Scaling Memcache at Facebook

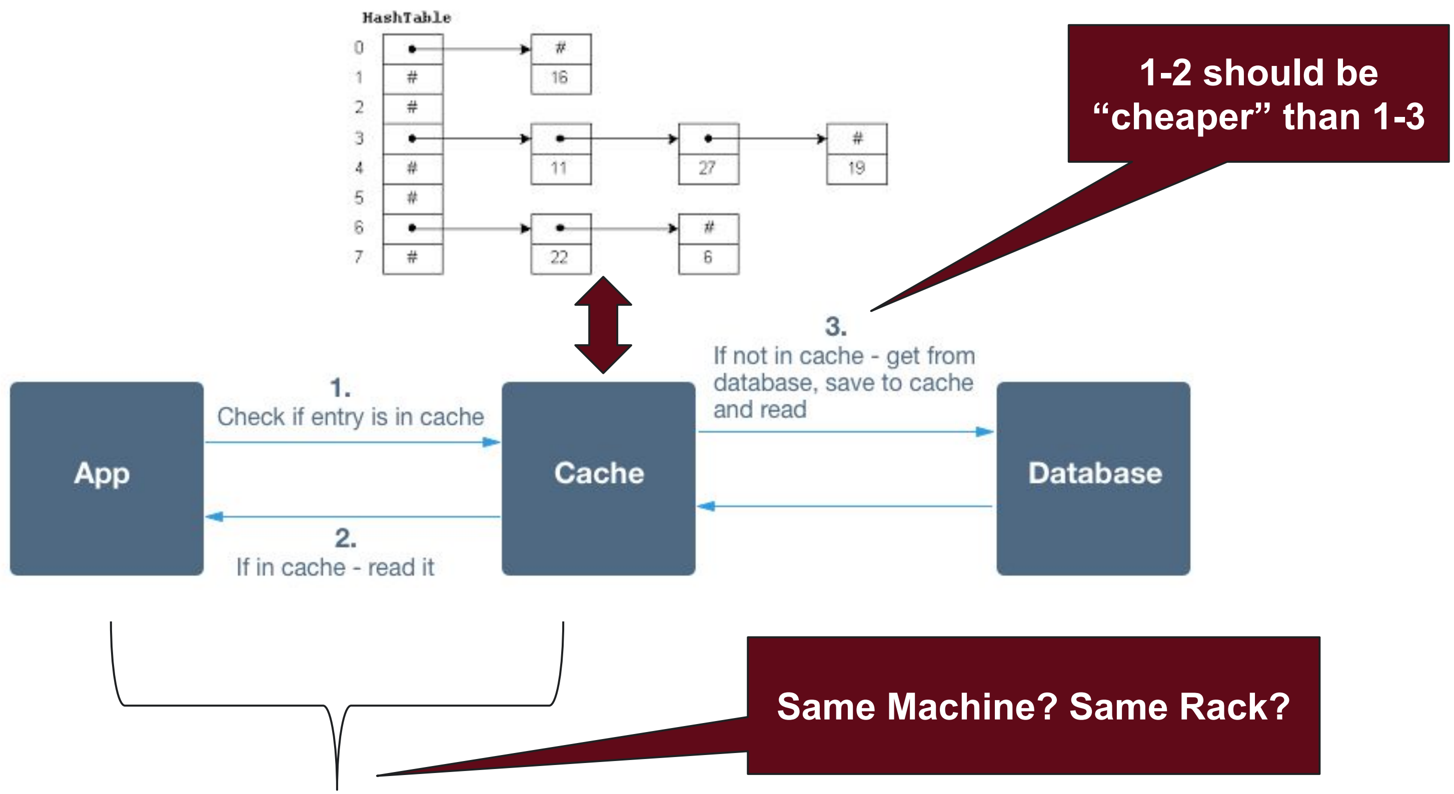
Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, Venkateshwaran Venkataramani

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



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]

