Memcached at Facebook

Overview

- memcached is an open source implementation of in-memory hash table that Facebook used to construct a distributed kv store
  - Processes billions of requests per second
  - Stores trillions of items
  - Default memcached only provides a single-machine in-memory hash table
- Properties that influenced design:
  - Users consume significantly more data than they create, indicating caching has signifiant advantages
  - Read operations fetch data from a variety of sources, so caching strategy should be flexible
- memcached will refer to source code on a running binary, memcache will refer to the distributed system they built
- Two uses:
  - Query cache:
    - When web server needs data, first asks memcache for key
      - If it does not exist, query is made and cache populates
    - When write requests are made, it updates database and then deletes from cache (doesn't repopulate)
      - It doesn't repopulate because if memcache updates go in the wrong order as database updates, this can cause memcache to become permanently out of sync until the next write
  - Generic cache:
    - Use memcache as a general kv store to store pre-computed results from applications

Scaling in a Cluster

Reducing Latency

- At this scale, most efforts focus on reducing latency of fetching cached data or the load imposed due to a cache miss
- They provision hundreds of memcached servers in a cluster and use consistent hashing to shard the kv store
  - As a result, all web servers have to communicate with every memcached server in a short period of time
  - This can lead to a single server becoming a bottleneck for many webservers
  - Data replication can help but leads to significant memory inefficiency
- To reduce latency, mainly focus on the memcache client that runs on each webserver
  - Create a DAG representing data needed to be fetched and use it to maximize the amount of parallel requests
  - For GET requests, use UDP to communicate with servers which is much faster and doesn't require setup / maintaining connection
    - We can drop packets and get out of order, but that's fine
  - For SET / DELETE, use TCP since we need the reliable data to be sent
  - Clients use a sliding window to avoid incast congestion / overloading the servers
    - Use something similar to TCP's congestion control where the size of this sliding window grows upon success and shrinks upon fail
    - This sliding window applies to all of a client's requests to any server, not just to a specific one

Reducing Load

- We want to reduce frequency of fetching data along more expensive paths
- Leases:

    - We have two main problems:
        - Stale sets: a web server sets a value in memcache that isn't the most up to date one (due to concurrent updates that get reordered weirdly)
        - Thundering herd: a specific key undergoes heavy read / write activity which causes cache to stale fast
    - To fix both of these, we use a lease system
        - When the client has a cache miss, the client is given a lease to set data
        - We can only make queries to the DB and consequently write to cache while we have this lease
            - This removes problem of stale set
        - We also add a limit to the rate at which the server gives leases (i.e. once every 10 seconds)
            - This mitigates thundering herds
        - If stale values are more or less fine in a given application, we can set a setting that allows clients to access recently deleted items and serve them instead (so it serves data it knows is stale, but that's fine)
- Memcache Pools
    - Separate keys based on their characteristics into different Memcache pools / clusters
    - For instance, designate one pool (named wildcard) as the default
        - Designate other pools whose presence in wildcard can be problematic
        - I.e. a small pool for keys accessed frequently but have inexpensive cache miss
        - Large pool for infrequently accessed keys whose cache misses are very very expensive
- Replication within pools
    - For some pools, use replication when:
        - The app routine fetches many keys simultaneously
        - The entire data fits in 1-2 memcached servers
        - The request rate is much higher than the throughput of a single server
    - In this case, we want to use replication over dividing the key space

Handling Failures

- If an entire cluster has to be taken offline, we divert user web reqests to another cluster which removes all load from memcache within that cluster
- For small outages, after a few minutes, we use a small set of machines named Gutter to take over the responsibilities of a few failed servers
    - The client, upon receiving no response from memcached, just goes to the Gutter machine instead
    - We shunt load to idle servers to avoid the risk of rehashing keys amount remaining memcached servers because of non-uniform key access frequency (a hot key might cause another server to be overloaded)
    - One key is that Gutter servers do not delete from their cache, they just expire fast
        - This is because it would require too much overhead to delete from Gutter since Gutter machines can store any key, so every write to the DB would have to invalidate all Gutter machines' cache

