Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Sistemas Operativos I

R-322

Informe Trabajo Práctico Final

Mini Memcached Distribuido

Alumnos

Rassi, Octavio `R-4549/7`
Sferco, Martín `S-5656/1`

15 de Abril, 2025

1. Leeme.

Para compilar el servidor y el cliente, basta con correr el comando `make` desde el directorio `Memcached`.

Ahora, podemos correr una instancia del cliente utilizando el comando `make run-client`, que es un atajo. Para correrlo sin utilizar `make`, podemos movernos a `Memcached/src/client/` y correr el intérprete de Erlang con el comando `erl`. Luego, como `make` ha compilado el cliente, ya podremos acceder a sus funciones.

Por último, para poner en funcionamiento un servidor Memcached desde el directorio base `Memcached`, podemos correr `./bin/server -h`, donde el uso de la `-h` generará un mensaje indicando como pasar los parámetros al servidor.

1.1. Lanzando nuestros primeros servidores y cliente.

Una manera rápida de probar el programa es compilarlo con `make` desde el directorio `Memcached` y levantar dos servidores locales en los puertos 8000 y 9000 con los comandos

```
./bin/server -p 8000
./bin/server -p 9000
```

donde, por omisión de las banderas `-m` y `-t`, se establece la cantidad de memoria y cantidad de hilos a sus valores por defecto, que son los máximos disponibles.

Luego, corriendo `make run-client` estaremos dentro del intérprete de Erlang, y utilizando la función `client:startDefault/0` el cliente se conectará a los dos servidores creados sin necesidad de pasar su dirección IP y puerto, como si lo requiere la función `client:start/1`.

Ahora, podremos probar las funcionalidades del cliente con la interfaz dada por

`client:put/3`,	que toma un par clave-valor y lo inserta,
`client:get/2`,	que toma una clave y busca su valor asociado,
`client:del/2`,	que toma una clave y la elimina del servidor,
`client:stats/1`,	que obtiene información sobre los servidores, y
`client:status/1`,	que obtiene información sobre la distribución de las claves almacenadas.

Observación. Todas las funciones toman como primer argumento el PID del cliente que las invoca, que se obtiene al llamar a `start/1`.

2. Servidor

2.1. Diseño.

La idea general detrás del diseño de nuestra `Memcached` es separar el problema de almacenar los pares clave-valor del problema de llevar las prioridades o la frecuencia de acceso a cada par. Por un lado, tendremos una *least-recently used queue* o `LRUQueue`, que ordena a los pares clave-valor según su frecuencia de acceso, y por otro lado tendremos los `HashNode`, que son los que realmente llevan los pares y conforman una tabla hash.

Para unir estas dos partes del problema general utilizamos la estructura `Cache`. La estructura contiene a la tabla hash, formada por los `HashNode` y sus mecanismos de sincronización, y a la `LRUQueue`, que le da un orden a esos nodos. De esta manera, las operaciones que realmente queremos ofrecer al usuario, que son las de `PUT`, `GET`, `DEL`, y `STATS`, estarán definidas en la interfaz de `Cache`.

Para poder implementar estas operaciones, la `Cache` hará uso de las interfaces que ofrecen los `HashNode` y la `LRUQueue`, cumpliendo el rol de nexo entre estas dos estructuras aparentemente separadas.

2.1.1. Los LRUNode y HashNode.

La `LRUQueue` que definimos anteriormente se compone por nodos que llamamos `LRUNode`. Su estructura es la siguiente:

Definición de la estructura LRUNode

```
1 struct LRUNode {
2     struct LRUNode* prev;
3     struct LRUNode* next;
4
5     HashNode hash_node;
6     unsigned int bucket_number;
7 };
```

Notemos que se trata de un nodo de una lista doblemente enlazada, pero en vez de almacenar un valor, tenemos una referencia a un `HashNode` y un *bucket number*.

