Informe Trabajo Práctico Final

Juan Bautista Figueredo, Bautista José Peirone

2023

Sistemas Operativos 1

Contents

1 INTRODUCCIÓN

Como trabajo práctico final de la materia, se requiere implementar una **memcached**, es decir, un sistema de memoria caché con pares Clave-Valor accesibles por la red. Además, se debió implementar una interfaz en erlang para acceder a la memcached.

El siguiente informe tiene como propósito explicar y detallar las decisiones tomadas a la hora de implementar la memcached.

Para compilar y correr el programa se puede referir al archivo `README.md` en el codigo fuente.

2 MEMCACHED

2.1 Interfaz básica

La memcached funciona almacenando datos con el formato Clave-Valor, tal que a cada clave almacenada le corresponde un único valor. La memcached puede ser accesible en dos modos distintos, ambos con distintos protocolos: El modo de texto y el modo binario.
La memcache ofrece como interfaz al usuario 4 operaciones:

- **PUT** C V: La operación PUT agrega el valor V a la memcache, con la clave C. Realizar un PUT sobre una clave C ya existente, reemplazará el valor anterior por el nuevo valor V.
- **GET** C: Utilizando GET, el usuario puede conseguir el valor asociado a la clave C almacenado en la memcached.
- **DEL** C: Mediante DEL, se puede eliminar de la memcache el par Clave-Valor relacionado con la clave C.
- **STATS** : La función STATS provee al usuario información sobre las estadisticas actuales de la memcache, es decir la cantidad de operaciones realizadas y cuantos elementos se encuentran en la memcache.

2.2 Estructura principal

Para representar la memcache se implementaron una serie de estructuras de datos. Principalmente y a grandes rasgos, la memcache esta compuesta por una tabla hash y una cola de prioridad.
Para poder combinar estas estructuras y llevar a cuenta de distintos estados e información

importante de la memcache (como las estadísticas de uso o el tamaño) se encapsuló la estructura general de la memcache en la estructura `Cache`:

```
struct _Cache {
  List *buckets;
  LRUQueue queue;
  struct Stats text_stats, bin_stats;
  pthread_rwlock_t *row_locks;
  pthread_mutex_t ts_lock, bs_lock;
  uint32_t nregions, size;
};
```

Listing 1: Definición de estructura Cache

En esta estructura se almacena la información vital para la memcache, los parámetros `buckets`, `nregions` y `size` corresponden a la estructura principal de la tabla hash, mientras que `queue` es la cola de prioridad relacionada con la política de desalojo implementada (LRU). Las estructuras tipo `Stats` almacenan la información de uso de la memcache para cada modo por separado. Por último, la estructura `Cache` también posee la información sobre los "locks" implementados para proteger a la memcache de problemas de concurrencia.

2.3 Tabla Hash

2.3.1 Implementación

Como se comentó anteriormente, la memcache contiene una tabla hash que se utiliza para obtener una rápida búsqueda de las claves.
Para el manejo de las colisiones se optó por un enfoque de encadenamiento, principalmente por la facilidad de implementar concurrencia en esta. Esto resultó en que la estructura principal de la tabla sea un arreglo de listas doblemente enlazadas, este arreglo se encuentra en la estructura `Cache`, bajo el nombre de `buckets`. El tamaño de la tabla esta dado por la variable `size`. Cada elemento del arreglo, es decir cada lista doblemente enlazada de la tabla hash fue implementado con un nodo centinela, facilitando la actualización de cada lista.
Los nodos de las listas, `LLNode`, están compuestos por dos atributos, los datos a almacenar y la prioridad del nodo. Los datos se almacenan encapsulados en una estructura `Data`, que contiene información sobre el modo del dato (binario o texto), la clave, la longitud de la clave y el valor y la longitud del valor. La prioridad se consigue de la cola de prioridad y esta relacionada con la política de desalojo, que se verá más adelante.

```
typedef struct {
  char *key, *val;
```

```
3   uint64_t klen, vlen;
4   char mode;
5 } Data;
6
7 struct _LLNode {
8   Data data;
9   LRUNode lru_priority;
10  struct _LLNode *prev, *next;
11 };
```

Listing 2: Estructura Data y nodos de listas enlazadas

Para la función de hash en si se utilizó una función hash multiplicativa, en particular, se tomo la propuesta por Kernigan y Ritchie (The C programming language, second edition), y se hashea según el valor ingresado.

