mcache: An In-memory Caching System

Benjamin W. Gafford gaffordb@grinnell.edu
Samuel B. Stickels stickels@grinnell.edu
Maison M. Dodge dodgemai@grinnell.edu

OVERVIEW

When clients across a network are frequently computing the same data, some sort of caching system is oftentimes necessary. A caching system would act as an intermediary between more persistent storage solutions, such as disk storage (which is also more inefficient), and a full-fledged database. In order to approach this issue and alleviate database load, we sought to implement an in-memory caching system, similar to that of "Memcached." Memcached is an open source, distributed memory caching system used to speed up dynamic web applications [4]. Our system aims to achieve a similar goal, and so we utilized a similar interface. The purpose of our program is to increase program efficiency and performance, specifically regarding accesses to items in memory, or act as an intermediary between a more persistent storage option and having to manually manage the memory. We accomplish this by temporarily storing frequently-requested data in memory as opposed to holding it on the disk. Since data is more accessible from memory, it speeds up common datastore queries. For example, if a network is attempting to compute certain values, it is inefficient for each client to recompute the value for each iteration. Instead, they can simply compute it once, store it on the cache, and access it from the cache as necessary. We implemented this system in two ways, one which acts within the client and another which runs independently and acts as a server which can be queried using the mcache interface provided to the client. Our implementation was tested using a variety of testing suites in order to verify program correctness. These tests include ensuring that the system is effectively able to store and retrieve data of different types (structs, arrays, and basic types), and that the number of objects that were stored is consistent with the number of objects that should be stored, given an artificial cache size constraint. In order to determine the necessity of the server implementation, we compared the efficiency of the different implementations with several constraints. In the end, we observed that the in-memory caching method is more successful in an environment where a large amount of queries for distinct data were necessary. With fewer queries, the network method was superior. We would say, however, that if you have a single client it would likely be a better idea to adopt the in-process system rather than the server implementation.

DESIGN AND IMPLEMENTATION

Cache To implement the storage for the in-memory cache, we utilized a fixed-size hashmap with string values for keys and values consisting of a pointer to the byte sequence received from the user, and the corresponding object size. A hashmap seemed like the logical choice, as it allows for efficient and effective sets and gets for a map/association list data structure. The original Memcached uses a hashmap as well, however they include various

optimizations and an alternate method of allocation than the standard malloc calls that we use [5]. The hashmap has the standard API, available, and is implemented using chaining. In order to ensure thread-safety, each bucket has a lock associated with it, and must be obtained before accessing the ordered linked list that is associated with the bucket. Object size for our cache currently has an upper limit of 2^{15}, although this can be changed with relative ease. This limitation is simply due to the fact that in order to send objects over the network, we were first sending/receiving the size of the message, and this is done using a 2-byte integer value. We figured that 2 bytes would be sufficient for almost all objects, and users can always store larger objects (ie very large arrays) in chunks rather than in a single key. Our API largely mirrors that of Memcached, but with some simplifications [1, 5]. The following lists the available commands and the purposes of the commands:

- `mcache_init(server address)`: Must be called to establish connection with the server (or set up all of the appropriate structures, in the case of the in-client implementation)
- `mcache_set(key, value, obj_size)`: Stores the specified value using the identifier key in the cache. If the key already exists, it will be updated.
- `mcache_add(key, value, obj_size)`: Stores the specified value using the identifier key in the cache. If the key already exists, the value is left alone.
- `mcache_get(key)`: Returns the information stored at the specified key.
- `mcache_delete(key)`: Deletes the key/value pair from the cache.
- `mcache_exit()`: Cleans up any connections, in the case of the server implementation, or data structures in the case of the in-process implementation

Multiple clients can store frequently computed values in memory rather than on disk. To achieve this, the server devotes a specific amount of its memory to caching data for clients in the hash table.

A primary feature of the cache that distinguishes it from a database is that the cache temporarily holds the data. This provides a benefit to the user as data can be stored and cleaned up behind the scenes, and the user can indiscriminately store and get data. The cache size is currently 300 bytes but can be changed using macro in the file mcache_types.h. Since the cache has a fixed size, eviction is necessary. Other than being deleted from the cache via client request, items are also evicted passively using an implementation of the least recently used method (LRU). In other words, the data at the end of an ‘accessed’ queue that we manage internally is removed first. When trying to place an item into the cache that exceeds capacity, least recently used items are evicted until there is enough space available. If an object is greater than the cache size, the object is not stored and no items are evicted. The keys of the hashmap are stored in a thread-safe doubly-linked queue. Whenever data is accessed, the key moves from its current position to the front of the queue, making it so the least recently used keys are at the end and are therefore removed eventually. Our eviction implementation could definitely be improved in a number of ways, as every time a key is used, the key list needs to be traversed to remove the key if it exists, and then needs to be added to the front of the list. This could be improved by adding in the actual node to the hashmap, and then saving yourself a traversal, however we did not have time to implement this, as we figured

ensuring thread-safety with this implementation would be much more difficult, and it would require some semi-major changes to our hashmap. Another method of improving this would be to have a separate thread dedicated to dealing with updating the eviction queue, however this makes the fixed-size cache less precise, and would introduce more complexity into our code that we simply didn't have enough time to deal with.