Read

Write

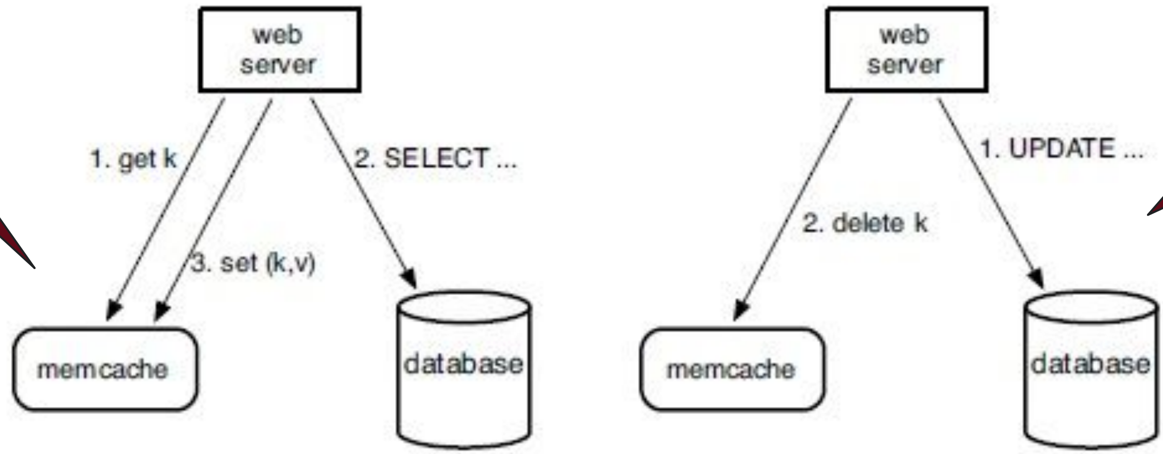
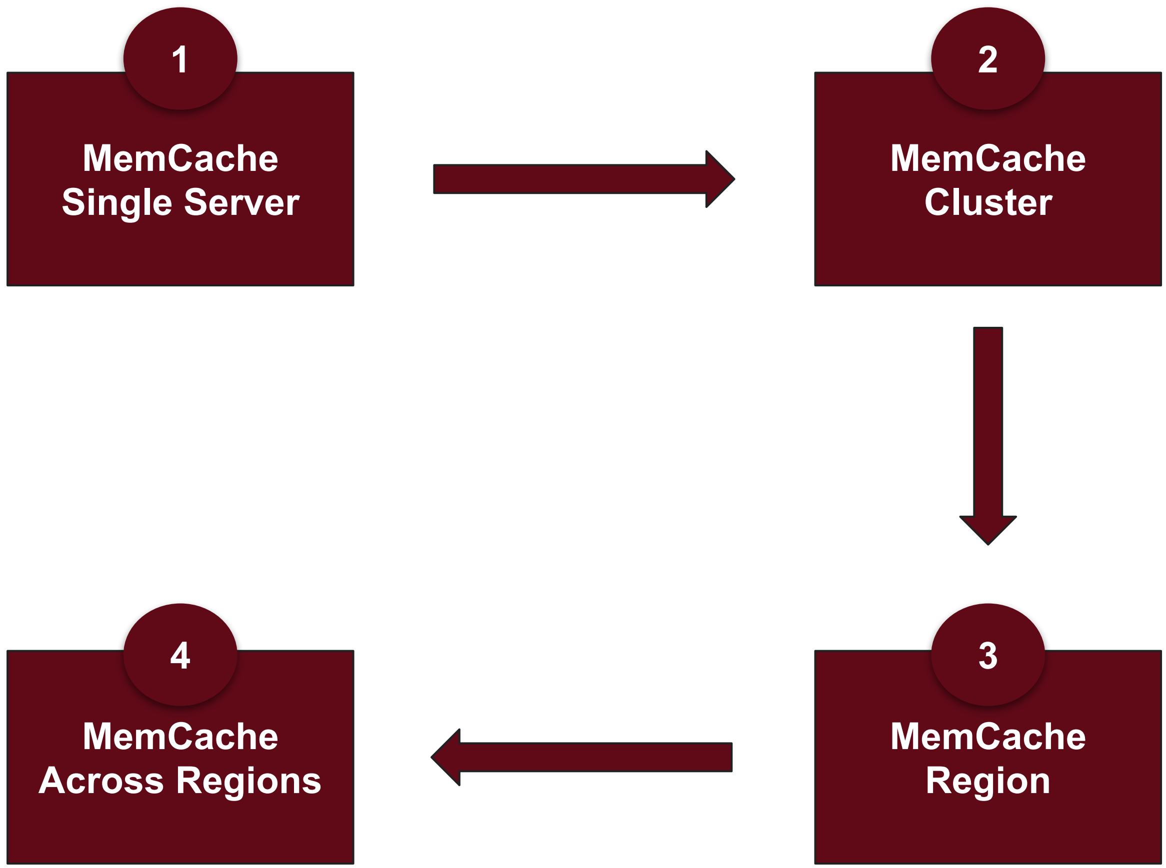


