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

1. Introducción: memcached

`memcached`[1] es un sistema de memoria caché con pares claves-valor 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 con funcionalidad relativamente completa. Las características principales que debe tener son:

- Soportar un protocolo de texto, que puede usarse desde, por ejemplo, `netcat`.
- Soportar un protocolo binario para datos que no sean texto. Ambos protocolos están totalmente definidos en este documento, y deben respetarse para asegurar la interoperabilidad.
- 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*.
- Además, se debe implementar una librería en Erlang para interactuar con la misma de manera cómoda.

2. Protocolo de Texto

La caché aceptará conexiones en el puerto 888 TCP. Una vez formada una conexión, el cliente puede enviar pedidos, cada uno de los cuales será respondido. Cada pedido y respuesta son exactamente una línea. En el modo texto, ninguna clave y ningún valor pueden contener espacios ni caracteres no imprimibles. Además, ningun pedido ni respuesta puede superar 2048 caracteres. Si un pedido es demasiado largo, el servidor debe contestar `EINVAL`. Si el pedido es correcto pero la respuesta pasa de 2048 caracteres, el servidor debe contestar `EBIG`. Un pedido es siempre una secuencia de palabras separadas por espacios, terminado por un caracter de nueva línea (`'\n'`). La primer palabra es el comando y el resto (alguna cantidad) son los argumentos al comando. Ninguna palabra contiene espacios ni caracteres no alfanuméricos. Los pedidos posibles son:

- `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`.
- `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.
- `TAKE` k: Similar a `GET`, pero remueve *atómicamente* el valor de la caché.
- `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`. Las respuestas del servidor siempre terminan con `'\n'`. (El servidor debe poder probarse fácilmente con `netcat`.) Por supuesto, deben soportarse conexiones simultáneas de varios clientes.

Nota: debido a que existe el modo binario, no todos los valores son representables como texto (pueden incluir espacios, saltos de línea, caracteres nulos, etc). Si se realiza `GET` de una clave que está asociada a un valor binario, el servidor deberá contestar `EBINARY` si (y sólo si) el valor no es representable como texto.

3. Protocolo Binario

El protocolo binario tiene los mismos pedidos y respuestas que el protocolo de texto, pero codificados de otra forma para poder manejar datos arbitrarios tanto en las claves como en los valores. Los clientes deben conectarse al puerto 889 si desean usar el modo binario.

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
`TAKE`	14
`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*.

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.

Para el comando `STATS`, que no tiene una forma definida, el modo binario contesta `OK` con los mismos datos exactos que el modo texto, en el mismo formato (una sucesión de `V=n`). También, por supuesto, indicando la longitud del campo. Así un cliente puede pedir `STATS` sin conocer exactamente las estadísticas presentes en el servidor.

En modo binario 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.

4. 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.

5. Bajando Privilegios

El puerto 888 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`[2] de Daniel J. Bernstein.

Hay otras altenativas (ej. ver `man 7 capabilities`). Elija una e implementelá de manera que su programa sea lo más seguro posible.

6. Bindings para Erlang

También vamos a implementar un módulo de Erlang que permita interactuar con la caché sin requerir que cada cliente implemente el protocolo.

El módulo debe exportar una función `start/0` que se conecte al servidor (también puede tener aridad 1, tomando como parámetro la IP/hostname del servidor). 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[1]. La librería debería abrir un *único* socket (al momento de hacer `start`) para comunicarse con el servidor: no es aceptable hacer una conexión nueva por pedido.

Opcional: Es aceptable (y de hecho preferible) que `start` devuelva un "identificador" de la conexión, 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, mientras que con la versión anterior no es posible operar con más de una instancia a la vez. El identificador puede ser cualquier cosa: los clientes de la librería lo consideran opaco.

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

7. Requerimientos

La implementación debe ser lo más cercana posible a calidad industrial. 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).

Referencias

[1] *memcached - a distributed memory object caching system.* `https://memcached.org/`.
[2] *The `tcpserver` program.* `https://cr.yp.to/ucspi-tcp/tcpserver.html`.

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