Esta referencia a un `HashNode` no es mas que el puntero al nodo de la tabla hash asociado a este `LRUNode`. Es decir, si estamos recorriendo la `LRUQueue` y queremos saber a que nodo esta asociada una posición dada en la cola, podemos seguir esta referencia para obtener la estructura deseada. La utilidad del *bucket number* la explicaremos mas adelante, pues se relaciona con como la `Cache` liberará memoria cuando esta sea insuficiente.

Del lado de la tabla hash, la estructura de sus nodos es la siguiente:

Definición de la estructura HashNode

```
1 struct HashNode {
2     void* key;
3     void* val;
4     size_t key_size, val_size;
5
6     struct HashNode* prev;
7     struct HashNode* next;
8     struct LRUNode* prio;
9 };
```

Nuevamente, tenemos referencias al nodo anterior y siguiente, formando una lista doblemente enlazada. En este caso si almacenamos el par clave-valor en los campos `key` y `val` respectivamente, que como podemos ver son punteros a las direcciones de memoria que son asignadas dinámicamente. Tambien almacenamos el tamaño de cada uno, pues es necesario para el momento de responder a un pedido de un usuario.

Al igual que en la estructura anterior, mantenemos una referencia a un `LRUNode` en cada `HashNode`. Esto vincula al nodo con su posición en la *least-recently used queue*. Notemos que desde cada nodo es posible encontrar su nodo asociado de forma que, si `L` es un `LRUNode` con `HashNode` asociado `H`, entonces `H` tiene como `LRUNode` asociado a `L`, y viceversa.

2.1.2. Estructura LRUQueue

Veamos ahora como se define una `LRUQueue`, la estructura conformada por los `LRUNode` que definimos antes.

Definición de la estructura LRUQueue

```
1 struct LRUQueue {
2     LRUNode most_recent;
3     LRUNode least_recent;
4     pthread_mutex_t* lock;
5 };
```

Vemos que es una estructura simple para una cola doblemente enlazada con punteros al inicio y al final, que designamos con los nombres `most_recent` y `least_recent`, en referencia a cuan recientemente fueron accedidos los nodos en cuestión.

Adicionalmente, la estructura cuenta con un mecanismo de sincronización que es un `pthread_mutex_t`. Este *lock* será utilizado para garantizar la atomicidad de las operaciones, y funciona como un bloqueo global. Esta es una forma mas simple de sincronización que la que implementamos para la tabla hash, pues se tiene un único lock para toda la `LRUQueue`.

La estructura `LRUQueue` induce un orden en la tabla hash, donde el nodo al que apunta `least_recent` es el menos recientemente accedido y, avanzando a través de los punteros `next`, llegamos a `most_recent`, que es el mas recientemente accedido. De esta manera, los nodos mas cercanos a `least_recent` en la cola son los mas propensos a ser expulsados cuando la `Memcached` necesite liberar memoria.

La interfaz de `LRUQueue` nos permitirá, además de las funciones básicas de creación y destrucción, insertar a un nodo como el mas reciente y reubicar a un nodo que ya pertenecía a la cola para colocarlo en la posición de mas reciente. No ubicaremos a un nodo explícitamente en las posición de `least_recent`, si no que naturalmente los nodos decaerán hacia esa posición conforme se inserten nuevos nodos o se accedan a algunos que ya formaban parte de la cola.

2.1.3. Estructura Cache.

Habiendo definido ya la `LRUQueue` estamos en condiciones de presentar la estructura `Cache`. Notemos que omitimos en este informe definir una estructura como `HashTable`, y el motivo es que esa estructura está contenida dentro de `Cache`. En un principio implementamos a la `HashTable` como una estructura separada, quedando la `Cache` definida como una estructura que contenía a una `HashTable` y a una `LRUQueue`, pero notamos que esto no resultaba práctico.

Esto viene de que las operaciones sobre la tabla hash requieren constantemente de llamados a la `LRUQueue`, por lo que la interfaz de la `HashTable` requería tomar por argumento una referencia a la *least-recently used queue* en todas sus funciones, que era algo que buscabamos evitar.