2.3.2 Protección de la tabla hash

La tabla hash de la memcache es una estructura que se accede constantemente por distintos hilos, luego cada fila es una **región crítica** susceptible a problemas de **race condition**. Bloquear toda la tabla resulta en aplicar **exclusión mutua** sobre una región demasiado amplia, mientras que dedicar un sistema de exclusión mutua para cada fila implicaría un gasto elevado de recursos. Por tanto, se tomó la decisión de particionar la tabla en regiones.

La idea es que cada región abarque una porción arbitraria de la tabla, y que cada una cuente con su lock. Estos locks fueron implementados con los que provee POSIX para lectura/escritura(`pthread_rwlock_t`). De este modo, dos hilos que quieran acceder a la tabla en regiones diferentes pueden hacerlo sin problema de concurrencia, sin embargo, dos hilos que buscan escribir en la misma región, se verán obligados a esperar a que uno termine para que el otro pueda comenzar. Cabe destacar que al ser un lock de lectura/escritura, un hilo puede acceder a leer sobre una región si no hay ningún hilo escribiendo, lo que resulta en que múltiples hilos puedan acceder a leer al mismo tiempo.

2.4 Política de desalojo

La memcache cuenta con una limitación en la cantidad memoria que puede ser asignada para su funcionamiento, esto implica que puede darse que luego de un uso intensivo, se quede sin memoria para guardar nuevos pares Clave-Valor. Una solución es implementar una política de desalojo en la cual al quedarse sin memoria, se desalojen datos almacenados (se eliminen de la memcache) para dejar lugar a los nuevos datos.

2.4.1 Política LRU

Para este trabajo se optó por una política **LRU** (Least Recently Used) en la cual cada nodo de la tabla tiene una prioridad que se modifica según el orden en el cual se va accediendo a estos. Esto implica que al momento de desalojar datos, se eligen los nodos que hayan sido accedidos por última vez hace mucho tiempo.

2.4.2 Cola de prioridad

La política **LRU** es implementada en nuestra memcache mediante el uso de una cola de prioridad `LRUQueue`. Esta está implementada mediante una lista doblemente enlazada y cuenta con punteros al primer y último elemento.
La cola esta compuesta por nodos `LRUNode` que llevan información del dato que almacenan, como el índice de este en la tabla hash y el dato en sí. También se tiene información sobre el siguiente nodo y sobre el anterior.

```
1  struct _LRUNode {
2    unsigned idx;
3    List data_node;
4    struct _LRUNode *prev, *next;
5  };
6
7  struct _LRUQueue {
8    pthread_mutex_t lock;
9    struct _LRUNode *first, *last;
10 };
```

Listing 3: Estructura LRUNode y Estructura LRUQueue

Debido a la política **LRU**, el simple hecho de acceder a un dato (con un GET, por ejemplo) es suficiente para cambiar la prioridad del nodo. Esta "prioridad" se representa en la cola con la posición en la que cada nodo se encuentra, un nodo con prioridad baja (es decir, que es mas propenso a ser desalojado) se encontrará en las cercanías del comienzo de la cola, mientras que los nodos accedidos recientemente estarán ubicados en el final. De este modo, actualizar la prioridad de un nodo se reduce a quitar el nodo de su posición actual y llevarlo al final de la cola. Desalojar un elemento, sin embargo, no es tan simple, ya que el desalojo (se verá mas adelante) no depende de las operaciones de la interfaz y se produce automáticamente, esto puede llegar a provocar que el nodo que actualmente tiene la menor prioridad (el primer nodo de la cola) esté siendo utilizado por otro hilo y no se ha actualizado su prioridad todavía. Por esto el desalojo no siempre elimina el de menor prioridad sino que se desaloja el nodo con menor prioridad que no este siendo utilizado por otro hilo.
A diferencia de la tabla hash, la cola no puede ser fraccionada en regiones ya que al momento de realizar una actualización de prioridad, se puede quitar un elemento de cualquier lado de

la cola y llevarlo al final. Esto provoca que la única manera de proteger la estructura contra problemas de concurrencia es con un único lock que bloquee la cola entera. Esto puede parecer poco eficiente ya que los hilos solamente pueden acceder de a uno, sin embargo, esto se compensa con la velocidad de desalojo y la simplicidad de la política.

