Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencias de la Computación
R-322 — Sistemas Operativos I

Trabajo Práctico

Mini memcached distribuido

§1. Introducción: memcached

`memcached`[2] es un sistema de memoria caché con pares claves-valor para almacenar pequeños "chuncks" de datos de tipo arbitrario $dato(string, object)$ accesible por la red.

Los clientes pueden conectarse a la misma y ejecutar comandos para agregar pares a la caché, buscar el valor asociado a una clave, borrar una entrada, etc. Tiene uso extensivo en sistemas distribuídos de gran escala, por ejemplo para cachear consultas a una base de datos (siempre que sea aceptable tener un valor un poco desactualizado).

El objetivo de este trabajo práctico es implementar un `memcached` propio y distribuido con funcionalidad relativamente completa.

El sistema constará de 2 partes bien definidas:

- Cliente memcached que será quien expondrá las operaciones de toda la caché (get, del, etc.) y manejará los pedidos de los clientes de modo de balancear la carga de trabajo entre los servidores.
- Servidor memcached que será el proceso encargado de almacenar los pares clave-valor. El sistema funcionando contará con al menos 2 servidores corriendo en diferentes máquinas.

Cada servidor memcached debe:

- Soportar el protocolo de comunicación definido en este documento que debe respetarse.
- Proveer estadísticas sobre su uso (detalladas más adelante).
- Tener un límite de memoria configurable que debe respetarse haciendo que el demonio "olvide" valores viejos si no hay suficiente memoria.
- Soportar *multi-threading*: la caché debe correr un hilo por cada hilo de hardware disponible y atender pedidos en simultáneo lo más posible. **No** debe levantar un hilo por conexión. Se sugiere usar `epoll()`.
- Ser *eficiente* y *robusta*.

§2. Protocolo Binario

La caché aceptará conexiones en el puerto 889 [1] TCP. Una vez formada una conexión, el cliente puede enviar pedidos, cada uno de los cuales será respondido.

El protocolo binario tiene los siguientes pedidos y respuestas:

- `PUT` k v: introduce al store el valor v bajo la clave k. Si ya hay un valor asociado a k, entonces el mismo es pisado. El servidor debe responder con `OK`.

[1] El uso del puerto privilegiado es opcional, pueden usar 8889 en su lugar.

- `DEL` k: Borra el valor asociado a la clave k. El servidor debe responder con `OK` si había un valor asociado a k. Si no, contesta con `ENOTFOUND`.
- `GET` k: Busca el valor asociado a la clave k. El servidor debe contestar con `OK` v si el valor es v, o con `ENOTFOUND` si no hay valor asociado a k.
- `STATS`: Devuelve una línea con las estadísticas asociadas a esta ejecución de la caché, en el siguiente formato: `OK PUTS=111 DELS=99 GETS=381323 KEYS=132...` La respuesta debe contener *como mínimo* i) la cantidad de veces que se recibió cada tipo de pedido y ii) la cantidad de pares clave-valor presentes en la caché. Pueden agregarse más estadísticas a discreción. Los contadores internos para cada campo deben ser de al menos 64 bits para evitar overflows.

Ante cualquier otro mensaje el servidor responde con `EINVAL`.

Los comandos y respuestas son representados por un byte, dónde cada uno tiene un identificador único dado por la siguiente tabla:

`PUT`	11
`DEL`	12
`GET`	13
`STATS`	21
`OK`	101
`EINVAL`	111
`ENOTFOUND`	112
`EBINARY`	113
`EBIG`	114
`EUNK`	115

Los argumentos de cada comando se envían de forma consecutiva al código del mismo. Para los datos de longitud variable, prefijamos la longitud del componente como un entero de 32 bits en formato *big-endian*.

Nota: Pueden usar `ntohl` y `htonl` para convertir desde/hacia big-endian.