Entonces, finalmente la estructura se definió como

Definición de la estructura Cache

```
1  struct Cache {
2      // Hash
3      HashFunction         hash_function;
4      HashNode*            buckets;
5      int                  num_buckets;
6      pthread_rwlock_t**   zone_locks;
7      int                  num_zones;
8
9      // LRUQueue
10     LRUQueue   queue;
11
12     // CacheStats
13     CacheStats stats;
14 };
```

donde

- `hash_function` es la función de hash que utilizaremos para determinar a que bucket enviar cada clave.

- `buckets` es el *array* de `HashNode` que conforma la tabla hash, y tiene tamaño `num_buckets`.
- `zone_locks` es un *array* de locks de escritura y lectura, que utilizaremos para proteger a los *buckets*, y tiene tamaño `num_zones`.
- `queue` es la `LRUQueue` asociada la `Cache`.
- `stats` es una estructura que almacena las estadísticas de la Cache.

Las operaciones principales que ofrece `Cache` son `cache_create, cache_get, cache_put, cache_delete, cache_report, cache_free_up_memory,` y `cache_destroy`, cuya documentacion puede leerse en `cache.h`.

2.1.4. Tabla hash.

Como mencionabamos en la subsección anterior, la estructura de la tabla hash está integrada dentro de `Cache`. La principal decisión de diseño a tomar para este apartado fue el mecanismo de sincronización a utilizar para permitir el acceso concurrente a la tabla, pues en cuanto a lo demás se trata de una tabla hash convencional.

Para proteger el acceso a la tabla optamos por definir zonas y *buckets*. Un *bucket* no es mas que un `HashNode`, es decir, son los nodos como tal que conforman las distintas áreas donde puede almacenarse un par clave-valor. Las zonas, en cambio, definen conjuntos de *buckets* que serán protegidos por el mismo *lock*.

Una forma mas simple de proteger el acceso a los *buckets* sería mantener un único *lock* para toda la tabla, garantizando que en todo momento un solo hilo accede a cualquiera de los nodos. Esto sería algo similar a como protegimos el acceso a la `LRUQueue`. Sin embargo, para permitir un mejor desempeño cuando la `Memcached` se corre con múltiples hilos, optamos por sectorizar a la tabla hash con estas zonas. La idea es que mientras un hilo está accediendo a un *bucket*, los demás pueden acceder a cualquier otro *bucket* y no se producirán condiciones de carrera, pues no comparten recursos. Llevado al límite, podría definirse una zona distinta para cada bucket, pero esto conlleva también un gasto de memoria mayor y, en nuestra experiencia, no necesariamente implica una mejor *performance*.

En base al funcionamiento de la *Memcached* en los distintos casos de prueba que ejecutamos, decidimos que por cada hilo corriendo sobre la misma se creen 10 zonas, es decir, 10 *locks*. El problema con el que nos encontramos al definir menos zonas fue que el desalojo de memoria se ralentiza, pues su velocidad depende de la capacidad del proceso que esta desalojando memoria de obtener *locks* de zonas libres. Cuando el número de zonas es pequeño, muchos nodos se concentran en pocas zonas, e incluso existe la posibilidad de que el desalojo de memoria fracase si todos los *locks* están tomados. También definimos el tamaño de cada zona, optando por zonas con 100 *buckets* cada una.

2.1.5. Dynalloc.

En la implementación de `dynalloc` es donde recae la mayor parte del mecanismo de liberación de memoria que implementa la `Memcached`. Definida en `dynalloc.c`, la función se comporta

prácticamente como `malloc`, excepto en el caso donde la memoria disponible en la `Memcached` no es suficiente para asignar el bloque del tamaño deseado.

