UNR - FCEIA

SISTEMAS OPERATIVOS 1

Trabajo Práctico:
Mini Memcached

Luis Aseguinolaza, Spoletini Bruno

Trabajo Práctico Final
Mini Memcached

Spoletini Bruno, Luis Aseguinolaza 02/07/2023

Introducción

El objetivo de este trabajo práctico es implementar un memcached propio con funcionalidad relativamente completa.
Dentro de las principales características de este trabajo se incluyen:

- Soporte para protocolo de texto, y un protocolo binario, que puede aceptar caracteres que no sean texto
- Interoperabilidad entre ambos protocolos
- Limite de memoria configuarable
- Soporte para multi-threading, en el cuál la carga de las solicitudes de los clientes se divide entre todos los threads disponibles
- Una libreria en Erlang que permite interactuar con el protocolo binario.

Políticas de desalojo

Si se intenta agregar un nuevo par clave-valor al memcached, y se llega al límite de memoria, es necesario "desalojar" parte de la memoria ocupada. Para esto es necesario dedicir cuáles pares clave-valor olvidar primero, y lo ideal es elegir el par al que no se haya accedido por más tiempo.
Para esto diseñamos una lista doblemente enlazada de punteros a claves-valor, en la cuál:

- Cuando se agrega un par al memcached, se agrega un puntero que apunta al par en el final de la lista
- Cuando se elimina un par, también se elimina el puntero de la lista
- Cuando se busca un valor con el comando GET, el puntero que apunta a ese nodo pasa al final de la lista

De esta forma nos aseguramos que el primer elemento de la lista es el nodo menos accedido recientemente (LRU), y tenemos un rápido acceso al mismo cuando sea necesario liberar memoria ocupada.
Cuando vayamos a liberar la memoria, intentamos bloquear la lista de claves valor donde se almacena dicho nodo para eliminarlo. En caso de no ser posible, intentamos con el próximo nodo de la lista de LRU.

Estructura de datos utilizada

Para almacenar los datos del memcached, decidimos utilizar la siguiente tabla hash:

```
struct Hashtable{
    struct DList *row[TABLESIZE];
    pthread_mutex_t rlock[TABLESIZE], locklru;
    struct DList *LRU;
    struct Stats *stats;
};
```

La tabla consiste de un array de punteros a listas doblemente enlazadas "row", donde se guarda otra estructura con los pares clave-valor, un array de locks, un lock para el LRU, un puntero a una lista doblemente enlazada para el LRU, y un puntero a la estructura de estadísticas. En caso de una colision en la tabla, el data se almacena en el siguiente elemento de la lista doblemente enlazada.

La lista doblemente enlazada tiene la siguiente forma:

```
 struct DNodo {
  Node *dato;
  struct DNodo *ant;
  struct DNodo *sig;
  struct DNodo *othernode;
} ;
struct DList{
  DNodo *primero;
  DNodo *ultimo;
};
```

Cuenta con un puntero al inicio, y uno al final de la lista. Cada elemento de la lista apunta al elemento anterior, al siguiente, a un nodo "otro" y a una estructura de tipo Node. La funcionalidad del puntero a "otro", es para conectar los nodos de la DList de claves-valor, con los nodos de LRU correspondientes.
La estructura tipo Node tiene la siguiente forma:

```
struct Node{
    char * key, *value;
    int lenkey, lenvalue;
    unsigned long long slug;
    int hash;
    bool printable;
};
```

En esta estructura se almacenan punteros a la clave y al valor, enteros con la longitud de cada uno, el hash de la clave, y el slug, que representa una etapa previa del hash que no está limitada al tamaño de la tabla.

Manejo de conexiones

Para el manejo de conexiones usamos una instancia de epoll, agregamos el socket de texto y el socket binario a la lista de escucha de la instancia, creamos tantos hilos como se especifique en el archivo de configuración (o se pasen como parámetro) y cada uno de estos se pone a esperar eventos de epoll.
Si se detecta un evento en el file descriptor del socket de texto o el socket binario, significa que un cliente está intentando conectarse al servidor por medio de dicho socket. Aceptamos la conexion entrante, y agregamos el socket del cliente a la lista de escucha de epoll. Una vez que el cliente sea agregado, cada vez que epoll detecte que hay informacion para leer en uno de los sockets, se despertará un hilo para que lea la informacion del socket, la procese, y si es una solicitud correcta, la procese y responda.
Teniendo en cuenta que nada garantiza que los pedidos lleguen completos, o lleguen juntos uno con otro, es necesario almacenar la informacion de un pedido incompleto. Para esto usamos la estructura epoll_event, la cuál le especifica al kernel que informacion debe guardarse y recuperarse cuando un file descriptor está listo.

