CS 695: Virtualization and Cloud Computing

Lecture 16: In-memory caching: Memcache

Mythili Vutukuru

IIT Bombay

Spring 2021

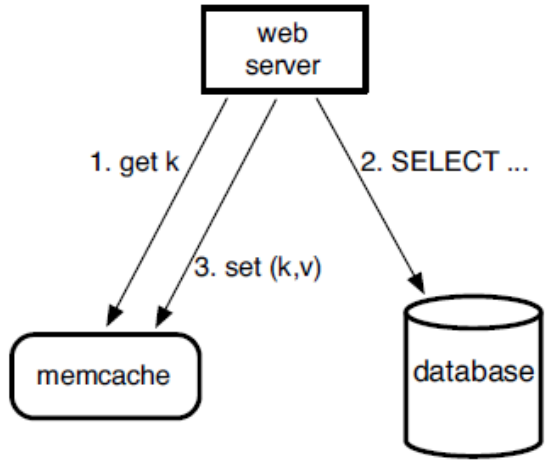
Facebook's Memcache

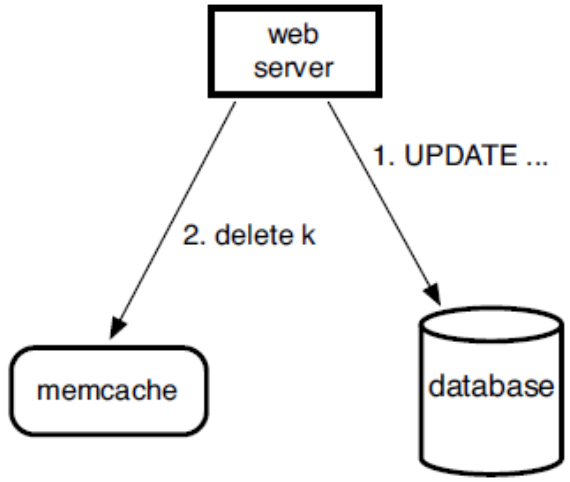
- Facebook starts with a single host version of in-memory key-value store (memcached) and builds a distributed, scalable in-memory caching system (memcache)
  - High performance of billions of requests per second
- Cache sits between web/application servers and backend databases
  - Generic cache that can be used across applications

Scaling Memcache at Facebook
Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, Venkateshwaran Venkataramani

Memcache

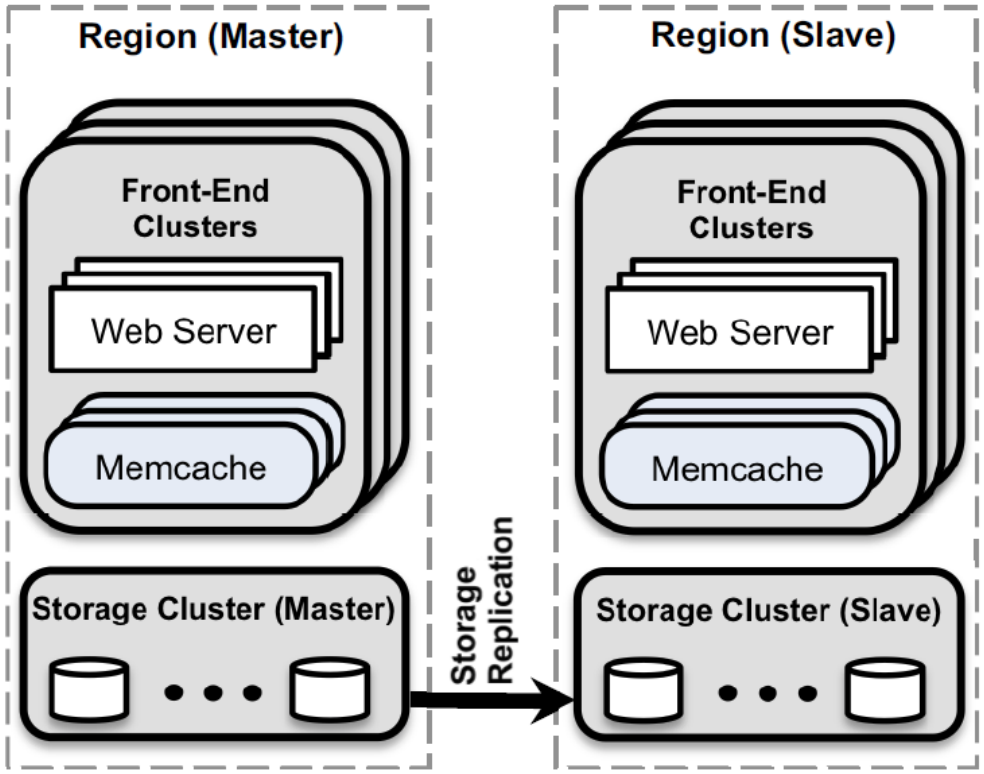
- Demand-filled look-aside cache
  - Cache results of queries to backend databases
- Web server reads data:
  - Look up cache, fetch if available
  - If cache miss, fetch from backend, populate cache
- Web server writes data
  - Write directly to backend
  - Invalidates cache data
- Difference from other No-SQL stores (Dynamo): memcache used for read-heavy workloads, not expected to be persistent/authentic source of data