El prototipo de la función junto con su documentación es:

Prototipo y documentación de dynalloc

```
1 /**
2  *  @brief Asigna un bloque de memoria. De no contarse con memoria ↩
     disponible, elimina los nodos menos recientemente utilizados de la↩
      cache hasta contar con espacio suficiente.
3  *
4  * @param sz El tamanio del bloque a asignar.
5  * @param cache La cache asociada a quien invoca a esta funcion.
6  *
7  * @return Un puntero a un bloque de memoria de tama o `sz`, o NULL ↩
     si se produjo un error al desalojar o alojar memoria para ↩
     satisfacer el pedido.
8  */
9 void* dynalloc(size_t sz, Cache cache);
```

Como podemos ver, toma el tamaño del bloque de memoria a asignar, y una referencia a la estructura `Cache` desde donde se invocará a la función. La idea es que toda asignación de memoria que sea asociada al funcionamiento de la `Memcached` se realice invocando a `dynalloc` en vez de a `malloc`. Esto no se reduce solamente a cuando queremos almacenar las claves o los valores, sino que también cuando creamos la estructura de datos que almacena la información de un cliente, o cuando creamos un nuevo `LRUNode` o `HashNode`, por ejemplo.

La implementación completa puede leerse en `dynalloc.c`, pero la idea general es primero intentar asignar la memoria normalmente con `malloc` y, si esta asignación fallara, entonces utilizar la interfaz de `Cache` para desalojar a los nodos menos recientemente utilizados y volver a intentarlo.

2.1.6. Auxiliares

Dentro de las estructuras auxiliares que implementamos, que se encuentran en el directorio `src/helpers` del proyecto, tenemos a los `LookupResult`, los `AtomCounter`, y la función `quit()`.

Los `LookupResult` permiten expresar el resultado de una operación de búsqueda, como lo es la operación `cache_get`. A través de la interfaz que definen, podemos consultar si el resultado de la búsqueda fue exitoso, si no se encontró el elemento buscado, o si se produjo un error.

Los `AtomCounter` son simplemente una estructura que permite llevar conteos, ofreciendo una interfaz que es segura en entornos concurrentes. La utilizamos para implementar la estructura `CacheStats`, donde almacenamos las estadísticas de uso del servidor.

2.2. Manejo de pedidos con `epoll`.

Para monitorear los *file descriptors* de los sockets y poder gestionar la concurrencia en los pedidos al servidor utilizamos `epoll`. No ahondaremos en los detalles del funcionamiento de `epoll`, pero si consideramos relevante la forma de la estructura que utilizamos para almacenar la información de los clientes.

2.2.1. La estructura `ClientData`.

Al detectarse un intento de operación de IO sobre uno de los *file descriptors* controlados por `epoll`, la función `epoll_wait` nos permite recuperar una estructura de datos que hayamos asociado previamente a ese evento. La estructura que escogimos cargar para cada cliente conectado es la siguiente,

Definición de la estructura ClientData

```
1  typedef struct {
2
3      int socket;
4
5      char command;
6
7      char key_size_buffer[LENGTH_PREFIX_SIZE];
8      int key_size;
9      char* key;
10
11     char value_size_buffer[LENGTH_PREFIX_SIZE];
12     int value_size;
13     char* value;
14
15     int parsing_index;
16     ParsingStage parsing_stage;
17
18     int cleaning;
19
20 } ClientData;
```

donde los campos refieren principalmente al estado en el que se encuentra el pedido del cliente. Teniendo eso en cuenta, y considerando que el envío de bytes de parte del cliente a través del socket puede interrumpirse (es decir, si bien el cliente puede haber enviado todo el mensaje, no necesariamente este llegará por completo en un mismo llamado a `recv`), los campos cumplen las siguientes funciones:

- `socket` es el *file descriptor* del *socket* por el cual nos comunicaremos con el cliente.

- `command` representa la operación que el usuario está intentando realizar (por ejemplo, `PUT` o `GET`).
- `key_size_buffer` es un buffer donde se leerá el tamaño de la clave a recibir.
- `key_size` almacena el valor leído en `key_size_buffer` pero en su forma de int, para evitar convertirlo constantemente.
- `key` almacena un puntero al bloque de la memoria dinámica donde se leerá la clave enviada por el usuario.
- `value_size_buffer` es un buffer donde se leerá el tamaño del valor a recibir
- `value_size` almacena el valor leído en `value_size_buffer` pero en su forma de int, para evitar convertirlo constantemente.
- `value` almacena un puntero al bloque de la memoria dinámica donde se leerá el valor enviado por el usuario.
- `parsing_index` lleva el índice o posición actual del buffer que se está parseando.
- `parsing_stage` es un tipo enumerado que marca en que estado del parseo de su pedido se encuentra el cliente.
- `cleaning` es una bandera que utilizamos en caso de querer vaciar el `socket` del cliente ante un posible error.

