Memcached vs Redis: Benchmarking In-memory Object Caches

Milan Pavlik

2016-01-18

1 Abstract

Abstract Goes here

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."* [1] 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

3.1 Purpose

In the traditional sense of way, a cache is a data structure which stores the results of a computation for later retrieval. Generally, a result of a computation is stored against a *key* with a *value* representing the computed result. When retrieving data from the cache, we say there is a *hit* if the key can be found in the cache and a *miss* otherwise. Caching is widely used across the hardware and software stack. For example, modern processors utilize a CPU cache to increase the speed of memory lookup. Similarly, a cache can also be found inside a web server to speed up retrieval of results from the server.

An object cache is a general purpose cache, generally designed as a standalone application, which provides an interface to cache any type of object - for example, an object cache can store large text documents, small tweets or binary data such as images.

A memory object cache is an object with additioanal restriction where data is only stored in the available (or assigned) RAM of the computer. This restriction is important as storing data on hardware slower than RAM would incure too high of a latency and would degrade quality of service.

Both Memcached and Redis are open-source implementations of memory object caches.

3.2 Desired qualities

TODO

3.3 Design and Implementations

TODO

3.4 Performance measurements and methodology

TODO

3.5 Current State of caches and Usage in industry

TODO

3.6 Memcached

From the early development stages, memcached has been designed in a client-server architecture. Therefore, a memcached applications receives a command based on its API, executes the command and returns a reply to the client. Memcached is deliberetly designed as a standalone application rather than being integrated into a particular system/framework in order to be able to act as a general purpose cache and allow decoupling of responsibilities in an system architecture.

Memcached implements its distributed protocol through consistent hashing on the client side. Therefore, keeping logic on the server side minimal and allowing the clients to figure out which instance to talk to. In order to further improve horizontal scaling properties of memcached, solutions such as Twemproxy [9] exist to support scalability of an individual shard of a distributed memcached deployment.

The memory requirements of memcached are specified as a configration option before the memcached application is started. Knowing an upper bound on the amount of memory memcached can use allows memcached to claim the required memory and handle memory management itself rather than using *malloc, free or realloc*. *Slabs* are structured to be blocks of memory with 1MB allocated to them. Each slab belogns to a *slab group* which determines the size of each chunk inside the slab. By default, the minimum chunk size is 80 bytes with a hard maximum of 1MB. The growth factor between different slab groups is 1.25. Within each slab group, a Least Recently Used (LRU) eviction policy is employed effectively evicting entries least recently used within a similar memory requirement first.

Private networks are the intended target of memcached where applications designed to be publicly exposed access memcached on behalf of the requestor rather than exposing the cache directly. By default, memcached provides access to all entries of the cache for all clients but there is also an option to be used with a Simple Authentication and Security Layer (SASL) option.

Memcached is a multi-threaded application which introduces the requirement to lock resources during critical sections in order to prevent race conditions. The main reason for designing multi-threaded applications is improve performance. Parsing a request and understanding the nature of a request can be in parallel with data retrieval from memory while a response is being constructed, all in their own respective threads. However, in order to achieve the apprearance of operation atomicity, a mutual exclusion lock is required. A request lifecycle is as follows [10]:

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

Given the outline above, we can see that steps 5. to 7. transform the parallel nature of processing a request into a serial process. Optimizations to the critical section have been well studied but it should be noted that memcahed suffers from overheads related to global lock acquizition and release.

3.7 Outline

Memcached is a simple distributed memory object cache [1]. It provides a simple interface to allow systems to store, retrieve and update the contents of the cache. It is developed as an open source project and has been extensively studied in the literature. Often, it is used as an application of choice to benchmark system configurations in terms of network throughput, memory allocation policies and more generally to understand how a system performs under stress. Furthermore, it is often used as an application for experiementation and implementation of next generation technology such as the use of a Field Programmable Gate Array (FPGA) [2].

The API supported by memcached is straightforward. Memcached philosophy is to execute commands against an item of the cache rather than manipulate the cache as a whole. Out of the box, memcached supports the following operations for retrieval: *set, add, replace, append, prepend and cas* [3]. Similarly, memcached supports the following storage commands: *get, gets, delete and incr/decr* [3]. The API is deliberetly designed to be intuitive making the effect of an action predictable. The action labelled *cas* perhaps requires further clarification, however. The full action name is *check and set*, data is stored only if the comparison with current value fails.

