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

Cesar Stuardo

THE UNIVERSITY OF CHICAGO
What is MemCache? [1/1]

- What is MemCached (or what was MemCached in 2013)?
  - High performance object caching
    - Fixed size Hash Table, Single threaded, Coarse locking

1-2 should be "cheaper" than 1-3
MemCache and Facebook [1/4]

Facebook
- Hundreds of millions of people use it every day and impose computational, network, and I/O demands
  - Billions of requests per second
  - Holds trillions of items

Main requirements
- Near realtime communication
- Aggregate content on the fly from multiple sources
  - Heterogeneity (e.g. HDFS, MySQL)
- Access and update popular content
  - A portion of the content might be heavily accessed and updated in a time window.
- Scale to process billions of user requests per second
MemCache and Facebook [2/4]

- **Workload characterization**
  - **Read Heavy**
    - Users consume more than they produce (read more than they write)
  - **Heterogeneity**
    - Multiple storage backends (e.g. HDFS, MySQL)
    - Each backend has different properties and constraints
      - Latency
      - Load
      - etc...
MemCache and Facebook [3/4]

Figure 1: Memcache as a demand-filled look-aside cache. The left half illustrates the read path for a web server on a cache miss. The right half illustrates the write path.

Look-Aside: Client controls cache (adds/deletes/updates data)
MemCache and Facebook [4/4]

Scaling MemCache in 4 steps

1. MemCache Single Server
2. MemCache Cluster
3. MemCache Region
4. MemCache Across Regions
MemCache: Single Server [1/3]

- Initially **single threaded with fixed size hash table**
- Optimizations
  - Automatic **size adaptation for hash table**
    - Fixed size hash tables can degenerate lookup time to \( O(n) \).
  - **Multithreaded**
    - Each thread can serve requests
    - Fine-grained locking
  - Each thread has **its own UDP port**
    - Avoid congestion when replying
    - No Incast
MemCache: Single Server [2/3]

- Memory Allocation
  - Originally, **slab classes** with different sizes. When memory ran out, **LRU** policy is used for eviction.
    - When slab class has no free elements, **a new slab is created**
    - **Lazy** eviction mechanism (when serving a request)

- Modifications
  - Adaptative
    - Tries to allocate considering “**needy**” slab classes
    - Slabs **move from one class to another** if age policy is met
    - Single global **LRU**
    - Lazy eviction for long-lived keys, **proactive eviction** for short-lived keys
MemCache: Single Server [3/3]

15 clients generating traffic to a single memcache server with 24 threads

Hit/Miss for different versions

Each request fetches 10 keys

UDP vs TCP performance

Figure 7: Multiget hit and miss performance comparison by memcached version

Figure 8: Get hit performance comparison for single gets and 10-key multigets over TCP and UDP
MemCache: Cluster [1/4]

- Data is partitioned using **consistent hashing**
  - Each node owns one or more partitions in the ring
- One request usually involves communication with multiple servers
  - **All-to-All** communication
  - **Latency and Load** become a concerns

- **Reducing Latency**
  - **Parallel** requests and **Batching**
  - **Sliding windows** for requests
  - **UDP** for get requests
    - If packets are out of order or missing then client deals with it
  - **TCP** for set/delete requests
    - Reliability

- Reducing Load
  - Leases
    - Arbitrate concurrent writes
      • Stale Sets
    - One token every 10 seconds
      • Thundering Herds
  - Pooling
    - For different workloads
    - For fault tolerance
      • Gutter Pool

Figure 5: Daily and weekly working set of a high-churn family and a low-churn key family
MemCache: Region [1/4]
MemCache: Region [2/4]

- **Positive**
  - Smaller Failure Domain
  - Network configuration
  - Reduction of incast

- **Negative**
  - Need for intra-region replication

- **Main challenges on replication**
  - Replication in a region: *Regional Invalidations*
  - Maintenance and Availability: *Regional Pools*
  - Maintenance and Availability: *Cold Cluster Warm-Up*
MemCache: Region [3/4]

Regional Invalidation

Daemon extracts delete statements from database and broadcasts to other memcache nodes.

Deletes are batched to reduce packet rates.