Por ejemplo, para enviar un `GET` de una clave de longitud 5283612, el mensaje tiene la forma:

13	0	80	159	28	*(bytes de la clave)*

Donde 13 es el código de `GET`; los 4 bytes siguientes representan la longitud de la clave ($0 \times 256^3 + 80 \times 256^2 + 159 \times 256^1 + 28 \times 256^0 = 5283612$); y luego viene la clave en sí.

El servidor contesta de la misma forma con un comando `OK`, usando un campo de longitud variable para la respuesta.

No hay *ninguna* restricción sobre los tamaños de las claves y de los valores (además de la obvia de que entren en un entero de 32 bits, es decir, que sean como máximo $2^{32} - 1 \approx 4\text{GiB}$). Debería ser posible guardar objetos de cualquier tamaño (ej. cientos de megabytes) mientras haya memoria disponible.

§3. Uso de Memoria y Desalojo

La caché debe "desalojar" pares clave-valor si llega a su límite de memoria. Para limitar la memoria, puede usarse la llamada al sistema `setrlimit`. Cuando se llegue al límite de memoria, `malloc()` devolverá `NULL` para nuevos pedidos. En ese momento, el servidor liberar (con `free()`) algunos valores hasta que el `malloc()` tenga éxito. La lógica de cuáles pares olvidar, llamada política de desalojo, *queda a criterio suyo*.

Una opción, que suele ser la mejor en el sentido de que olvida los valores menos útiles, es seguir una política LRU (*least recently used*) y olvidar los pares cuyo último acceso está más en el pasado. Sin embargo, hay otros factores a considerar: la implementación de esa lógica requiere llevar una cola, que requiere una protección, y puede ralentizar al servidor. Es posible que usar políticas de desalojo más laxas, pero que sean más eficientemente implementables, sea más eficiente en general.

No se espera que tomen una decisión completamente óptima, pero sí una que tenga alguna justificación la cúal deberá estar en el informe.

§4. Bajando Privilegios (opcional en 2024 / obligatorio en 2025)

El puerto 889 es un puerto *privilegiado* en Unix, y sólo un proceso corriendo como el usuario `root` puede `bind()`earse al mismo. Sin embargo, no queremos que la caché corra como `root` todo el tiempo, dado que si resulta vulnerable (ej. por un buffer overflow) esto comprometería al sistema entero.

Su implementación debe bajar los privilegios de alguna manera antes de comenzar a recibir conexiones. Está bien si la invocación inicial es ejecutada por `root` (por ejemplo vía `sudo`).

Una solución posible es usando un programa auxiliar que hace el `bind()` siendo `root`, cambia de usuario (por ejemplo con `setuid()`) y luego ejecuta (`exec()`) el programa que realmente implementa la caché. Esta es la solución tomada por el programa `tcpserver`[3] de Daniel J. Bernstein.

Hay otras altenativas (ej. ver `man 7 capabilities`). Elija una e implementelá de manera que su programa no tenga vulnerabilidades asociadas a tener demasiados privilegios.

§5. Cliente Memcached

Vamos a implementar un módulo de Erlang que permita interactuar con los servidores memcached y balancee la carga.

El módulo debe exportar una función `start/1` que se conectará a los servidores (tomando como parámetro la lista de IPs/hostnames de los servidores). Luego de llamar a `start`, puede llamarse a las funciones que implementan los comandos adecuados de la caché, con aridades adecuadas, por ejemplo `put/2`. La llamada `put(K,V)` debería funcionar para `K` y `V` de cualquier tipo Erlang, y comunicarse con la caché vía la red para efectuar el `PUT` (pista: ver `term_to_binary`). Luego, una llamada a `get(K)` debería devolver exactamente `{ok, V}`. En caso de error, puede devolver un átomo `enotfound` o similar[2]. La librería es responsable de enviar el pedido al servidor correcto.

La librería debería abrir un *único* socket por server (al momento de hacer `start`) para comunicarse con los servidores: no es aceptable hacer una conexión nueva por pedido.