La idea detrás de la estructura es que cuando `epoll` despierta a uno de los hilos, este cuenta con toda la información necesaria para poder retomar el parseo del pedido del cliente en el punto donde otro hilo lo dejó (que incluso podría haber sido el mismo hilo en un momento previo). También contiene la información necesaria para ejecutar el pedido en caso de que la etapa de parseo ya haya sido completada.

2.3. Decisiones relevantes.

En esta sección detallamos algunas de las dificultades con las que nos encontramos en el diseño e implementación del programa y como las pudimos resolver.

2.3.1. El problema de definir interfaces.

El primer problema con el que nos encontramos surgió a la hora de plantear la división antes mencionada de la `Cache` como un nexo entre la `LRUQueue` y la tabla hash.

Inicialmente, intentamos implementar por separado a las estructuras de `LRUQueue` y `HashTable`, pero en todas las iteraciones de diseño nos encontramos con problemas para establecer interfaces independientes para cada una de ellas. Es decir, no lograbamos implementar a las estructuras de manera que no se comuniquen entre ellas o tengan referencias de una a la otra.

Esto nos demoró considerablemente, pues apuntabamos a definir interfaces que nos permitieran desacoplar las estructuras. Finalmente, tras varios intentos, concluímos que realmente no es posible definir a las estructuras de manera independiente, pues están muy estrechamente relacionadas (en retrospectiva, no es un resultado tan sorprendente).

Teniendo esto en cuenta, relajamos las restricciones que teníamos en mente para las estructuras y optamos por un diseño en el que cada una de las partes (la `Cache`, la `LRUQueue`, y la "tabla hash") tiene acceso a las demás, aceptando que en las funciones de la interfaz de cada una de ellas se cuente con referencias a las otras. Mantuvimos igualmente la idea de que la `Cache` funcione como un nexo entre ambas partes, pero ya no rechazamos la idea de que una funcion de la `LRUQueue` pueda tomar como argumento un puntero a la `Cache` o a un `HashNode`, por ejemplo.

Este problema con las interfaces se evidenció aún mas cuando definimos a `dynalloc`, pues esta es invocada desde distintas partes de la `Memcached`, como puede ser al crear un `LRUNode` o un `HashNode`, pero su función de desalojo requiere de una referencia a la `Cache` desde donde se invoca a la función (para poder determinar de que `LRUQueue` expulsar a los nodos).

2.3.2. El problema de liberar memoria.

Cuando la `Memcached` se quede sin memoria, `dynalloc` fallará al llamar a `malloc`. Al recibir un puntero `NULL`, procederá a intentar desalojar memoria. Esto se ve reflejado en la implementación de `dynalloc`, que detallamos a continuación:

Implementación de `dynalloc`

```
1  void* dynalloc(size_t sz, Cache cache) {
2
3      if (DYNALLOC_FAIL_RATE > 0 && (rand() % 100) < DYNALLOC_FAIL_RATE)
4          PRINT("Falla de dynalloc simulada.");
5      else {
6          void* ptr = malloc(sz);
7          if (ptr != NULL)
8              return ptr;
9      }
10
11     // Liberaremos el maximo entre el 20% de la memoria ocupada actual↩
           y el size del bloque a asignar por un size_factor
12     size_t memory_goal = max(cache_stats_get_allocated_memory(
13                              cache_get_cstats(cache)) / 5,
14                              sz * SIZE_GOAL_FACTOR);
15     size_t total_freed_memory = 0;
16     ssize_t freed_memory;
17
18     int attempts = 0;
19
```

```
20      while (total_freed_memory < memory_goal && attempts < MAX_ATTEMPTS) {
21
22          freed_memory = cache_free_up_memory(cache, memory_goal - total_freed_memory);
23
24          // Si se produjo algun error, directamente retornamos NULL
25          if (freed_memory < 0)
26              return NULL;
27
28          if (freed_memory == 0) {
29              sched_yield();
30              attempts++;
31          }
32
33          total_freed_memory += freed_memory;
34      }
35
36      // Tanto si se logro el objetivo de memoria como si se agotaron los intentos, devolvemos directamente malloc.
37      // Si no alcanza la memoria, aceptamos que devuelva NULL pues la cache debe estar sobrecargada de pedidos.
38      return malloc(sz);
39  }
```