Figure 1: Memcache as a demand-filled look-aside cache. The left half illustrates the read path for a web server on a cache miss. The right half illustrates the write path.

Look-Aside: Client controls cache (adds/deletes/updates data)

MemCache and Facebook [4/4]

Scaling MemCache in 4 steps



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

Each request fetches 10 keys

Hit/Miss for different versions

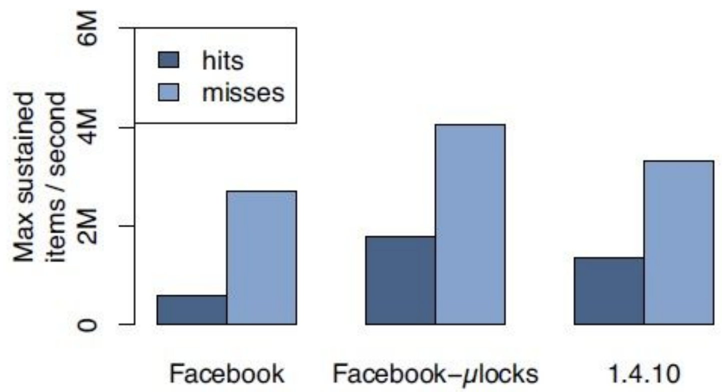


Figure 7: Multiget hit and miss performance comparison by memcached version

UDP vs TCP performance

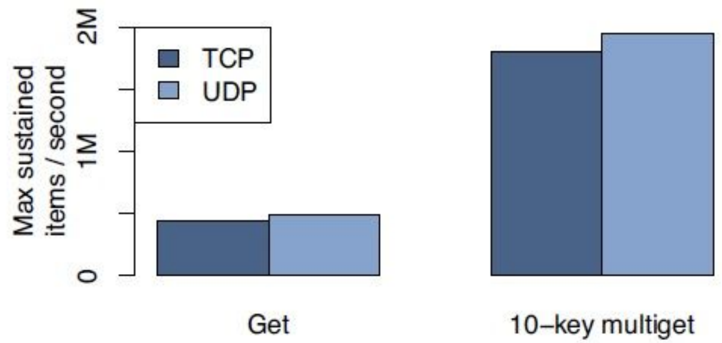
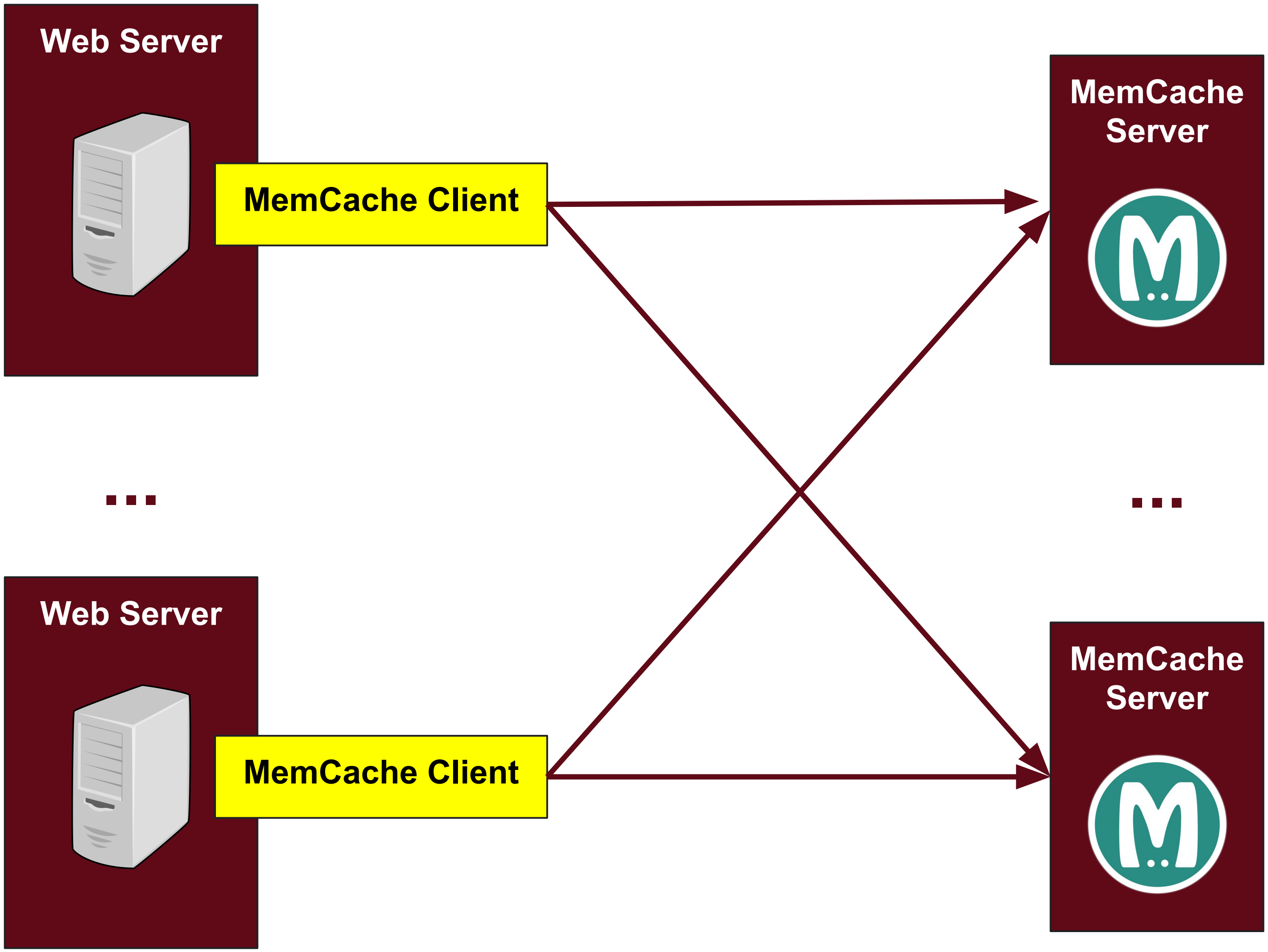


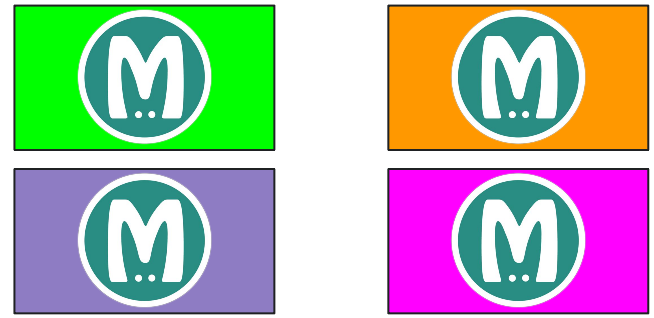
Figure 8: Get hit performance comparison for single gets and 10-key multigets over TCP and UDP