2.4.3 Realización del desalojo

El desalojo ocurre cuando la memcache no tiene mas memoria para poder seguir agregando datos y funcionando. Esto se traduce a que al momento de pedir memoria al sistema operativo tanto para crear estructuras como para almacenar datos nuevos el sistema no puede brindarnos mas memoria para trabajar (ya que la memoria está limitada). Para capturar estos casos y realizar el correspondiente desalojo se implementó una interfaz sobre la instrucción `malloc` que nos permite identificar cuando la memcache agotó la memoria y debemos desalojar. A esta interfaz la llamamos: `dalloc`.

```
1 void* dalloc(size_t size) {
2   void* ptr = malloc(size);
3   while (!ptr && !lru_empty(cache_get_lru_queue(cache))) {
4     lru_dismiss(cache); //Desalojo de la cola LRU
5     ptr = malloc(size);
6   }
7   return ptr;
8 }
```

Listing 4: Implementación de dalloc

En pocas palabras, `dalloc` se encarga de intentar asignar una cantidad especificada de bytes para un elemento, tal y como lo hace `malloc`, sin embargo si esta asignación falla, `dalloc` intenta realizar desalojos de la cola hasta poder conseguir suficiente espacio libre para poder asignar la memoria requerida. La interfaz `dalloc` es usada en la mayoría de la memcache para conseguir un mayor control sobre la memoria utilizada por la memcache.
NOTA: Todas las operaciones de la interfaz **LRU** garantizan seguridad sobre el acceso a la cola por múltiples hilos.

2.5 Implementación de operaciones

La implementación de las operaciones básicas sobre la memcache se lograron combinando las interfaces de la tabla hash con la cola de prioridad de desalojo.

2.5.1 Operación PUT

La operación PUT puede tener 2 posibles resultados, crear un nuevo nodo Clave-Valor o actualizar un nodo Clave-Valor cambiando el Valor por el valor ingresado.
Para cualquier caso, el primer paso es verificar la existencia de la clave en la memcache. Si esta no se encuentra, debemos crear un nuevo nodo Clave-Valor. Para esto primero se crea el nodo y se inserta en la tabla hash, consiguiendo el índice en según la llave. Con este índice se crea un nodo en la cola **LRU** y se consigue la prioridad del nodo. Como último paso se actualizan las estadísticas de la memcache y se agrega una clave mas.
En el caso de que la clave ya se encuentre, simplemente se accede al nodo y se modifica la estructura que contiene la información (estructura `Data`). Luego de esto se actualiza la prioridad mediante funciones de la interfaz **LRU**.
En ambos casos se registra en las estadísticas de la memcache el uso de la operación PUT.
Si la operación fue realizada correctamente, el cliente recibe el mensaje de confirmación "OK", sino se devuelve el código de error correspondiente.

2.5.2 Operación GET

La operación GET esta implementada de una manera relativamente sencilla. Como primer paso se busca en la memcache si la clave a buscar existe. Si lo hace, se copia el valor en un buffer y se retorna. Esta copia es implementada con el propósito de proteger la respuesta de la operación a posibles modificaciones del valor antes de que la respuesta haya sido entregada completamente al cliente. Se incrementa en las estadísticas la cantidad de operaciones GET utilizadas.
En caso del correcto funcionamiento, se envía al cliente el mensaje "OK valor", siendo "valor" el valor correspondiente a la clave recibida. En caso contrario, se retorna un código de error.