La idea es que ante una falla de `malloc`, `dynalloc` liberará el máximo entre el 20 % de la memoria que se encuentra asignada en claves y valores y el tamaño del bloque que se quiere asignar multiplicado por un factor (que definimos como 3).

El algoritmo que implementamos consiste en llamar reiteradamente a `cache_free_up_memory`, función de la interfaz de `Cache` que recorre la `LRUQueue` desde el `least_recent` hasta eventualmente el final intentando liberar los nodos en el camino para llegar a la meta de memoria dada por `memory_goal`.

También aprovechamos que si `cache_free_up_memory` retorna 0 como cantidad de memoria liberada entonces sabemos que ha recorrido toda la `LRUQueue` y no logró liberar ningún nodo. Si se da esta situación, en casos de prueba identificamos que puede que se produzca un *livelock*, ya que si los únicos nodos que pueden desalojarse están en zonas cuyos *locks* han sido adquiridos por procesos que están esperando que finalice el desalojo de memoria para continuar, entonces ninguno de ellos podrá avanzar.

Para prevenir esto, introdujimos un límite a la cantidad de pasadas que `dynalloc` puede realizar por la `LRUQueue` sin liberar memoria. Si este límite se supera, entonces estamos en una situación donde no es posible liberar memoria y abandonamos la tarea. Igualmente, `dynalloc` intentará asignar la memoria al retornar un llamado a `malloc`, pero no tendremos ninguna

garantía de que no volvera a fallar. Decidimos que si no logra desalojar memoria entonces la `Memcached` está bajo un gran número de pedidos, y asumimos el costo de posiblemente retornar `NULL` en ese caso.

2.3.3. El problema de los códigos inválidos

En el caso de que recibamos un código que no es válido, decidimos que se cerrara la conexión con el cliente que lo envió. Tomamos esta política debido a que si el código es incorrecto no es posible determinar que es lo que se envío a continuación. Un intento de solución que consideramos fue consumir bytes del socket hasta vaciarlo, pero esto no sería del todo correcto. Al tratarse de un flujo de bytes, no es posible determinar a partir de que punto se volvió a comunicar correctamente, si es que ese punto existe, pues no contamos con los prefijos de longitud que nos delimitan los mensajes. Notemos que si no se logra reconocer el comando, entonces el servidor no puede saber si debe esperar una clave y un valor (como en el caso de un `PUT`), o solo una clave (como en el caso de un `GET`), o nada más (como en el caso de `STATS`).

Para evitar que los mensajes subsecuentes se parseen de manera incorrecta, decidimos que la mejor opción era cerrar la conexión del cliente. Ésto también está respaldado por la idea de que existen los clientes como forma de interfaz entre el usuario y el servidor. Si el cliente respeta el protocolo, no debería darse la situación que describimos.

2.3.4. Como recuperarnos ante un `EBIG`?

Existe un caso extremo en el servidor que sucede cuando este intenta asignar un bloque de memoria y la memoria disponible no es suficiente, pero aún no se ha almacenado nada en la tabla hash o la liberación de memoria resulta imposible. Cuando detectamos que esto sucede, decidimos que el servidor responda con `EBIG`, haciendo alusión a que el mensaje es demasiado grande y no es posible liberar memoria para alojarlo.