La estructura epoll_event contiene los eventos, y una variable tipo void*, la cual usamos para guardar la información de cada socket de cliente utilizando la siguiente estructura:

```
1  typedef struct _eloop_data {
2    int fd, epfd;
3    int isText, einval, notPrintable;
4    char *buff;
5    int buffSize;
6    int cont, comm;
7    int keySize, valueSize
8    char* key;
9    char* value;
10   Hashtable* hTable;
11 } eloop_data;
```

Esta estructura es la que acompaña a cada cliente en cada evento I/O. Contiene el socket del cliente junto con el de la instancia de epoll, banderas que indican si el cliente se conectó usando protocolo de texto o protocolo binario, si el stream de datos que se está procesando es mas largo del permitido, y si la solicitud contiene caracteres no imprimibles.
Guardamos un buffer con su tamaño para los datos que procesamos en el modo texto, y dos buffers key y value con su tamaño respectivo para los datos procesados en el modo binario, junto con un contador usado para checkear que parte de la solicitud se está procesando, y una variable comm que lleva el código del comando ingresado. Junto con esto también guardamos un puntero a la tabla hash. Todo esto nos permite guardar la infomacion no procesada de los clientes que se conectaron a cualquiera de los sockets, y nos permite garantizar la interoperabilidad entre ambos.

Consideraciones

En caso de que se detecte un comando inválido durante la lectura de un protocolo binario, se ha tomado la determinación de desconectar al cliente. Esta decisión se basa en el hecho de que, debido a la forma en que se transmiten los datos en dicho protocolo, resulta imposible determinar cuándo finaliza el flujo incorrecto de datos y cuándo comienza uno nuevo que sea válido. Si continuáramos leyendo el flujo de datos, existe la posibilidad de que dentro del flujo incorrecto se encuentre un comando válido (PUT, GET, DEL o STAT). En tal caso, se procedería a leer el tamaño de la clave y asignar memoria para almacenarla en la tabla hash, para luego llenarla con el flujo incorrecto. Esto daría lugar a la ocupación de memoria en el servidor con datos erróneos.

Por esto, se ha decidido desconectar al cliente cuando se detecte un comando inválido, con el objetivo de evitar problemas de ocupación de memoria y garantizar el correcto funcionamiento del sistema.

Fuente del código

Las librerías de la tabla hash y la lista doblemente enlazada fueron hechas durante el cursado de Estructuras de Datos y Algoritmos 1.
El resto del código fue hecho en base a los contenidos dados en la materia, sumados a los archivos provistos por la cátedra.

Parámetros y ejecución

Los parametros del programa se pueden editar mediante linea de comando o editando el archivo $config.h$. En caso de querer hacerlo por consola, se debe ejecutar el comando:

sudo make ARGS="..."

Donde los posibles argumentos son:

- -nt p: donde p es el numero de threads que se desea que corran el servidor.
- -t p: donde p es el puerto del socket de texto deseado.
- -b p: donde p es el puerto del socket binario deseado.
- -m p: donde p es el limite de memoria en bytes deseado.

De ejecutar "sudo make", el programa se ejecuta con valores por defecto.

Fuentes consultadas

https://es.wikipedia.org/wiki/Bloqueo_mutuo

https://man7.org/linux/man-pages/man7/epoll.7.html

https://elcodigoascii.com.ar/

https://man7.org/linux/man-pages/man2/read.2.html

https://es.wikipedia.org/wiki/Endianness

https://www.ibm.com/docs/es/openxl-c-and-cpp-aix/17.1.0?topic=support-unary-binary-operators

https://stackoverflow.com/questions/888386/resolve-circular-typedef-dependency

https://docs.oracle.com/cd/E36784_01/html/E36870/netcat-1.html