Evaluation of Memcache

Evaluate the design of our highly scaled Memcache.

We'll cover the following

In this lesson, we will recap how well our system addresses the requirements we set in the first lesson.

Accessing and updating highly viewed content

Highly viewed content means that the load and, thus, the latency can peak randomly. To make the service more elastic, we allowed more front-end clients to scale the two layers independently. This meant that multiple clients could communicate with a few Memcached servers due to a non-uniform distribution key access. Having these many clients caused items in Memcached to face two problems: stale sets and thundering herds. We used leases to fix this issue. According to this study, query rates dropped from 17000 queries per second to 1300 queries per second when using the database with leases. This resulted in reduced load on the storage layer. Moreover, not all items have the same characteristics. To deal with the interference caused by these different requirements, we segmented our key-value items into different pools. The breakdown is given below:

Individual Pools

Pool	Benefits
Wildcard pool	This pool is the default pool. It observes the highest packet rates when compared to the other pools.
App pool	This pool has a high churn rate since content is accessed for a while before fading away. It also has the highest miss rate, since most of the items fade away in just a few hours.
Replicated pool	This pool has the highest `get` rate to manage the ever-increasing workload of highly popular items.
Regional pool	This pool has relatively large items, which are accessed less frequently.

We also used replication in the replicated pool to scale to the user demand. Replication means that more Memcached servers serve the same key-value items. The replicated pool has the least miss rate of 0.053%, while the second best for reference was the wildcard pool, with 1.76%.

While replication was one of our requirements, we also wanted Memcache to be memory efficient. To do this, we used regional pools that stored less frequently accessed keys with relatively larger items. This allowed our system to become more memory efficient while having the same throughput.

Reducing network load

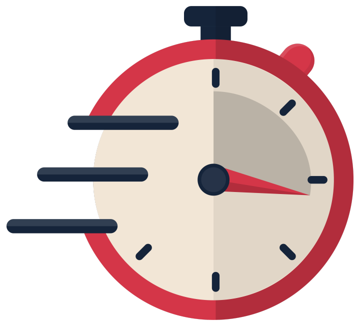
Due to the all-to-all communication pattern that occurs when many of the web servers are communicating with all the Memcached servers, we faced a significant load on the network. We took several steps to efficiently send requests to Memcache. We constructed a directed acyclic graph (DAG), and then the web server used the DAG to maximize the number of `get` requests that can be made concurrently. We then used a sliding window mechanism to further control the network flow, which allowed us to balance between unnecessary latency and incast congestion.

Insulating the storage layer

To protect our storage clusters from cascading failures, we used a few methods to deal with that. First, we designated a few percent of the Memcached servers to serve as the gutter pool, which ensured that the storage layer would still be protected if the primary Memcached servers failed. When we brought new clusters online, we warmed those cold clusters by using other warm clusters, thereby adding further insulation to the storage layer.

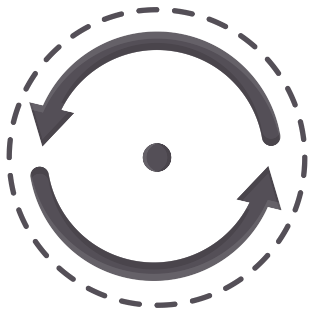
Latency

To improve the latency of the system, we used UDP instead of TCP, implemented a sliding window mechanism, and introduced regional pools. The median 7-day request latency is 333 microseconds. By latency, we mean the round-trip time taken for a response to reach the user from the Memcached server. There is a significant difference in latency between the 75th percentile (475 microseconds) and the 95th percentile (1.135 milliseconds). This is because the 95th percentile represents items with large response sizes. Any future work can be done to reduce the latency for such large items.



Consistency

We divided Memcached into multiple levels, where the regional and cross-regional levels have vastly different requirements for consistency because of different latencies. For the regional level, we expect to maintain strong read-after-write consistency, which is achieved using invalidations. For the cross-regional level, we wanted



eventual consistency, which we achieved by using remote markers on the cross-regional level.

Fault-tolerance

The failure of a single cache node has limited impact (assuming at any given instance in time the failing caching nodes are few). Popular keys are replicated multiple times in a data center.

The failure of some nodes will increase the load on the remaining replicas (until the system is doing further replication). In case of a full data center going offline (a relatively rare event), the DNS will re-route customers to a different data center. While the overall load will be higher initially, a new steady state is expected to reach after some time.

Conclusion

In this chapter, we examined the methods used to address problems when operating a key-value store on a global scale. We divided the problem into different levels and then optimized them individually. We balanced between consistency, availability, and latency by making different trade-offs from the base and the single server level, to the top of our system, and the cross-regional level.

System design wisdom in scaling Memcache

- This chapter showed us how systems evolve organically over time. Facebook started with an initial caching system and incrementally evolved it to meet new needs or to overcome bottlenecks. Additionally, by deploying the system for production systems, we received a valuable feedback loop where real applications exercise our system.

- Disaggregating the storage and caching layers enabled us to scale them separately. It has been a recurring theme in many design problems that bigger systems rely on other sub-systems as building blocks.
- A caching layer with high hit rates not only enables a fast response to the clients but also allows the storage layer to operate more cost-effectively. This is because the peak load with a cache will be substantially lower as compared to without-cache peak load (most services often provision storage layers for the peak load). However, much lower peaks are only sustainable when the overall system is in a steady state. If a system goes out of steady state and a large portion of caches go stale or die due to node restarts (for example, the widespread outage of Facebook systems in October 2021), special restart and cache warming mechanisms are required. Not taking such care can do more harm (for example, knocking out the storage layer due to excessive load).
- The scaled Memcache at Facebook has many fast paths to get keys (instead of slower path where clients go to the storage layer). Clients get good performance (low latency, high throughput) but we pay the cost in terms of numerous corner cases in dealing with data consistency, making the overall system complex.

In summary, we learned how a single-node system could be scaled for worldwide use by billions of users.

← Back
Cross-regional Level o...

Next →
Quiz on Memcache

Mark as Completed