2.5.3 Operación DEL

La implementación de la operación DEL no es muy diferente a la implementación de la operación GET. Se verifica que la clave ingresada pertenezca a la memcache y, si es así se procede a eliminar el nodo de la tabla hash. Luego, se remueve el nodo de la cola **LRU** y se liberan los nodos.
Se actualizan los valores de las estadísticas.
Luego de una operación exitosa, el cliente recibe el mensaje "OK". Si no tuvo exito se envia un código de error.

2.5.4 Operación STATS

La operación STATS tiene un comportamiento diferente al resto de las operaciones. Como se puede observar en la estructura `Cache` se almacenan dos estructuras del tipo `Stats`, una

llamada `text_stats` y otra llamada `bin_stats`.

```
struct Stats {
  uint64_t get, put, del, keys;
};
```

Listing 5: Estructura Stats

Estas simples estructuras `Stats` sirven para almacenar las estadísticas de la memcache para cada modo de uso. En cada llamada a las demás operaciones se incrementa el contador correspondiente a cada operación.
En la llamada a la operación STATS, se retornan los valores almacenados en la estructura correspondiente al modo en el cual se llamo a la operación. La respuesta que el cliente recibe tiene la siguiente forma:
"OK PUTS=nputs DELS=ndels GETS=ngets KEYS=nkeys"

2.5.5 Códigos de error

Las operaciones sobre la memcache no siempre resultan exitosas, pueden surgir errores por distintas razones, imposibilitando que la operación se lleve a cabo. Cuando esto sucede, el cliente recibe un código de error que especifica el tipo de error que ocurrió. Estos códigos son:

- **EINVALID:** El comando ingresado no es válido.
- **ENOTFOUND:** La clave a buscar no ha sido encontrada en la memcache.
- **EBINARY:** Sucede cuando un cliente en modo texto intenta acceder a un valor almacenado en modo binario.
- **EBIG:** La entrada del modo texto supera los 2048 caracteres.
- **EUNK:** El comando ingresado no existe.
- **EOOM:** Este error ocurre cuando la memcache ocupa toda la memoria disponible y no es posible hacer desalojos.

2.6 Protocolos de Entrada / Salida

Para la comunicación entre el cliente y el servidor (memcache) se pactan distintos protocolos sobre la forma de los mensaje que cada uno debe mandar para mantener una comunicación efectiva.
Como explicará mas adelante, la comunicación se logrará utilizando sockets. Esto implica que si el cliente quiere mandar un mensaje al servidor, debe escribir su mensaje en el socket

y el servidor debe leer el mensaje del socket para procesarlo y viceversa. Esta comunicación no siempre es perfecta, por lo que puede aparecer el efecto del **ruido**, que puede alterar la velocidad en la que se escribirá la información en el socket (Al ser comunicación por protocolo TCP se asegura que los bits enviados llegaran en orden). La alteración causa que la lectura al socket por parte del servidor se dificulte y surge la posibilidad de leer mensajes que todavía no están completos.
De aquí, surge la necesidad de crear una estructura `ClientData` que será única para cada cliente (no para cada hilo) y contendrá información necesaria para mantener un contexto del estado de la comunicación entre el cliente y el servidor.

```
1 struct ClientData {
2   char *buffer; //buffer del mensaje
3   long buf_size; //cantidad de bytes dedicados al buffer
4   long current_idx; //longitud del mensaje actual almacenado en el
       buffer
5   int client_fd; //file descriptor del socket para comunicacion
6   int mode; //modo del cliente
7 };
```

Listing 6: Estructura ClientData

De este modo, si se llega a producir ruido en la comunicación entre el cliente y el servidor, la estructura del cliente almacenará el mensaje incompleto y a medida que los fragmentos del mensaje van llegando se agregan a la estructura hasta construir un mensaje válido.
Esta construcción de un mensaje válido cambia según el modo en el cual el cliente se conecta con el servidor (modo texto o binario).