Notemos que este error puede darse a mitad del parseo de un pedido. En ese caso, la `Memcached` notifíca al cliente del error, pero puede que queden bytes por leerse en el *socket*. Frente a esta situación, podríamos haber optado por cerrar la conexión con el cliente, pero realmente el cliente respetó el protocolo, solo que la `Memcached` estaba sobrecargada, por lo que no consideramos correcto desconectarlo como si lo haríamos ante un mensaje que no respeta el protocolo establecido.

En el intento de poder continuar recibiendo bytes del *socket* del cliente notamos que, si bien no contamos con memoria suficiente para ejecutar su pedido, si contamos con memoria para leer los prefijos de longitud que nos envió (pues sus *buffers* fueron asignados de estática al crear la estructura `ClientData`). Entonces, a diferencia del caso de un código incorrecto, la `Memcached` si es capaz de determinar cuantos bytes se corresponden con este mensaje dentro del flujo.

Aquí es, entonces, donde utilizamos el campo de `cleaning` de la estructura `ClientData`. Al detectar la situación, marcamos al cliente en modo limpieza. Esto provocará que su mensaje siga siendo parseado, pero que los bytes correspondientes a la clave y valor sean simplemente descartados (son leídos a un *buffer* basura). Cuando se finaliza el parseo, el servidor simplemente

notifíca al usuario y reanuda su funcionamiento habitual, habiendo ignorado el mensaje recibido.

3. Cliente

A continuación, presentaremos el cliente que implementamos en Erlang. Para comenzar, veamos cómo organizamos los archivos:

- `client.erl`: Archivo principal en donde se exponen todas las funciones principales del cliente.
- `utils.erl`: Se compone principalmente de funciones auxiliares que se utilizan en el archivo principal.
- `common.hrl`: Define records, tipos, y algunas constantes que se utilizan en todos los archivos.
- `protocol.hrl`: Declara las constantes asociadas al protocolo.

3.1. Estructuras principales y proceso cliente

Definimos dos *records* para el cliente. El primero, `serverInfo`, lleva información individual de cada uno de los servidores a los cuales nos conectamos. Ésto incluye el `socket` por el cual el cliente se comunica con el servidor, el `id` que usamos para representarlo dentro de nuestra lista de servidores, el `keyCounter` que lleva un aproximado de cuantas claves propias tiene el servidor, y la `address`, definida por la dirección IP y el puerto.

El segundo *record*, `serversTable`, lleva información sobre todos los servidores a los cuales el cliente está conectado, el identificador del cliente, y una lista que se utiliza como una tabla hash. En el se define el `size`, que es el tamaño de la lista de sockets, `servers` que se utiliza como tabla hash, `initial_num_server`, que es el número de servidores iniciales, `identifier` que es el identificador propio del cliente, y `servers_info` que es una lista de la información de los servidores a los cuales el cliente está efectivamente conectado (es decir, no contiene servidores que posiblemente hayan caído).

Para la implementación del cliente, la función `start/1` se encarga de preparar la `serversTable` correspondiente, y luego lanza un proceso que ejecutará en bucle la función cliente. El ID de dicho proceso se registrará y será conocido globalmente. Luego, todas las funciones de la interfaz del cliente, le mandan mensajes a dicho proceso y esperan su respuesta.

3.2. Rebalanceo de cargas

Un problema que nos planteamos al principio del trabajo es como debería responder el cliente ante la potencial caída de un servidor. Está por demás decir que se perderían todas las claves que teníamos almacenadas, pero quedaba por definir que se podría hacer luego de detectar la caída. Nuestra idea fue que, al detectar el apagado o desconexión de un servidor, se aplique un

rebalanceo de cargas que prevenga que el cliente siga enviando pedidos a un servidor que ya no responde. Este rebalanceo lo implementamos reemplazando cada una de las apariciones del *socket* del servidor que queremos eliminar por un *socket* aleatorio de entre el resto de servidores que siguen disponibles (de ésto se encarga la función `rebalance_servers/2`).