Figure 6: Invalidation pipeline showing keys that need to be deleted via the daemon (mcsqueueal).
MemCache: Region [4/4]

Maintenance and Availability

- **Regional Pools**
  - Requests are *randomly routed* to all clusters
    - Each cluster roughly has the same data
  - Multiple front end clusters *share the same memcache cluster*
    - Ease of maintenance when taking a *cluster offline*

- **Cold Cluster Warm-Up**
  - After maintenance, cluster is brought up but is *empty*
  - Cold Cluster takes *data from another cluster* and warms itself up
Fig. 2: Overall architecture
MemCache: Across Regions [2/3]

- **Positive**
  - Latency reduction (locality with users)
  - Geographic diversity and disasters
  - Always looking for cheaper places

- **Negative**
  - Inter-Region consistency is now a problem

- **Main challenges on consistency**
  - Inter-Region Consistency: **Master Region Writes**
  - Inter-Region Consistency: **Non-Master Region Writes**
MemCache: Across Regions [3/3]

Write consistency

- **From Master Region**
  - Not really a problem, *mcsqueal* avoids complex data races.
- **From Non-Master Region**
  - Remote markers
    - Set remote marker for a key
    - Perform the write to master (passing marker)
    - Delete in local cluster
  - Now *next request* can go to master if remote marker is found
MemCache: Workloads [1/3]

Figure 9: Cumulative distribution of the number of distinct memcached servers accessed.
MemCache: Workloads [2/3]
### MemCache: Workloads [3/3]

#### Table 2: Traffic per server on selected `memcache` pools averaged over 7 days

<table>
<thead>
<tr>
<th>pool</th>
<th>miss rate (%)</th>
<th>get(s)</th>
<th>set(s)</th>
<th>delete(s)</th>
<th>packets(s)</th>
<th>outbound bandwidth (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>wildcard</td>
<td>1.76%</td>
<td>262k</td>
<td>8.26k</td>
<td>21.2k</td>
<td>236k</td>
<td>57.4</td>
</tr>
<tr>
<td>app</td>
<td>7.85%</td>
<td>96.5k</td>
<td>11.9k</td>
<td>6.28k</td>
<td>83.0k</td>
<td>31.0</td>
</tr>
<tr>
<td>replicated</td>
<td>0.053%</td>
<td>710k</td>
<td>1.75k</td>
<td>3.22k</td>
<td>44.5k</td>
<td>30.1</td>
</tr>
<tr>
<td>regional</td>
<td>6.35%</td>
<td>9.1k</td>
<td>0.79k</td>
<td>35.9k</td>
<td>47.2k</td>
<td>10.8</td>
</tr>
</tbody>
</table>

#### Table 3: Distribution of item sizes for various pools in bytes

<table>
<thead>
<tr>
<th>pool</th>
<th>mean</th>
<th>std dev</th>
<th>p5</th>
<th>p25</th>
<th>p50</th>
<th>p75</th>
<th>p95</th>
<th>p99</th>
</tr>
</thead>
<tbody>
<tr>
<td>wildcard</td>
<td>1.11 K</td>
<td>8.28 K</td>
<td>77</td>
<td>102</td>
<td>169</td>
<td>363</td>
<td>3.65 K</td>
<td>18.3 K</td>
</tr>
<tr>
<td>app</td>
<td>881</td>
<td>7.70 K</td>
<td>103</td>
<td>247</td>
<td>269</td>
<td>337</td>
<td>1.68 K</td>
<td>10.4 K</td>
</tr>
<tr>
<td>replicated</td>
<td>66</td>
<td>2</td>
<td>62</td>
<td>68</td>
<td>68</td>
<td>68</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>regional</td>
<td>31.8 K</td>
<td>75.4 K</td>
<td>231</td>
<td>824</td>
<td>5.31 K</td>
<td>24.0 K</td>
<td>158 K</td>
<td>381 K</td>
</tr>
</tbody>
</table>
Conclusions [1/2]

- **Lessons learned (by them)**
  - Separating cache and persistent storage systems allowed to **independently scale them**
  - Features that **improve monitoring, debugging and operational efficiency** are as important as performance
  - **Keeping logic in a stateless client** helps iterate on features and minimize disruption
  - The system must support **gradual rollout and rollback of new features** even if it leads to temporary heterogeneity of feature sets
Conclusions [2/2]

- Lessons Learned (by us)
  - Trade-off based design
    - Stale data for performance
    - Scalability for accuracy
  - Decoupled design focused on fast rollout
    - Ease of maintenance
    - Scalability
  - Contribution to the open source world

- but...
  - Why was it accepted to NSDI?
  - How does the paper contributed to the network community?
Thank you!

Questions?