2.6.1 Protocolo de texto

Los mensajes de modo texto se caracterizan por la existencia de un delimitador '\n' al final de cada mensaje. De este modo, un posible mensaje del cliente puede tener la siguiente forma:

$$\texttt{GET} \quad clave\backslash n$$

Del mismo modo, una respuesta del servidor puede ser:

$$\texttt{OK} \quad valor\backslash n$$

Esta convención facilita el trabajo a la hora del parseo de la entrada válida. Si se leyó un mensaje del socket en modo texto pero no contiene el carácter '\n' entonces el mensaje no esta completo y por lo tanto se almacena el mensaje actual en la estructura `ClientData` y se vuelve a esperar a que el resto del mensaje llegue al socket.
En caso de que el mensaje si contiene el carácter delimitador, se puede procesar correctamente

el mensaje y determinar que operación se debe ejecutar.
Si la operación no es válida, o el mensaje recibido tiene mas de 2048 caracteres, se devuelve un código de error.

2.6.2 Protocolo binario

Contrariamente al protocolo de texto, los mensajes en modo binario no contienen ningún tipo de delimitador que marquen el final de un mensaje. Como alternativa, el protocolo binario especifica que antes de cada argumento se incluyan 4 bytes que marquen la longitud (en bytes) del argumento a consumir.Ademas, las operaciones y códigos de error se encuentran códificadas de modo que todas pueden ser representadas con un byte, en particular, el primer byte del mensaje. De esta manera, los mensajes tendrán la siguiente forma:

$$\texttt{Op} \quad Len1 \quad Arg1$$

Para parsear los mensajes en modo binario debemos controlar constantemente la cantidad de bytes que fueron leídos y almacenados en el buffer de la estructura `ClientData`. Si se pudo leer al menos el primer byte del mensaje, entonces se puede determinar la operación a realizar y por lo tanto, la cantidad de argumentos a consumir. Si la operación es STATS, no se consume ningun argumento, si en cambio la operación es GET o DEL, se esperará un solo argumento (la clave a buscar), Como último caso, la operación PUT requerirá que se consuman 2 argumentos (La clave y valor). Luego, según la cantidad de argumentos, se procede a intentar leer los siguientes 4 bytes que marcan la longitud. Si no hay suficientes bytes, el mensaje no esta completo y se vuelve a esperar a que el resto mensaje llegue al socket para luego volver a procesar el mensaje completo. Si los hay, se consume la longitud n (La longitud esta almacenada en modo big-endian, por lo que para leerla correctamente se debe transformar su endianness), y se intenta nuevamente consumir ahora n bytes del buffer. Si no los hay, el mensaje no esta completo y el comportamiento es el mismo que al intentar leer la longitud. En el caso que si se pueda consumir esa cantidad de bytes, se verifica del mismo modo que se puedan leer el resto de los argumentos.
Una vez que todos los argumentos hayan sido leídos correctamente se almacenan y se procede a ejecutar la operación especificada.
NOTA: La no existencia del delimitador en el protocolo binario no permite distinguir entre un mensaje mal formado y un mensaje incompleto (que el resto del mensaje todavía no llego).

3 MANEJO DE CONEXIONES

Un punto clave de la memcache es lograr que esta sea accesible mediante la red por múltiples clientes de manera simultánea. Para esto, se utilizan múltiples hilos, sin embargo, no se utiliza un hilo por cada conexión al servidor, sino que solamente se utiliza una cantidad fija de hilos, en particular, la cantidad de hilos de hardware del sistema en el cual se esta corriendo la memcache. De esta manera, un hilo atenderá a mas de un cliente.

