Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Sistemas Operativos I

Trabajo Práctico Final: Mini Memcached

Bacci Mateo, Giulianelli Sebastian

Rosario, Santa Fe, Argentina

18 de diciembre de 2023

Índice

1. Servidor

1.1. Funcionamiento general

Esta implementación *memcached* consta de una primera etapa de inicialización, en la que se limita la memoria, se establecen los manejadores de señales, y se crean las estructuras de datos, *sockets* y procesos ligeros (hilos) necesarios para el correcto funcionamiento del servidor.
Una vez que se tiene todo listo, con el uso de una interfaz *epoll*, se agregan los *sockets* por los que se recibirán clientes, uno para el modo texto y otro para el modo binario. Luego todos los procesos se mantienen bloqueados hasta que en alguno de estos *sockets* se detecte algún evento, es decir, hasta que se conecte un nuevo cliente. Una vez con el *socket* de este cliente, se lo agrega a la instancia *epoll* para sus futuros mensajes.
Los hilos (habrán tantos como *cores* tenga el procesador) procesarán los eventos que lleguen sobre los *sockets* de los clientes registrados y luego volverán a su estado de espera.
Para compilarlo basta con ejecutar *make* en la terminal y luego ejecutar *./memcached* que acepta un argumento opcional que representa el límite de memoria en bytes. Por defecto, el servidor utiliza 3 GB (o la cantidad de memoria ram disponible en caso de ser menor). Además, como internamente el servidor necesita al menos 40 MB para las estructuras, hilos y demás, el valor ingresado como argumento será la cantidad de bytes que el servidor tendrá disponible además de esos 40 MB que ya utiliza por defecto. Entonces, por ejemplo, si se ingresa 20.000, el servidor utilizará 40 MB más esos 20.000 bytes.

1.2. Estructuras de datos

En el servidor se llevaron a cabo dos estructuras de datos distintas:

- **Cola**: Implementación con dos punteros, uno al primer nodo y otro al último, permitiendo mayor eficiencia a la hora de *pushear* y *popear* elementos. Tiene dos usos, uno es mantener un registro del orden en el que se guardan los datos en la tabla hash. Debido a la política de desalojo elegida (explicada en el siguiente apartado), se necesita conocer el orden en el que los pares son utilizados por los clientes (ingresados por primera vez, modificados o consultados), para eliminar los utilizados hace más tiempo. Se utilizó una implementación concurrente para que sólo la utilice un trabajador a la vez. Para no tener información repetida, sus nodos en vez de almacenar los pares clave-valor, almacenan el índice en el que dicho par se encuentra en la tabla. Su segundo uso es en la tabla hash para resolver colisiones.

- **Tabla Hash**: Guarda los pares clave valor ingresados por los clientes. Internamente utiliza colas para manejar las colisiones. Cuenta con su versión concurrente en la que se utilizaron diez *locks* distintos, así un trabajador no bloquea toda la tabla cada vez que tenga que utilizarla. De esta

manera, se podría tener hasta diez trabajadores accediendo a ella (si cada uno toma un lock distinto) garantizando la consistencia en los datos.

1.3. Política de desalojo

La política de desalojo elegida fue *LRU (least recently used)* ya que al eliminar los elementos utilizados hace más tiempo, se espera que sean los menos útiles o menos importantes por el momento.
Para implementarlo se utilizó una *queue* o cola, la cual fue llevada a cabo como se explicó en el apartado anterior. Cuando se alcanza el límite de memoria, el servidor realiza los siguientes pasos:

1. Elimina el último elemento de la cola de desalojo, es decir, el elemento que se usó hace más tiempo. En esta cola, los elementos son de la forma $(n, NULL, NULL)$ donde n es un índice de la tabla hash.

2. Como el nodo eliminado es el utilizado hace más tiempo entre todos los elementos de la tabla, particularmente, también lo es entre los nodos de su misma casilla. Por lo tanto, se elimina el último nodo de la casilla n.

3. En caso de necesitar más memoria, se repiten los pasos 1 y 2 hasta que haya memoria suficiente, o bien, hasta que no queden elementos en la tabla.

Para insertar, consultar o eliminar un valor se necesitan ambas estructuras por lo que se deben tomar dos mutexes distintos, particularmente, se toma primero el de la tabla (ya que para las tres operaciones ya se conoce el índice) y segundo el de la cola. Sin embargo, para desalojar memoria, lo primero que se hace es popear de la cola y una vez obtenido el índice, eliminar en la tabla hash, por lo que el orden de los mutexes es el de la cola y luego el de la tabla. Esto puedo ocasionar un deadlock si un hilo toma uno de los mutexes de la tabla para insertar, consultar o eliminar, y otro el de la cola para desalojar. Para solucionar esto utilizamos la función *try_lock*, de forma que si una vez popeado un elemento de la cola, el mutex correspondiente a su índice está tomado, lo vuelve a pushear y popea el siguiente. Realiza esto hasta que pueda tomar un mutex de la tabla.

1.4. Manejo de pedidos

- **Interfaz *epoll***: Una vez que los *sockets* de los clientes se obtienen a partir de la aparición de eventos en los *sockets* del servidor, éstos son agregados a la instancia *epoll* para eventos con las opciones 'EPOLLIN', 'EPOLLET' y 'EPOLLONESHOT' activadas. Con 'EPOLLIN' somos notificados únicamente cuando el *socket* del cliente está listo para realizar lecturas, es decir, que el cliente envió un mensaje y desde el servidor podemos leerlo. Con 'EPOLLET' nos aseguramos que en el servidor no se generen continuas notificaciones para un mismo mensaje de un cliente.