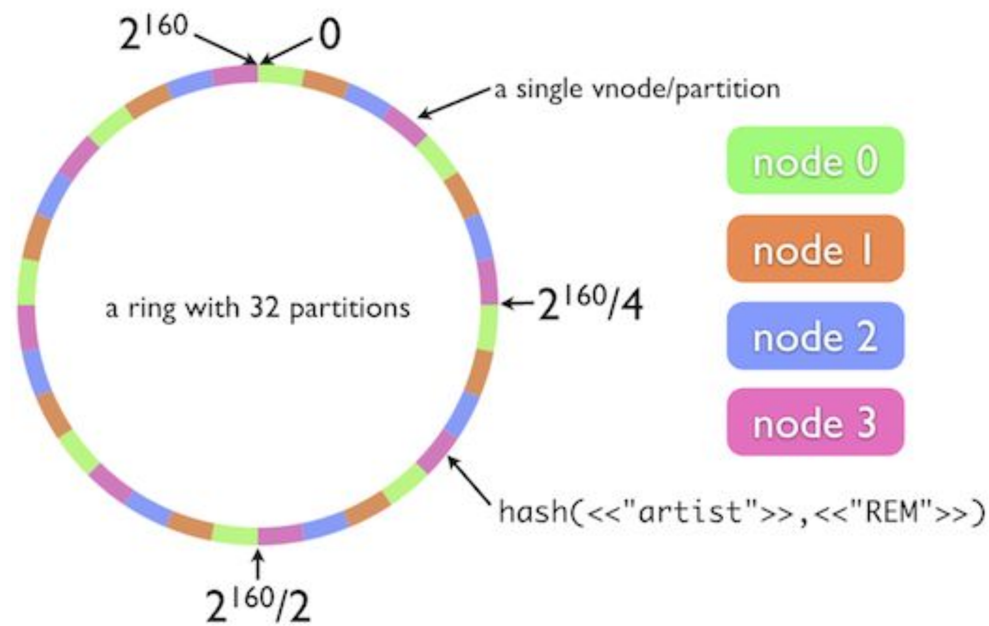
MemCache: Cluster [1/4]



MemCache: Cluster [2/4]

- 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





MemCache: Cluster [3/4]

- **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

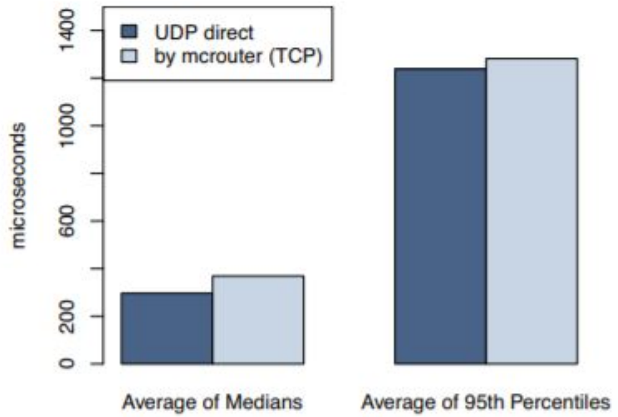


Figure 3: Get latency for UDP, TCP via mcrouter

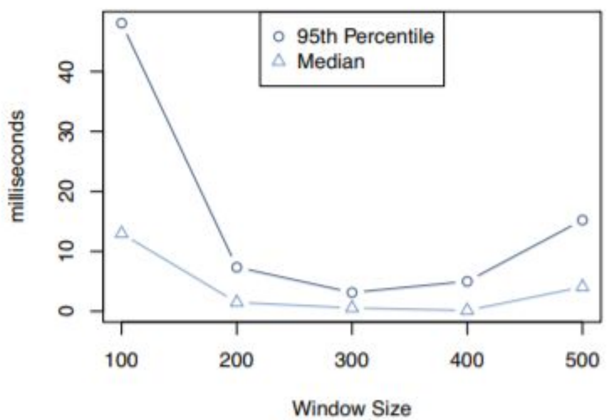


Figure 4: Average time web requests spend waiting to be scheduled

MemCache: Cluster [4/4]

