Memcached vs Redis: Benchmarking In-memory Object Caches

Milan Pavlik

2016-01-18

1 Abstract

Abstract Goes here

Contents

2 Motivation

As the world's demand and reliance on the Internet and near instantaneous communication increases, so do the requirements of computer systems. Clock speed improvements in CPU architectures and shift to multiprocessing architectures are by themselves not sufficient to provide sufficient required computing power. With the improvement in commodity hardware and a shift to commodity computing, it has become increasingly important to design applications capable of utilizing both multiprocessing on a single machine as well as capable of exploiting distributed computing.

Parallelization of work has also introduced an increased complexity system architectures as well as application architectures. The increased complexity is derived primarily from the effort to better utilize multiprocessing. As a result, coherence, scalability and resiliency becomes of great concern to system architects.

The general approach to improving performance is to *"(a) Work harder, (b) Work smarter, and (c) Get help."* [4] Utilizing distributed computing aims to achieve *c).* An approach to work harder can be utilize parallel architectures better. Additionally, a system cannot always be fully parallelized due to access to shared resources. Due to Amdahl's Law, such systems will not be able to fully utilize the potential speedup provided by advances in architectures alone. Conversely, thinking in the opposite direction in terms of *b)* we can ask if a simpler and less complex architecture can perform better? And if so, what are the aspects of it's design that do make it more performant?

As complexity increases, system architectures are dependent on the ability to perform effectively. Therefore, it is important to understand how a single server scalability is influenced by applications with single and multi threaded architectures.

The motivation behind this study is to understand and evaluate two state of the art object caches with varying architectural decisions in terms of performance scalability on a commodity server. Firstly, we focus is on a well studied object cache called *Memcached*, a high performance application designed with multi-threading as a core feature. Secondly, we focus on a younger cache - *Redis* - a single threaded high performance cache. Finally, having analyzed the performance of two architecturally different object caches, we can evaluate their performance and gain a better insight into the effectiveness of each of their respective design.

3 Memory Object Caches

Traditionally, a cache is a data structure in either hardware or software capable of storage and retrieval of data. Generally, a *value* of a computation is stored in the data structure with a given *key*. A cache is generally used to speed up data retrieval. Often, a pattern of execution is to first attempt to retrieve a *value* from the cache by it's *key*. If the *key* is present in the cache, *value* is returned which is called a *hit*. Otherwise, a failed retrieval is indicated and the attempt to access the cache is called a *miss*. If a miss occurs, data is frequently computed or retrieved elsewhere and stored in the cache to speed up the next execution cycle.

Caches are a heavily used across hardware and software systems. For example, the CPU uses multiple levels of caches in order to speed up memory access. Another example of a cache is in database servers to cache queries and reduce computation time. Efficient use of caching can drastically improve time required to retrieve data.

3.1 Purpose

Firstly, the purpose of a memory object cache is to use the machine's available RAM for key-value storage. The implication of a *memory object cache* is that data is only stored in memory and should not be offloaded on the hardware in order to not incur hard drive retrieval delay. As a result, memory caches are often explicitly configured with the maximum amount of memory available.

Secondly, an *object* cache implies that the cache itself is not concerned with the type of data (binary, text) stored within. As a result, memory object caches are multi-purpose caches capable of storage of any data type within size restrictions imposed by the cache.

Finally, memory object caches can be deployed as single purpose servers or also co-located with another deployment. Consequently, general purpose object caches often provide multiple protocols for accessing the cache - socket communication or TCP over the network. Both caches in question - Memcached and Redis - support both deployment strategies. Our primary focus will be on networked protocols used to access the cache.

3.2 Example

TODO: Example usage of a cache for a web server?.

3.3 Desired qualities

Firstly, an object cache should support a simple interface providing the following operations - *get*, *set* and *delete* to retrieve, store and invalidate an entry respectively.

Secondly, a general purpose object cache should have the capability to store items of arbitrary format and size provided the size satisfies the upper bound size constraints imposed by the cache. Making no distinction between the type of data is a fundamental generalization of an object cache and allows a greater degree of interoperability.

Thirdly, a cache should support operation atomicity in order to prevent data corruption resulting from multiple simultaneous writes.

Furthermore, cache operations should be performed efficiently, ideally in constant time and the cache should be capable of enforcing a consistent eviction policy in the case of memory bounds are exceeded.

Finally, a general purpose object cache should be capable of handling a large number of requests per second while maintaining a fair and as low as possible quality of service for all connected clients.

