Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Sistemas Operativos I

Trabajo Práctico Final: Mini Memcached

Bacci Mateo, Giulianelli Sebastian

Rosario, Santa Fe, Argentina

6 de septiembre 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, todo el sistema se mantiene bloqueado 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.
Con respecto a los hilos, el proceso principal sera el que realice todo el trabajo anterior y será el encargado de registrar los futuros clientes que se vayan conectando y los agregará a una cola de clientes listos para ser atendidos. Los hilos(habrán tantos como *cores* tenga el procesador menos el proceso principal que ya mencionamos) esperarán a que se agreguen nuevos clientes a la cola. Una vez que un cliente es agregado, alguno de estos hilos (llamados "trabajadores") lo sacará de la cola y procesara su pedido.
Una ventaja de esta implementación es que luego de procesar un pedido de un cliente y responderle, ese trabajador ya estará disponible para atender a otro. De esta manera, si ese mismo cliente vuelve a mandar un mensaje, volvería a agregarse a la cola y lo atenderá (probablemente) un trabajador distinto. De esta manera maximizamos la paralelización aprovechando la disponibilidad de los hilos trabajadores.
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.

1.2. Estructuras de datos

En el servidor se llevaron a cabo tres 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 distintos: I) controlar el flujo de clientes, y II) mantener un registro del orden en el que se guardan los datos en la tabla hash.

    i) A medida que los clientes registrados envían mensajes al servidor, se van agregando a una cola para ser atendidos. Los trabajadores toman clientes que estén en la cola y los atienden. La elección de una cola en vez de, por ejemplo, una pila, es para respetar el orden en el que llegan sus solicitudes al servidor y además, hacerlo de una manera eficiente, ya que tanto la inserción como la eliminación se realizan en tiempo constante.

ii) Debido a la política de desalojo elegida (explicada en el siguiente apartado), se necesitó conocer el orden en el que los pares eran ingresados por los clientes, de modo de eliminar los agregados/modificados hace más tiempo. Para no tener información repetida, sus nodos en vez de almacenar los pares claves valor, almacenan el índice en el que dicho par se encuentra en la tabla, y la posición de la lista dentro de esa casilla. Se utilizó una implementación concurrente para que sólo la utilice un trabajador a la vez.

- **Tabla Hash**: Guarda los pares clave valor ingresados por los clientes. Internamente utiliza listas enlazadas simples 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.

- **Lista enlazada simple**: Utilizada para resolver las colisiones de la tabla hash. Como en la cola de desalojo se almacenan la posición de un elemento en la tabla hash y en la lista, eliminar un elemento implicaría cambiarle la posición a todos sus nodos siguientes, y para eso habría que recorrer la cola completa buscando los valores que estén en esa posición de la tabla y estén en posiciones mayores que el nodo eliminado dentro de la lista. Para evitar esto, eliminar un elemento consiste en dejar su nodo vacío sin modificar la estructura de la lista, para que cuando se quiera insertar un nuevo elemento, use ese mismo nodo y no cree uno al final.

1.3. Política de desalojo

La política de desalojo elegida fue *LRU (least recently used)* ya que al eliminar los elementos guardados 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. La idea principal es que al momento de necesitar más memoria para guardar nuevos datos, se *"popeen"* o eliminen los últimos elementos, que en definitiva son los más antiguos (por ser una estructura *FIFO*), y a medida que se pueda agregar nuevos al principio de la misma.

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, estos 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 esta 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 mas de una vez. Para lograr esto aprovechamos la opción 'EPOLLONESHOT' donde por cada aparición de un evento, este 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 y un tiempo de espera en microsegundos donde si un mensaje no puede ser completado luego de haber intentado leerlo dicha cantidad de veces y esperado un tiempo mínimo, el servidor lo desconecta. De esta forma, mientras mayor es dicho límite, mejor va a ser el procesamiento de los mensajes. Pero al mismo tiempo va a un poco mas ineficiente ya que, en casos extremos, puede haber pequeños momentos de *"busy waiting"*. Por ejemplo cuando se arma un mensaje en modo binario erróneamente por parte del cliente y las longitudes prefijadas son mayores que la de los datos que le corresponden, el servidor intentará leer bytes que nunca llegarán alcanzando el límite de intentos.

  i) Para el modo texto, intentamos leer a través del *socket* del cliente hasta 2048 caracteres. Si se supera la cantidad de intentos de lectura máxima, lo desconectamos ya sus mensajes no se pueden procesar correctamente. Si en cambio, se pueden leer un mensaje, 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 no se puedan leer mas mensajes.

  ii) Para el modo binario usamos la misma estrategia que para el modo texto. Es decir, intentamos leer la cantidad de bytes necesarios y si no es posible leer, se va incrementando un contador de intentos de lectura hasta que este alcance un límite máximo establecido. Si la lectura es exitosa, se separa todo en diferentes buffers y 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, la codifican de binario a *string* para mostrarla por pantalla.
Para utilizar 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 de algún 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.