Network The interface of our caching system is provided with a client-server network connection. This was an obvious design decision, as one of the key aspects of our caching system is the ability of multiple clients to utilize the same cache, so the server only needs to access this memory from one location. Additionally, implementation as a separate process allows for a larger cache size, because the process can then dedicate all of its memory to caching. The server listens for connections, and establishes TCP connections with the users, and reads for input in a child thread. Requests are sent from the clients to the server in plaintext, and if accompanying information is provided with the requests, such as in the case of a set, the data is serialized and sent over the network as a sequence of bytes [2, 3]. This sequence of bytes is stored directly in the hash table of our caching system, as well as the size of the stored data in bytes. Since we are directly storing the byte sequence, we therefore do not need to take into account the endianness of each respective machine, nor the actual structure of the data. One thing to keep in mind when transferring this data is that you cannot rely on duplicating the socket into a file stream, and then reading input using `getline`. As the unstructured data is being sent, it is possible (and likely) that there will be individual bytes that are equivalent to a newline character, and then are interpreted as such and will falsely terminate the message. Since data is stored in this unstructured format, it is left to the user to maintain knowledge of what data type corresponds to their keys. Maintaining an internal database of what key corresponds to what type is left to the client for the sake of simplicity, portability, and versatility of our system.

After the server receives a request from the client, the server parses through the command and, depending on the request, will return an appropriate response. For `set`, `add`, and `delete`, no response is sent and if anything internal fails, this is hidden from the client. In the case that the user issued a `get` request, the client is sent the object size and the unstructured data. If a request is made that is not included in the aforementioned list, the query is dropped and the server continues silently. The following describes an overview of how the server carries out each function:

- `set`: The message is received by reading from the socket, first reading the message size, then reading the actual message, then reading the object size. The received message is parsed into tokens. The key is then stored into the hashmap with a pointer corresponding to the data (which has been copied). If the object size is less than the maximum cache allocation, stored objects are evicted until there is enough room for the new object. Update the global memory allocation by putting the key and data in the hashmap and update the keys queue.
- `add`: Add is implemented the same as set, but checks to see if the key exists in the hashmap, and if so, simply touches the key and exits.
- `get`: The data is written to the socket (returned to the client) by getting the data from the hashmap