Por último, necesitábamos que si un mensaje del cliente llega en paquetes separados (por ejemplo debido al ruido), el servidor no sea notificado más de una vez. Para lograr esto aprovechamos la opción 'EPOLLONESHOT', donde por cada aparición de un evento, éste es desconectado hasta que el servidor lo restablezca explícitamente. Entonces, luego de que un cliente envía un pedido, un hilo trabajador lo toma, lo procesa y luego agrega nuevamente el evento a la instancia *epoll* para futuras notificaciones.

- **Procesamiento de pedidos**: Para el modo de texto y binario procesamos los mensajes siguiendo una estrategia similar, estableciendo los *sockets* de los clientes en modo no bloqueante para la lectura. De esta manera establecimos un límite de intentos de lectura de 10. Si el servidor luego de dichos intentos no recibió más bytes por parte del cliente, guarda los datos recibidos en la estructura del cliente y vuelve a entrar en modo escucha. Cuando.

  i) Para el modo texto, intentamos leer a través del *socket* del cliente un mensaje de hasta 2048 caracteres. Luego intentamos parsearlo. Si es posible, los separamos en tokens y lo procesamos con una función *text_handle*. Si no, seguimos leyendo caracteres hasta poder hacerlo o hasta que se vacíe el socket del cliente. En este caso de que el pedido no llegue completo en el socket, es decir no se llegó a leer un '$\backslash n$', el servidor guarda los datos consumidos en la estructura del cliente y se pone en modo escucha hasta que llegue el resto del mensaje.

  ii) Para el modo binario usamos la misma estrategia que para el modo texto. Es decir, intentamos leer la cantidad de bytes necesarios según cada parte de la estructura del mensaje binario. A medida que se va cargando cada parte del mensaje (la orden, luego la longitud de la clave, luego la clave, etc.) lo vamos almacenando en la estructura del cliente. Si se logra completar un mensaje entero, se procesa con una función *binary_handle*.

1.5. Estadísticas

Para llevar el contador de *KEYs*, *PUTs*, *GETs*, y *DELs* del servidor decidimos crear una estructura *"stat"* con cada variable para cada hilo trabajador y una general que la llamamos *"gStat"*. A medida que los clientes van mandando las distintas peticiones, cada hilo va modificando su respectivo *stat*. Cuando alguien pide *STATS*, el trabajador que lo toma se encarga de sumar todos *stats* de todos los otros hilos y los carga en *gStat*, el cual luego le manda a dicho cliente. De esta manera logramos que el manejo de la concurrencia sea únicamente cuando alguien pide dicha instrucción y no cada vez que tenemos que incrementar un contador.

2. Cliente

2.1. Funcionamiento general

La interfaz de cliente implementada en un módulo de *Erlang* consiste en seis funciones: *start/1*, *monit/1*, *cli/1*, *put/2*, *get/1*, *del/1* y *stats/0*.
La primera es la encargada encender la bandera para el manejo de errores y crear el hilo que ejecutará *monit/1*. Una vez creado el hilo, *monit/1* establece la conexión con el servidor, crea el hilo que funcionará como intermediario entre éste y el usuario, y lo registra como *client*. Luego, se mantiene a la espera de mensajes de error. Cuando uno llega, como se corta la comunicación con el servidor, vuelve a llamarse para repetir el proceso.
El hilo que ejecuta *client/1* hace de intermediario entre el usuario que ejecuta funciones desde la terminal y la memoria caché, ya que es el único que conoce el puerto por el que está conectado el servidor. *cli* consiste en un bucle infinito que espera un mensaje para saber qué comando y con qué información enviarle al servidor, luego, la codifica en binario, la envía y le replica la respuesta que obtenga al hilo que le envió el primer mensaje.
Las otras cuatro funciones sirven para enviar el comando deseado a la memoria caché. Las cuatro toman tantos argumentos como se necesite para el comando que envían y esperan alguna respuesta. En caso de ser una respuesta sin errores, se muestra el átomo ok, o una tupla con dicho átomo y la respuesta del servidor.
Para utilizarlo se debe compilar el archivo y ejecutar la función *start/1* que toma como argumento la dirección IP/*hostname*. Una vez establecida la conexión con el servidor, ya se pueden utilizar las funciones para enviar comandos.

2.2. Manejo de bits y envío de mensajes

En la consigna del trabajo se pide que el primer byte se reserve para la instrucción, los siguientes cuatro para el tamaño de la entrada en *big-endian* y el resto de bytes para la entrada en binario. Como *Erlang* internamente agrega una cabecera al convertir información de algún tipo de dato a binario (con la función *term_to_binary/1*), el tamaño real de la entrada es algo menor (entre dos y cuatro bytes, dependiendo del tipo de dato de la entrada original) que el que termina apareciendo en los bytes reservados para el tamaño. Sin embargo, también se guarda esta información ya que *Erlang* la utiliza para decodificar el mensaje (con *binary_to_term/1*) cuando se la solicite en el futuro. Un caso especial es el de los strings que se traducen directamente a binario, es decir, sin la cabecera y datos adicionales que puede llegar a agregar *Erlang*.