Memcached has grown to be a very popular general purpose cache in the industry. Currently, Facebook is considered to have the largest deployment of memcached in production [4] while there are many

other companies utilizing large deployments of memcached as building blocks of their infrastructure, these include Twitter[5], Amazon [6] and many others.

In the simplest memcached deployment, an instance of memcached can be run alongside another application, for example a web server. In such a setup, no network communication is required and the web server can talk to memcached over a local unix socket. This configuration has disadvantages, for example, horizontally scaling the web server would require another instance of the cache to be deployed as well potentially leading decreased cache hit rate.

More complicated deployments generally utilize a memcached instance running on a seperate host with all instances of, for example, web servers communicating with a single memcached host. The advantage of such a setup is decreased coupling and increased potential for scalability by adding more instances of both web server and memcached.

In the largest scenarios, such as Facebook, a large number of client applications are talking to a number of memcached clusters responsible for a given type of information. Effectively creating a data layer where any client application can request information from any pool increasing modularity and interoperability of the infrastructure. [Ref?]

(Diagram to illustrate deployments here?)

To illustrate the importance and also the size of a memcached deployment, a workload charasterization from Facebook will be used [7]. Figure below illustrates the throughput observed in a Facebook pools deployed over the course of 7 days. We can observe that the total number of requests is close to 1.26 trillion requests over 7 days, this is on average 2.08 million requests a second. The volume itself is large, however, considering Facebook has 1.44 billion active monthly users [8], however, it does demonstrate the scale at which Facebook utilizes memcached and the impact memcached has on ability to scale and handle traffic at Facebook.

3.8 Configuration options

Memcached provides a convenient command line configration options to tweak the performance of memcached through various parameters, the most important ones are:

- *-d* runs application in daemon mode
- *-p port* binds application to a port (18080 by default)
- *-m memory* defines how much memory to allocate to memcached.

Given the host hardware has 8GB memory, 6GB will be allocated to memcached to leave some memory for the underlying operating system. Throughout this paper, mostly options outlined above will be utilized. Where applicable, further settings will be explained.

4 Methodology

* Quality of Service (99th ¡ 1ms?) * Testing setup * Comparison to testbeds in other papers * Memtier and comparison to others * Justify Memtier * Open loop vs closed loop * What are we interested in * Discuss error conditions and repeatability

4.1 Quality of Service

Firstly, it is important to define an acceptale quality of service (QoS) for an object cache in question. Distributed systems are increasingly more popular with responses to requests being a composition of smaller responses from respective sub-systems. Given all sub-systems must return a response before the complete response is serviced to the requestor, the slowest of all smaller responses will determine the overall response time. Frequently, the QoS aimed for is sub-1ms latency. Similar target is used by Leverich and Kozyrakis [1]. Therefore, in this study the aim will be to achieve tail latency under 1ms, that is in 99

4.2 Testing setup

The performance benchmarks are run on 8 machines with the following configuration: 6 core Intel(R) Xeon(R) CPU E5-2603 v3 @ 1.60GHz [2], 8 GB RAM and 1Gb/s Network Interface Controller (NIC).

All the hosts are connected to a Pica8 P-3297 [3] switch with 48 1Gbps ports with a star as the network topology. A single host is used to run an object cache system while the remaining seven are used to generate workloads against the server.

4.3 Workload generation

Workload for the cache server is generated using Memtier Benchmark [4]. The Memtier Benchmark provides a configurable parallel workload generation for both Memcached and Redis. Additionally, it allows for a high level of configrability.

4.3.1 Memtier Benchmark Behavior

Memtier Benchmark provides various paramters allowing for a variable configuration. As part of the configration, the user is allowed to specify the number of threads and the number of connections per each thread memtier should make. The standard lifecycle of each thread is as follows:

1. Set up n connection configurations 2. For each connection configuration, initiate the connection over the desired protocol (default: TCP) 3. Make a request 4. Tear down the connection 5. Repeat iterations

4.3.2 Open loop vs Closed loop

Memtier is closed loop

References

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