The effect of cache size, number of values, and implementation type on time

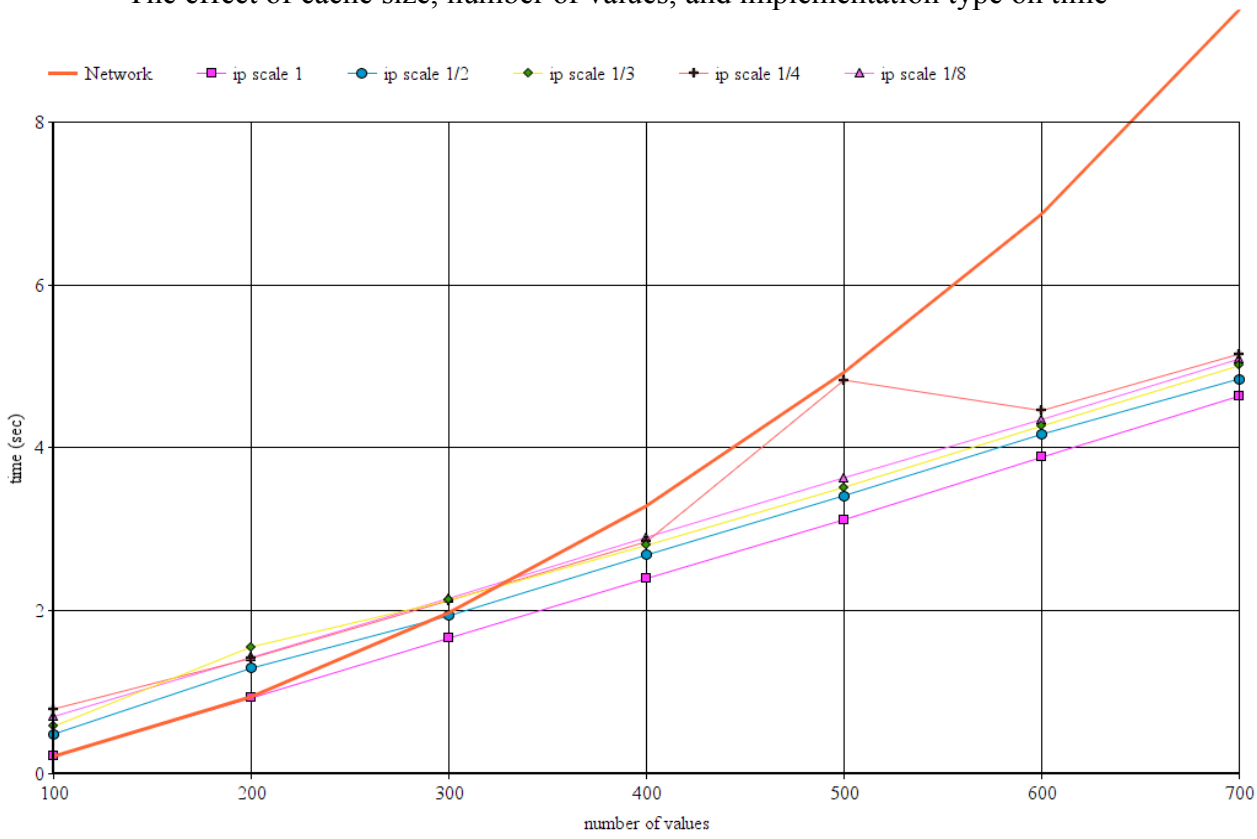

Figure 1

using the provided key. Update key queue.

- `delete`: The hashmap's remove function is called, and the eviction list is updated to no longer include the key.

Our initial implementation only served one client at a time, and then once we had this running successfully, we implemented concurrency. To accomplish concurrency, we made our internal hash table thread-safe by utilizing locks on each bucket. With a thread-safe cache available, the server maintains an internal list of client sockets and has a thread for each socket dedicated to waiting for the client to issue a command. The global memory allocated counter also needed a dedicated mutex lock, and the eviction key list needed to be thread-safe as well. Once provided with client input, each thread handles the client's query appropriately, and, if appropriate, writes back to the client with relevant data. When a client wishes to disconnect, they do so silently, and the failed socket is dropped by our server.

EVALUATION

Our evaluation strategy was to simulate a use-case for this system and comparing the within-process implementation to the server implementation to determine efficacy in different environments. If the server implementation were useful for a single client, then we would see that the larger cache size sufficiently made up for the latency associated with querying over the

network. Since the primary purpose of having a server implementation is to allow for a larger cache size, we limited the size of the in-client implementation to varying sizes and tested the performance in each case. Additionally, we induced an artificial penalty to a cache miss, which would be realistic to a real-world scenario as the purpose of the cache is to avoid these expensive computations when possible. This penalty is simply computing the 30^{th} number in the Fibonacci sequence in the recursive non-dynamic programming way, which results in about a .09 second delay for the user every cache miss. We collected this data in the following environment: Macbook Pro (2015) using macOS Sierra 10.12.6 with 2.7 GHz Intel Core i5 and 8 GB RAM. We collected the time data points using the `time` UNIX function. As illustrated by Figure 1, the in-client implementation with a full cache size (in this case 300 bytes) is optimal and reducing the cache size simply shifts the y-intercept up. Therefore, when dealing with a fewer number of distinct values, the network implementation will be superior due to the larger cache size, but is also non-linear, and is far worse at dealing with a larger number of values. For that reason, we would conclude that with a single client, it would almost always be more favourable to use the within-process implementation. However, the network implementation still has a use-case if you have multiple clients that are working over the same (or similar) dataset.

REFERENCES

[1] Brown, M. (2010, August 03). Applying memcached to increase site performance. Retrieved from https://www.ibm.com/developerworks/library/os-memcached/

[2] C – serialization techniques. (2011, May 14). Retrieved from https://stackoverflow.com/questions/6002528/c-serialization-techniques

[3] Converting struct to byte and back to struct. (2012, December 08). Retrieved from https://stackoverflow.com/questions/13775893/converting-struct-to-byte-and-back-to-struct

[4] Memcached. (2009). Retrieved from https://memcached.org

[5] Memcached community. Memcached/memcached. *Github*. Retrieved from https://github.com/memcached/memcached/wiki/Overview#how-does-it-work