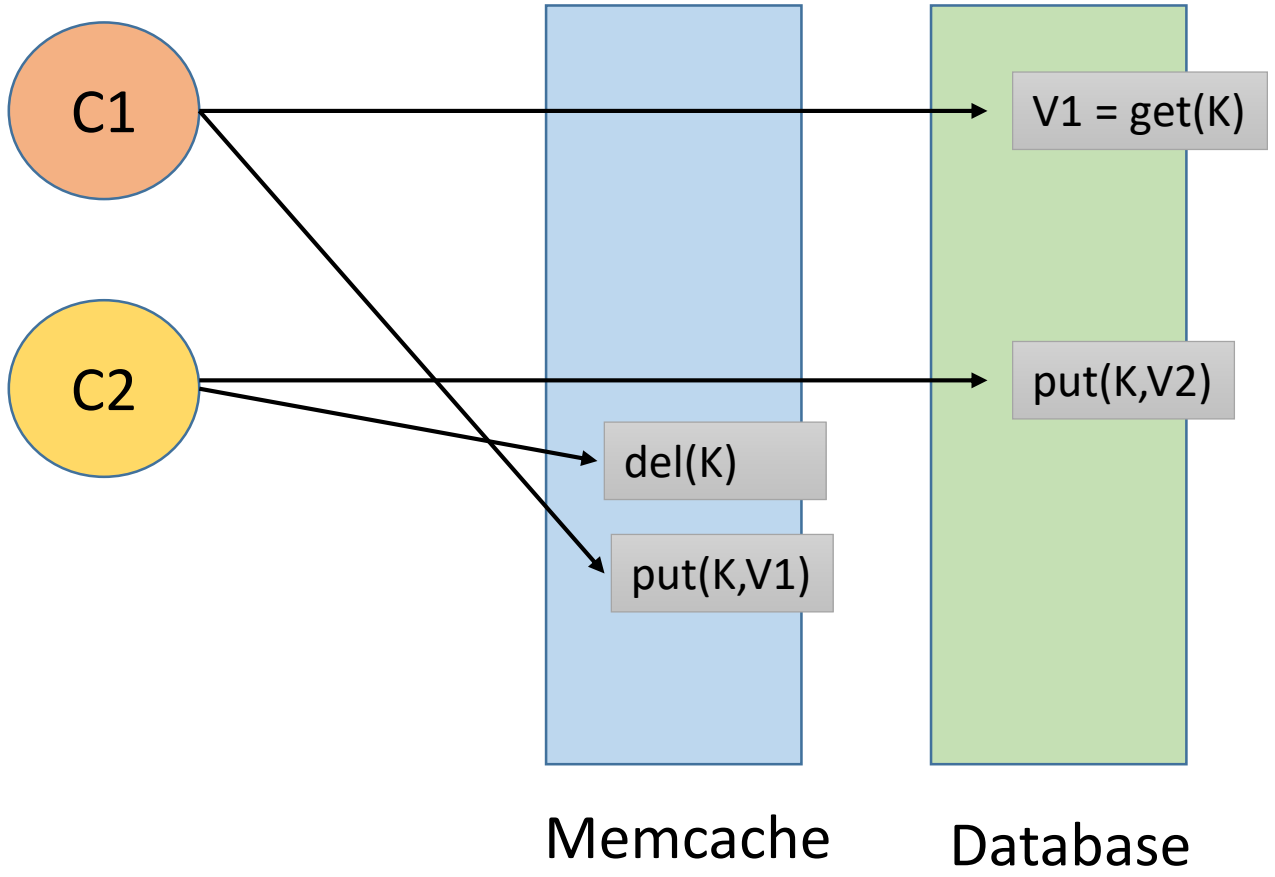
Overall architecture

- Front end clusters:
  - Web server (memcache clients)
  - Memcache (memcached servers)
- Backend storage cluster
  - MySQL databases
- Frontend and backend clusters arranged into regions
  - Region is a failure domain
- Keys divided between memcached servers by consistent hashing
- Web servers (memcache clients) contact the server responsible for a key via a client-side library or a proxy
  - Get requests over UDP (with a sliding window for flow control), put requests over TCP