Además debería ser posible consultar el estado de la caché a traves de `status/0` que dirá la cantidad de claves almacenadas por el cliente y que porcentaje está en cada servidor. Ej:

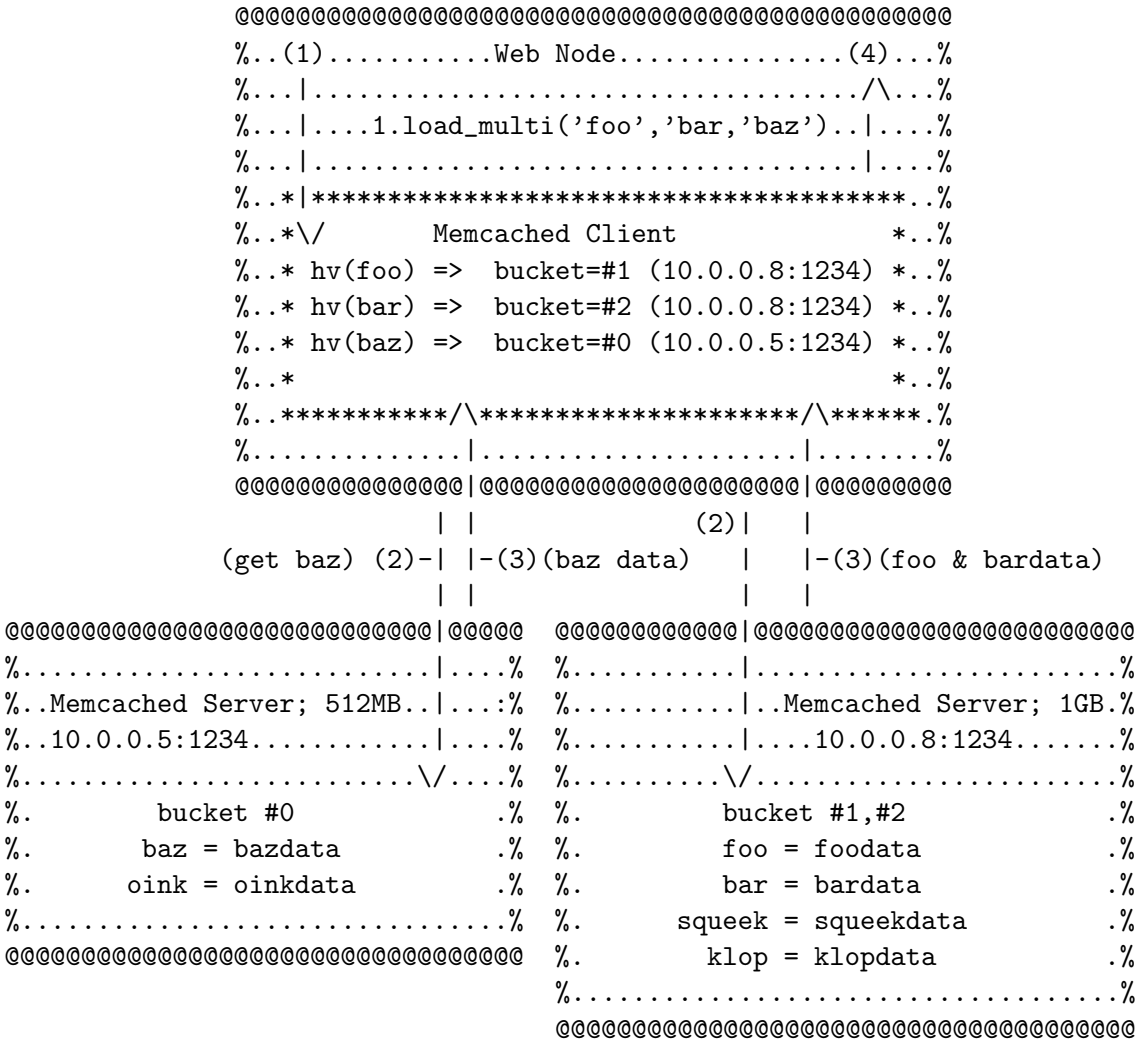
```
> status().
> Total 57 claves: 35 % server1, 43 % server2, 22 % server3
```

