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."* [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

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

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.

3.6 Current State of caches and Usage in industry

TODO

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.