3.1 Utilización de Epoll

Para administrar la asignación de clientes a hilos se decidió usar la interfaz provista Linux, Epoll. Epoll nos permite multiplexar la entrada y salida de varios sockets, tanto los de escucha como los sockets de las conexiones TCP establecidas con los clientes. De esta forma, un hilo es despertado solo cuando se puede realizar una operacion sobre alguno de estos sockets, en especial nos interesan operaciones de lectura, luego usamos los eventos `EPOLLIN` de epoll.
Sockets de escucha: Los sockets de escucha son dos, aquel para el modo texto y el de binario. La idea que implementamos consiste en un hilo que es despertado por actividad en un socket de escucha debe aceptar todas las conexiones encoladas en este socket. Para esto, configuramos ambos sockets de escucha como no bloqueantes y el evento de escucha en epoll es de modo edge-triggered (`EPOLLET`). Este modo hará que epoll solo genere un evento cuando no había clientes encolados y de pronto llegan conexiones, y con la bandera `EPOLLEXCLUSIVE` nos aseguramos que un único hilo se despierta para encargarse del trabajo.
Sockets de clientes: Para evitar race conditions de sockets o que multiples hilos se despierten por distintos eventos de uno, configuramos epoll para usar el modo One Shot (`EPOLLONESHOT`). Con este modo, cuando se termina de atender la entrada recibida de un cliente, se vuelve a añadir el socket a epoll para que vuelva a generar un evento cuando haya nueva información entrante.

3.2 Conexión a puertos privilegiados

Para la conexión entre el cliente y el servidor, se denominan ciertos puertos para cada modo de conexión. De manera por defecto, los puertos son el 8888 para el modo texto y el puerto 8889.
Sin embargo la memcache permite al usuario conectarse a los puertos privilegiados 888 para modo texto y 889 para modo binario. Estos puertos privilegiados están protegidos por el sistema y no pueden ser accedidos por cualquier usuario ya que un mal uso de ellos puede causar problemas en el sistema operativo. En esta implementación, el usuario puede acceder a la conexión a los puertos privilegiados si ejecuta la memcache con privilegios "sudo".
Para evitar problemas con el mal uso de los puertos, los privilegios de sudo solamente se usan para realizar la conexión con los puertos, luego de realizar la conexión correctamente, se bajan los privilegios y se pierde el poder de sudo.
Esto se logró utilizando el `uid` (user id) del proceso que ejecuta la memcache para determinar si se cuenta o no con los permisos de sudo (es decir, si el `uid` es 0). Si se ejecutó con sudo y se pudo conectar a los puertos privilegiados, solo queda bajar los privilegios, esto se logra cambiando el `uid` al original del usuario que llamo a sudo. La variable de entorno `SUDO_UID` mantiene un registro de este `uid`, por lo que simplemente cambiando el `uid` actual por el almacenado en `SUDO_UID` es suficiente para eliminar los privilegios y poder trabajar de forma segura en los puertos. La función `getenv()` permite recuperar el valor de la variable de entorno.
NOTA: En el usuario `root`, la variable de entorno `SUDO_UID` no existe, lo que implica que

no se podrán bajar los privilegios. Por esto se implementó que el programa no puede ser ejecutado desde `root` e intentar hacerlo no tendrá ningún resultado.

4 INTERFAZ EN ERLANG

Se construyó en erlang una interfaz de la memcache que facilita el uso de esta y brinda una capa de abstracción por sobre el código fuente del programa en si.
La interfaz exporta el módulo `client` que brinda al usuario 6 funciones para lograr una comunicación óptima con la memcache. Estas son:

- **start/2:** Inicializa y conecta el cliente con el servidor.
- **put/3:** Agrega o modifica un par Clave-Valor a la memcache.
- **get/2:** Busca el Valor relacionado a la Clave a buscar.
- **del/2:** Elimina el par Clave-Valor de la memcache, recibe la Clave del par a eliminar.
- **stats/1:** Muestra las estadísticas de uso de la memcache.
- **exit/1:** Finaliza la conexión con la memcache y cierra el cliente.

El cliente en erlang provee una interfaz en la que el usuario se comunica con la memcache utilizando texto, sin embargo, el cliente realiza una conexión en modo binario.

4.1 Implementación de la interfaz

4.1.1 Manejo de hilos en erlang

La estructura principal de la interfaz en erlang se basa en la existencia de un hilo trabajador que se encargará de manejar la conexión entre el usuario y el servidor. Luego de haber aceptado la conexión, se invoca un hilo que permanecerá en constante escucha a pedidos de las distintas operaciones. Es por esto que las operaciones deben utilizar el identificador de hilo `Id` para saber qué cliente es el que esta invocando a la función.