§6. Ejemplo de uso de un memcached

Si bien memcached puede usarse en muchas aplicaciones, se usa más comúnmente para acelerar web dinámicas al aliviar la carga de la base de datos Lanzado por Brad Fitzpatrick en 2004[1] y pronto fue

[2] Otra opción es devolver exactamente `V` en el caso exitoso y lanzar una excepción en los errores.

adoptado por sitios populares como Wikipedia, Youtube, Twitter, Facebook. El truco es que, para una clave determinada, necesita elegir el mismo nodo o servidor de Memcached de manera consistente para manejar esa clave, y al mismo tiempo distribuir el almacenamiento (claves) de manera uniforme entre todos los nodos. Simplemente podemos pensar en todos los nodos de Memcached en la red como buckets en una tabla hash. Para una clave determinada, Memcached realizará una función hash para determinar qué depósito es responsable de manejar esa clave en particular, para distribuir las claves de manera uniforme en todos los servidores.



A través de la figura podemos ilustrar un ejemplo en el que se da la siguiente secuencia:

- Paso 1: la aplicación solicita las claves `foo`, `bar` y `baz` utilizando la biblioteca del cliente, que calcula los valores hash de las claves y determina qué servidor Memcached debe recibir las solicitudes.
- Paso 2: el cliente Memcached envía solicitudes paralelas a los servidores Memcached relevantes.
- Paso 3: los servidores de Memcached envían respuestas a la biblioteca del cliente.
- Paso 4: la biblioteca cliente de Memcached agrega respuestas para la aplicación.

Notar que no es necesario que haya una relación uno a uno entre la cantidad de buckets y los servidores

disponibles.

Importante: Es necesario que `start` devuelva un "identificador" de la conexión (para cada una de las conexiones con cada servidor), que luego se pasa a cada función (como `put`), además de sus argumentos originales. Esto permite tener muchas conexiones en simultáneo a distintas instancias de la caché que es lo que necesitamos. El identificador puede ser cualquier cosa: los "usuarios" del cliente lo consideran opaco.

La representación interna en la caché y el modo de comunicación está libre a elección.

Múltiples clientes: Si 2 clientes distintos acceden a un servidor en común y quisieran guardar la misma clave `foo`, podría pasar que un cliente sobreescriba el valor del otro o que obtenga el valor que guardó el otro cliente; esto como mínimo representa un problema de consistencia, para evitarlo cada cliente deberá anteponer a la clave que quiere guardar en la caché un identificador único. Entonces si la clave a guardar es `foo` y el identificador del cliente es `id`, entonces guardará `idfoo`.

§7. Requerimientos

La implementación debe ser robusta: la caché no puede romperse ante entradas mal formadas, ni comportarse de manera incorrecta.

Las estructuras de datos internas deben estar diseñadas para poder responder a los pedidos de manera eficiente, y paralelizar tanto como sea posible.

Debería ser relativamente eficiente y manejar miles de peticiones (simples) por segundo sin problema.

El código debe estar documentado correctamente.

La entrega debe hacerse en un archivo comprimido con un `Makefile` adecuado.

A la vez debe contener un informe detallando las decisiones de diseño tomadas (ejemplo: estructuras de datos internas, manejo de conexiones, política de desalojo, etc), pruebas realizadas, y forma de correr apropiadamente el servidor y el cliente. También dediquen un párrafo mencionando la forma en la que trabajaron. Y si hubo uso de Inteligencia Artificial, aclarar en el informe donde y cómo fue usada (en la generación de código, redacción del informe, etc).

Referencias

[1] Brad Fitzpatrick. «Distributed caching with memcached». En: *Linux J.* 2004.124 (ago. de 2004), pág. 5. ISSN: 1075-3583.
[2] *memcached - a distributed memory object caching system.* `https://memcached.org/`.
[3] *The `tcpserver` program.* `https://cr.yp.to/ucspi-tcp/tcpserver.html`.