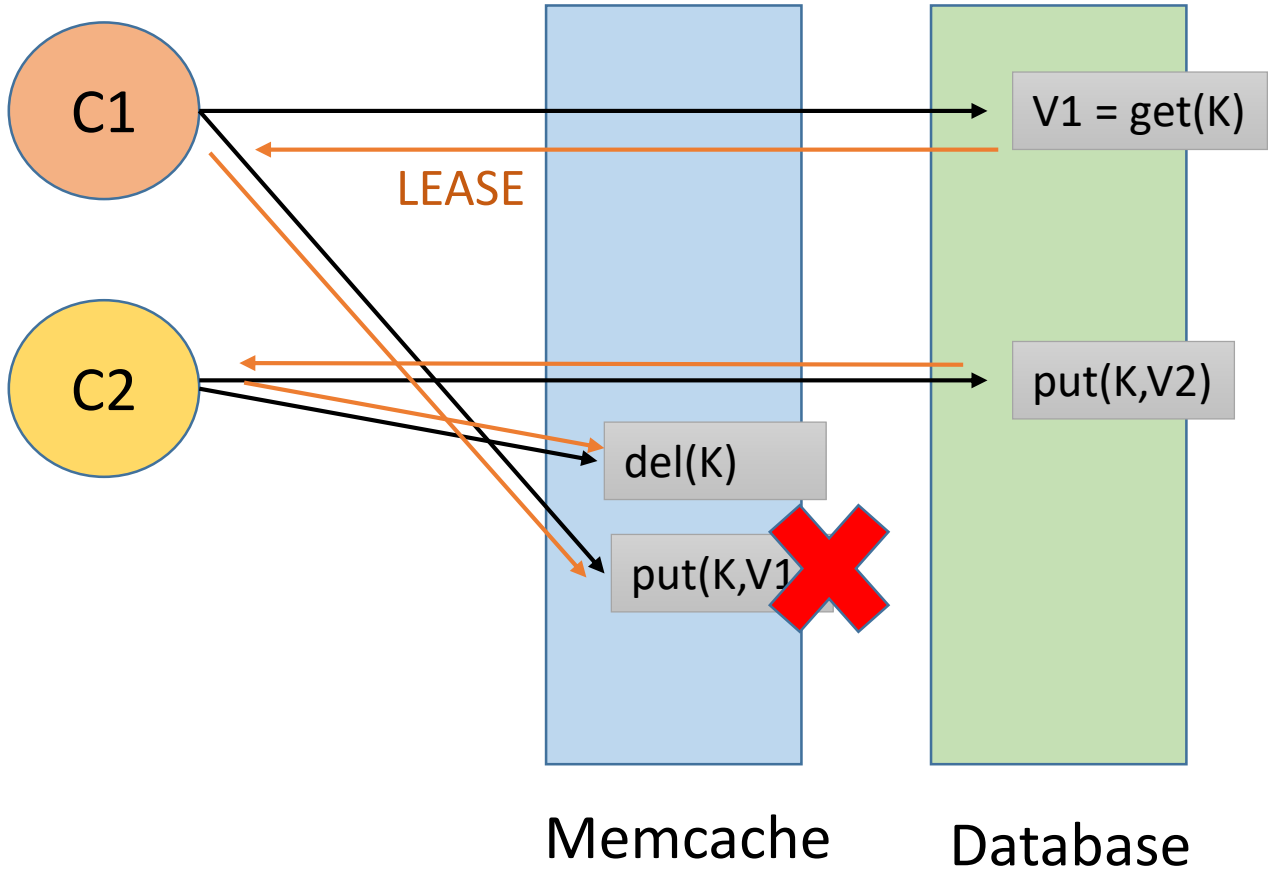
Handling stale values in cache

- C1 gets key K, misses in cache, gets V1 from DB
- C1 puts this value V1 into cache but this put is delayed
- Meanwhile, C2 puts new value V2 into DB and invalidates key K in cache
- Put(K,V1) arrives after del(K)
- Cache contains (K,V1) while DB contains (K,V2)
  - Inconsistent values



Handling stale values in cache: Solution

- When C1 puts value V1 in database, it gets a lease (64-bit version number)
  - This lease to be shown when performing put into cache
- C2 also gets a lease with a higher sequence number
- C1's put wont be accepted at the cache since it has an expired (older) lease
- More mechanisms in the paper on handling consistency across regions (not covered in lecture)



Avoiding the thundering herd

- C0 writes to a key and invalidates cached copy
- C1, C2, .. Cn all perform reads to the key in a short while after invalidation, will miss in cache
- It is enough for one of them to fetch from backend
  - C1 fetches from backend and populates cache
  - C2, ..Cn read from cache – this is ideal scenario
- However, C2, ..Cn may all contact the backend before C1 puts in cache
  - Thundering herd problem
  - Large number of accesses to backend right after cache invalidation
- Fix: C1 is given a lease by backend, but no further leases issues to C2, ..Cn
  - Issue a lease no more than once every 10 seconds
  - C2,..Cn told to wait and check cache later
- Alternately, return stale values for a short time after invalidation (if app permits)

Server “pools”

- Memcache stores different types of keys in different "pools" of servers
  - E.g., some application keys have low churn and long life, others have high churn and short life. If such keys stored together, the high churn keys can replace the low churn keys, which is undesirable
  - Separate key pools for different types of keys avoids impact of one type of application/workload on another
- What happens when servers fail?
  - Permanent failure: remap keys of failed server to another server. Problem: overload at new server (especially if a hot key is remapped)