3.4 Design and Implementations

The design and implementation of a general purpose cache system is heavily influenced by the desired qualities of a cache.

Firstly, high performance requirement and the need for storage of entries of varying size generally requires the cache system to implement custom memory management models. As a result, a mapping data structure with key hashing is used to efficiently locate entries in the cache.

Secondly, due to memory restrictions, the cache is responsible for enforcing an eviction policy. Most state of the art caches utilize least recently used (LRU) cache eviction policy, however, other policies such as first-in-first-out can also be used.

In the case of *Memcached*, multi-threaded approach is utilized in order to improve performance. Conversely to Memcached, *Redis* is implemented as a single threaded application and focuses primarily on a fast execution loop rather than parallel computation.

3.5 Performance metrics

Firstly, the primary metrics reflecting performance of an in memory object cache are *mean latency*, *99th percentile latency* and *throughput*. Both latency statistics are reflective of the quality of service the cache is delivering to it's clients. Throughput is indicative of the overall load the cache is capable of supporting, however, throughput is tightly related to latency and on it's own is not indicative of the real cache performance under quality constraints.

Secondly, being a high performance application with potentially network, understanding the proportion of CPU time spent inside the cache application compared to time spent processing network requests and handling operating system calls becomes important. Having an insight into the CPU time breakdown allows us to better understand bottlenecks of the application.

Finally, the *hit* and *miss* rate of the cache can be used as a metric, particularly when evaluating a cache eviction policy, however, the hit and miss rate is tightly correlated with the type of application and the application context and therefore it is not a suitable metric for evaluating performance alone.

3.6 Current State of caches and Usage in industry

TODO

3.7 Memcached

Memcached is a "high-performance, distributed memory object caching system, generic in nature, but intended for use in speeding up dynamic web applications by alleviating database load." [3] Despite the official description aimed at dynamic web applications, memcached is also used as a generic key value store to locate servers and services [1].

3.7.1 Memcached API

Memcached provides a simple communication protocol. It implements the following core operations:

- `get key1 [key2..N]` - Retrieve one or more values for given keys,
- `set key value [flag] [expiration] [size]` - Insert *key* into the cache with a *value*. Overwrites current item.
- `delete key` - Delete a given key.

Memcached further implements additional useful operations such as `incr/decr` which increments or decrements a value and `append/prepend` which append or prepend a given key.

3.7.2 Implementation

Firstly, Memcached is implemented as a multi-threaded application. "Memcache instance started with n threads will spawn n + 1 threads of which the first n are worker threads and the last is a maintenance thread used for hash table expansion under high load factor." [2]

Secondly, in order to provide performance as well as portability, memcached is implemented on top of *libevent* [5]. "The libevent API provides a mechanism to execute a callback function when a specific event occurs on a file descriptor or after a timeout has been reached. Furthermore, libevent also support callbacks due to signals or regular timeouts." [5]

Thirdly, Memcached provides guarantees on the order of actions performed. Therefore, consecutive writes of the same key will result in the last incoming request being the retained by memcached. Consequently, all actions performed are internally atomic.

As a result, memcached employs a locking mechanism in order to be able to guarantee order of writes as well as execute concurrently. Internally, the process of handling a request is as follows:

1. Requests are received by the Network Interface Controller (NIC) and queued
2. *Libevent* receives the request and delivers it to the memcached application
3. A worker thread receives a request, parses it and determines the command required
4. The *key* in the request is used to calculate a hash value to access the memory location in *O(1)*
5. Cache lock is acquired *(entering critical section)*
6. Command is processed and LRU policy is enforced
7. Cache lock is released *(leaving critical section)*
8. Response is constructed and transmitted [6]

We can observe that steps *1-4* and *8* can be parallelized without the need for resource locking. However, the critical section in steps *5-7* is executed with the acquisition of a global lock. Therefore, at this stage execute is not being performed in parallel.

References

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[2] Solarflare Communications Inc. Filling the pipe: A guide to optimising memcache performance on solarflare hardware. (2), 2013.

[3] Danga Interactive. Memcached, 2006.

[4] Gregory F Pfister. *In search of clusters*. Prentice-Hall, Inc., 1998.

[5] Niels Provos and Nick Mathewson. libeventan event notification library.

[6] Alex Wiggins and Jimmy Langston. Enhancing the scalability of memcached. *Intel document, unpublished*, 2012.