Otra detalle que implementamos fue que si el cliente está haciendo un operación de `PUT` y cuando envía el pedido detecta que el servidor ha caído, entonces desencadena un rebalanceo de servidores y el pedido se reenvía al cliente para que éste puede realizar el `PUT` sobre la lista rebalanceada (para las operaciones de `DEL` y `GET` directamente se responde que no se ha encontrado y también se desencadena un rebalanceo). La operación `STATUS` no detecta que un servidor se desconectó hasta que se realice un rebalanceo y `STATS` no genera un rebalanceo de servidores (aunque hubiera detectado que un servidor dejó de funcionar).

3.3. Identificador de cliente

Para dificultar que las claves que fueran guardadas en la cache pudiesen ser obtenidas por otros clientes que se conecten al mismo servidor, implementamos un identificador de cliente. A la hora de crear el cliente, se genera un identificador que se coloca al principio de las claves a guardar. De ésta manera, aunque dos clientes distintos guardasen la misma clave, para el servidor serían diferentes por el prefijo único que poseen. Éste identificador se calcula tomando el timestamp al momento de invocarse a `start/1` para luego aplicarle una función de hash. Aunque ésto no nos asegura unicidad (dos clientes se podrían crear al mismo tiempo, o dos timestamps distintos nos podrían dar el mismo resultado), consideramos que es una buena solución para el alcance del trabajo.

3.3.1. Uso de múltiples clientes en una misma terminal.

Además, nuestra implementación permite que se puedan lanzar múltiples instancias de clientes dentro de una misma terminal de Erlang. Es decir, por cada `start/1` que hacemos, éste nos devuelve un identificador único que se asocia al cliente recién creado. Luego, las funciones definidas para el cliente toman un argumento más (que es el identificador del cliente para el cual queríamos realizar la operación).

De esta manera, si usaramos esta interfaz de cliente como una librería, se nos permitiría tener múltiples cliente corriendo, cada uno posiblemente asociado a un conjunto diferente de servidores y con propósitos totalmente distintos.

4. Forma de trabajo

Para la organización del trabajo comenzamos por plantear cuales serían las principales partes a desarrollar. Claramente existía una división dada por el desarrollo del cliente y el desarrollo del servidor, pero además dentro del servidor identificamos que por un lado habría que desarrollar las estructuras de datos que conformarían a la caché en si, y por otro lado las estructuras y funciones que gestionarían los pedidos con `epoll` y efectivamente se comunicarían con los clientes utilizando los `sockets`.

Con las áreas a desarrollar bien delineadas, comenzamos a plantear ideas para el diseño de cada una. El área de gestión de pedidos y concurrencia con `epoll` ya la habíamos resuelto en gran parte durante el cursado, por lo que quedaba ver como estructurar a la caché y como plantear el cliente. Esto fue lo que mas tiempo nos llevó, pero una vez que nos pusimos de acuerdo en un diseño la implementación fue relativamente rápida.

Lo que no fue tan rápido fue el proceso de arreglar los problemas con el servidor una vez ya implementado, sobre todo en lo relacionado al desalojo de memoria que implicó varias modificaciones sobre el diseño original al darnos cuenta de las limitaciones con las que lidiabamos.

4.1. Uso de IA

Durante el desarrollo del trabajo práctico utilizamos herramientas de inteligencia artificial principalmente para las siguientes tareas:

- Implementación de *getters* y *setters* básicos para las estructuras implementadas.
- Acceso a la documentación de Erlang, ya que el principal recurso que se utilizan son los ejemplos y no la descripción de lo que hace la función.
- Correcciones en scripts de bash que utilizamos para testear los servidores.
- Recomendación de funciones de C para ciertas operaciones, como calcular el número de páginas físicas, calcular el número de hilos del procesador o administrar permisos de los ejecutables.
- Diseño de *templates* para la escritura del informe en LaTeX, como por ejemplo los bloques de código.
- Ayuda para compilar la documentación que escribimos con Doxygen (C) y EDocs (Erlang).