  - To avoid remapping keys (for transient failures in particular), use a temporary "gutter pool" of servers
  - If a client finds that its assigned server has failed, it gets the key-value pair from the database and puts it into the gutter pool. Other clients also check the gutter pool when they discover the server failure.

Handling server overload

- Suppose caching traffic to server is 1M req/s, server capacity is only 500K req/s. What to do?
- One solution: split keyspace of server. Add another server and give away some keys to new server.
  - Many requests are “multiget”, e.g., client fetches 100 keys together
  - In such cases, both servers will see 1M req/s, but with fewer keys in each get request. Overall load stays same (serving 100 keys or 50 keys incurs similar overhead).
- Another solution: replication. Replicate key-value pairs across two servers
  - Each server gets 500K req/s, each requesting multiple keys
- Replication is better than splitting key space when lot of keys requested together in multiget
  - Splitting key space is not always best solution

Single memcached server optimizations

- Starting point: single memcached server with fixed size hash table
- Automatic expansion of hash table to prevent lookup times from becoming O(n)
- Make server multithreaded with fine-grained locking
- Each thread has a separate UDP port to listen for get requests, to avoid contention
- Slab allocators of various sizes to reduce dynamic memory allocation overheads
  - Adaptive slab sizes to match workload (slabs which are seeing more data will grow bigger)
- Proactively evict short-lived keys instead of waiting for them to be evicted via LRU
  - Short-lived keys stored in a separate transient item cache

Summary

- Techniques to build a caching layer between webservers and backend storage clusters
  - Reuse existing components (memcached server)
  - Add mechanisms to avoid inconsistent results
  - Simple mechanisms instead of stronger guarantees, for better scalability