4.1.2 Operación start/2

La operación `start/2` se encarga de conectar al cliente con el servidor a través del puerto dedicado a la comunicación en modo binario. La operación toma dos argumentos, el primero es la dirección IP de el lugar en donde se esta ejecutando el servidor. El segundo argumento

es el puerto a conectar el cliente. Puede ser 8889 si se quiere realizar una conexion a un servidor común, u 889 si se quiere conectar a un servidor corriendo en los puertos privilegiados.

En cuanto al modo de conexión al puerto, se implementó una conexión en modo pasivo, logrando un mayor control sobre las información que se lee del socket cuando el servidor manda una respuesta.
La operación crea un actor que se encargará de enviar y recibir los mensajes al y desde el servidor. La función retorna el identificador de este hilo, que deberá ser utilizado en las demás operaciones de la interfaz.
Una llamada correcta a la operación tiene la siguiente forma:

```
Id = client:start(Ip, Port).
```

4.1.3 Operación put/3

La operación `put/3` la implementación de la función toma como argumentos, el identificador del hilo `Id`, y los elementos `clave` y `valor`. La función toma estos valores y los convierte a una cadena binaria. Esta transformación es posible gracias al uso de la función `term_to_binary()` de erlang, que permite a erlang codificar distintos tipos de datos a una cadena de bits, agregando headers necesarios para la propia decodificación de erlang. Finalmente se contruye una cadena de bits que contiene tanto el código de la operación `PUT` (11), la longitud del primer argumento (en big endian, y ocupando 4 bytes), la cadena de bits resultante de la transformación de la clave, y análogamente, se incluye la información para el valor.
Un ejemplo de llamada a la operación es:

```
client:put(Id, Clave, Valor).
```

4.1.4 Operación get/2 y del/2

La operación `get/2` y `del/2` no varían demasiado en implementación. Ambas funciones esperan dos valores, el primero es el identificador de hilo `Id`, mientras que el segundo es la `clave` del elemento a buscar. Del mismo modo que con la función `put/2`, se hace uso de la función `term_to_binary()` para convertir la clave ingresada en una cadena de bits. Luego, se completa la cadena incluyendo el código de operación (12 para el `DEL` y 13 para el `GET`) y el largo (big endian y ocupando 4 bytes) de la clave.
La funciones se invocan de la siguiente manera:

```
client:get(Id, Clave).

client:del(Id, Clave).
```

4.1.5 Operación stats/1

`stats/1`, a diferencia del resto de las operaciones sobre la memcache, no requiere de ningún argumento además del identificador del hilo. Esta función envía solamente la cadena de bits que codifica a la operación `STATS` (21). La función se debe llamar de la siguiente manera:

```
client:stats(Id).
```

4.1.6 Operación exit/1

El funcionamiento de `exit/1` solamente se basa en realizar una desconexión del socket y solamente recibe el identificador de hilo `Id`. Una llamada exitosa de la operación retorna el átomo `exit`. Una ejemplo de invocación correcta de la función es:

```
client:exit(Id).
```

5 TESTING

Para testear el servidor utilizamos distintos clientes variando distintos parametros, comandos y cantidad de clientes. En concreto, utilizamos:

- test_cliente.c
- test_server.c
- cliente_binario.c
- netcat

netcat lo utilizamos para probar el parser de texto y todas las funciones relacionadas al handling de clientes en este modo, probando los distintos comandos y observando los resultados que generaba el programa. De manera análoga, para el modo binario utilizamos cliente_binario.c (el provisto por la cátedra).
Para probar más a fondo el rendimiento del programa, junto con la concurrencia de las estructuras de datos y el ruido en las conexiones, lo que hicimos fue probar con grandes cantidades de clientes. Los archivos test_client.c y test_server.erl se usaron para esta misma tarea.

6 Bibliografía

- Operating Systems Concepts (quinta edición) - Silberschatz, Galvin.
- Erlang programming - Cesarini, Thompson.
- Epoll: https://idea.popcount.org/2017-02-20-epoll-is-fundamentally-broken-12/
- The Linux Programming Interface - Kerrisk.
- Slides de la cátedra.