego llama a parse_text_bin, esta revisa la cantidad de bytes que contiene la request y detecta si son suficientes o si el comando esta incompleto. Es decir, primero revisa si tiene el byte que contiene el código del comando, si este no es STATS requiere una key como mínimo, entonces se observa si contiene los 4 bytes del tamaño y si tiene esos bytes en la clave. Luego si el comando es PUT se verifica este mismo procedimiento pero con el value. En caso de que no esté completa la instrucción se guarda lo recibido en un buffer en el que se continua la escritura al momento de recibir otra request, y en caso de que el comando sea válido, se pasan los valores obtenidos a la función input_handler_bin que se encarga de leerlos y ejecutar la instrucción solicitada.

Ejecución de instrucciones

STATS

STATS es el comando más sencillo de ejecutar, el seguimiento de los valores de interés se hacen mediante variables globales las cuales tienen un lock exclusivo. Al solicitar esta instrucción se hace lock de todas las variables globales y se escriben en un string que se devuelve al cliente. STATS incluye: cantidad de PUTS, cantidad de DELS, cantidad de GETS y cantidad de elementos KEY-VALUE actualmente en la memoria.

GET *clave*

La función GET se encarga de, dada una clave, obtener el valor asociado a esta. Para lograr esto se hashea la clave dada para obtener la supuesta posición de la lista enlazada que la contiene en la tabla hash, luego se hace un lock del mutex que contiene a la zona de esta, una vez obtenido se recorre esta lista comparando este hash con el hash de la clave de cada nodo, y si coinciden, se pasa a comparar carácter por carácter la clave. Si se encuentra un match devuelve OK *VALUE*, en caso contrario, ENOTFOUND.

PUT *clave valor*

Dada una clave y un valor, PUT coloca estos en la tabla hash. Si la clave ya se encuentra en la tabla se actualiza el valor, si no se crea un nuevo nodo con los datos provistos. Primero se trata de localizar la clave dada en la tabla hash, en caso de ser encontrada se actualiza el valor y se aumenta la variable PUTS, que lleva la cuenta de los puts hechos, en uno. Si no se encuentra la clave, se crea un nuevo nodo y se coloca al final de la lista enlazada a la que pertenece, se aumenta en uno tanto la variable PUTS, como la de KEYVALUES, que lleva la cantidad de pares clave-valor que están guardados en memoria. Si el valor es colocado exitosamente en la tabla, devuelve OK.

DEL *clave*

DEL se encarga de localizar y de eliminar de la memoria una clave dada. El funcionamiento se basa en hashear la clave provista, localizar la lista enlazada a la que debería pertenecer, bloquear el mutex de la zona y luego buscarla en esta lista de la misma manera que en la instrucción GET. Si es localizada, se borra de la lista enlazada y la memoria es liberada, en ese caso devuelve OK, en caso contrario se devuelve ENOTFOUND.

Actualización de la cola de borrado:

En todas las instrucciones, exceptuando DEL y STATS, el valor con el cual se esta interactuando es colocado al principio de la cola de borrado, es decir, va a ser el último elemento a borrar en caso de que se quede sin memoria el programa, debido al protocolo LRU (least recently used) utilizado.

Estructuras utlizadas

CompString

CompString contiene dos miembros, string y len, el primero siendo una cadena de caracteres y el segundo un entero. Es una estructura utilizada para facilitar el guardado y la comparación de claves y valores sin la necesidad de utilizar la librería "string.h"que no soporta cadenas que contengan '\n'. El primer miembro, string, contiene la cadena, mientras que len el largo de esta.

```
typedef struct CompStringStruct{
  char * string; // cadena de caracteres
  int len;   // largo de la cadena
} CompString;
```

Word

Word representa un nodo de la lista enlazada, contiene varios miembros:

1. word: CompString que contiene la clave del nodo
2. value: CompString que contiene el valor del nodo
3. bin: Un entero que indica si el nodo se guardo en modo binario (1) o en modo texto (0)
4. hash: Entero sin signo que contiene el hash de la clave.
5. next_delete/prev_delete: Dos punteros a otras words que representan el elemento siguiente y anterior en la cola de borrado respectivamente
6. next/prev: Punteros al word siguiente y anterior de la lista enlazada.

```
typedef struct WordStruct{
  CompString word; // clave del nodo
  CompString value; // valor del nodo
  int bin; // flag que indica si es binario
  unsigned int hash; // hash de la clave
  struct WordStruct* next_delete; // elemento siguiente a eliminar luego de este
  struct WordStruct* prev_delete; // elemento anterior a eliminar antes de este
  struct WordStruct* next; // siguiente elemento en la lista enlazada
  struct WordStruct* prev; // elemento anterior en la lista enlazada
} Word;
```

SocketData

Estructura que se le asigna a los eventos de epoll que contiene información extra para la protección de ruido y para el reconocimiento del tipo de instrucción. Tiene como miembro:

1. fd: File descriptor asignado al evento
2. bin: Entero que representa si el socket es de modo binario (1) o texto (0)
3. buf: Cadena de caracteres que contiene el buffer de instrucciones de ese file descriptor en caso de que un comando este incompleto
4. index: Entero que representa el índice en el buffer
5. size: Entero que representa el tamaño utilizado en el buffer
6. a_len: Tamaño total del buffer

```
typedef struct SocketDataS {
    int fd; // file descriptor del socket
    int bin; // flag para saber si es binario
 char * buf; // buffer de entrada
 int index; // indice en el buffer
 int size; // bytes usados del buffer
 int a_len; // tamaño total del buffer
} SocketData;
```