Scaling in a Region

- Just buying more servers to scale a cluster doesn't work because highly requested items only become more popular as more webservers are added
- Instead we split a cluster into smaller clusters, which defines a region
    - We have multiple frontend clusters and then a storage cluster
- Storage cluster holds authoritative copy of data
    - User demand may replicate data into frontend clusters
    - Storage cluster has to invalidate cached data to keep frontend clusters consistent
    - To do this, a user write request has to also delete from its local memcached cache after modifying the storage cluster
    - Uses daemons called mcsqueal that analyze the committed SQL statements and broadcasts these updates to the memcache deployment in every frontend cluster
        - To reduce the packet rate of these updates, we batch these updates and send them to intermediate instances that then send disperse these updates

        - Chose to use this option over web servers notifying because the mcsqueal pipeline is more efficient at batching deletes and also prevents errors
- Each frontend cluster independently caches data based on the user requests sent to it
    - However, this replicates some data in the cache, so we can have multiple frontend clusters share the same set of memcached servers
    - They form a regional pool
- When clusters are first brought back online, the caches have poor hit rates
    - To mitigate this, we first retrieve data from a "warm cluster" to get a cluster back to full capacity instead of a few days
    - We get data from an existing cluster but we set the data into the cache of the new cluster

Across Regions

- Each region consists of a storage cluster and several frontend clusters
    - We designate one region to hold the master databases and others to contain read-only replicas
    - Use MySQL's replication to keep replica databases up-to-date with masters
- Maintaining consistency between data in memcache and persistent storage is primary challenge because replica databases can lag behind the master
    - Facebook aims for best-effort eventual consistency
- When we write from the master region, the fact that the daemons that handle invalidiating caches analyze the SQL database for this helps avoid the race condition in which an invalidation arrives before the data has been replicated
- Writes from a non-master region cause issues because what if a user makes another request that doesn't see the recent change because of replication lag
    - To min probability of reading stale data, use a remote marker that indicates the data in local replica is potentially stale and the query should be directed to master region
    - When a web server wants to update data, it has to set a remote marker, then performs write to master, and then deletes data in local cluster
        - On a subsequent request, the web server will miss cache, and then see the remote marker when it asks replica database, and then ask the master

Single Server Optimizations

- Automatically expand the hash table to avoid lookup times drifting to $O(n)$
- Make server multi-threaded with a global lock
- Give each thread its own UDP port
- Other things...

```
Q: What is the "stale set" problem in 3.2.1, and how do leases solve it?

A: Here's an example of the "stale set" problem that could occur if
there were no leases:

1. Client C1 asks memcache for k; memcache says k doesn't exist.
2. C1 asks MySQL for k, MySQL replies with value 1.
   C1 is slow at this point for some reason...
3. Someone updates k's value in MySQL to 2.
4. MySQL/mcsqueal/mcrouter send an invalidate for k to memcache,
   though memcache is not caching k, so there's nothing to invalidate.
5. C2 asks memcache for k; memcache says k doesn't exist.
6. C2 asks MySQL for k, mySQL replies with value 2.
7. C2 installs k=2 in memcache.
8. C1 installs k=1 in memcache.

Now memcache has a stale version of k, and it may never be updated.

The paper's leases fix the example:
```

1. Client C1 asks memcache for k; memcache says k doesn't exist,
   returns lease L1 to C1, and remembers the lease.
2. C1 asks MySQL for k, MySQL replies with value 1.
   C1 is slow at this point for some reason...
3. Someone updates k's value in MySQL to 2.
4. MySQL/mcsqueal/mcrouter send an invalidate for k to memcache,
   though memcache is not caching k, so there's nothing to invalidate.
   But memcache does invalidate C1's lease L1 (deletes L1 from its set
   of valid leases).
5. C2 asks memcache for k; memcache says k doesn't exist,
   and returns lease L2 to C2 (since there was no current lease for k).
6. C2 asks MySQL for k, mySQL replies with value 2.
7. C2 installs k=2 in memcache, supplying valid lease L2.
8. C1 installs k=1 in memcache, supplying invalid lease L1,
   so memcache ignores C1.

Now memcache is left caching the correct k=2.

Q: What is the "thundering herd" problem in 3.2.1, and how do leases
solve it?

A: The thundering herd problem:

* key k is very popular -- lots of clients read it.
* ordinarily clients read k from memcache, which is fast.
* but suppose someone writes k, causing it to be invalidated in memcache.
* for a while, every client that tries to read k will miss in memcache.
* they will all ask MySQL for k.
* MySQL may be overloaded with too many simultaneous requests.

The paper's leases solve this problem by allowing only the first
client that misses to ask MySQL for the latest data.