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

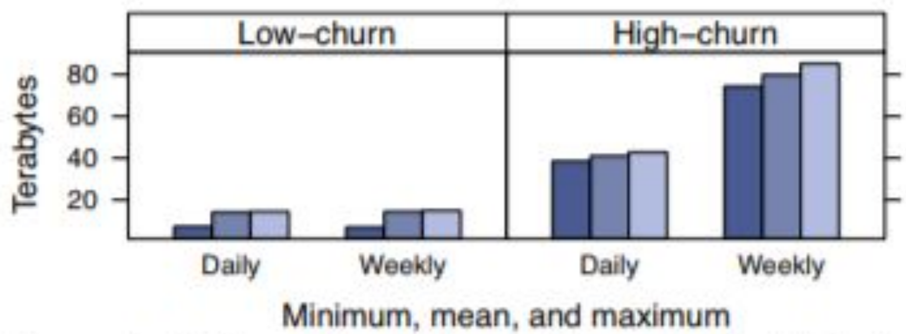
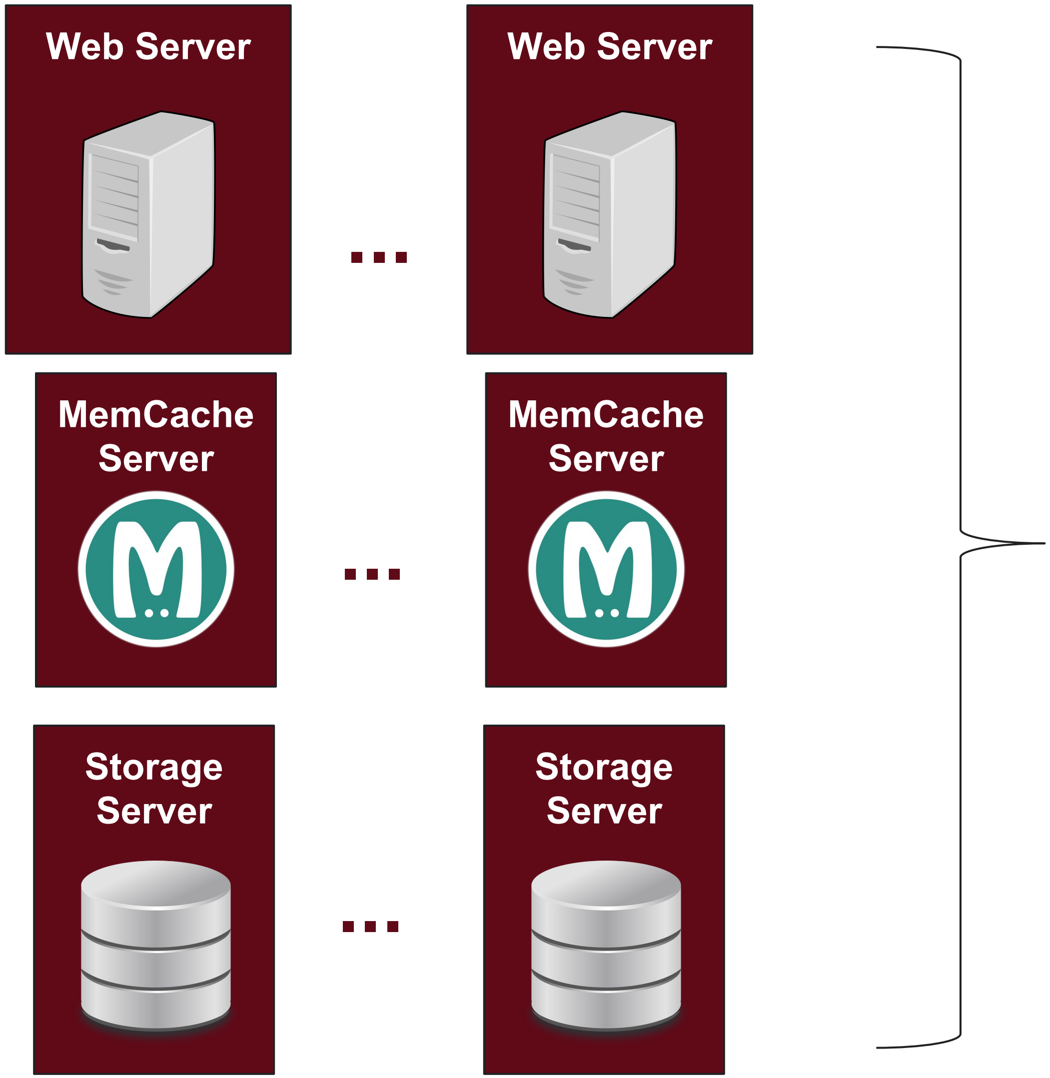


Figure 5: Daily and weekly working set of a high-churn family and a low-churn key family

MemCache: Region [1/4]



Web Servers
+
MemCache Servers
+
Storage Servers
=
Region

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

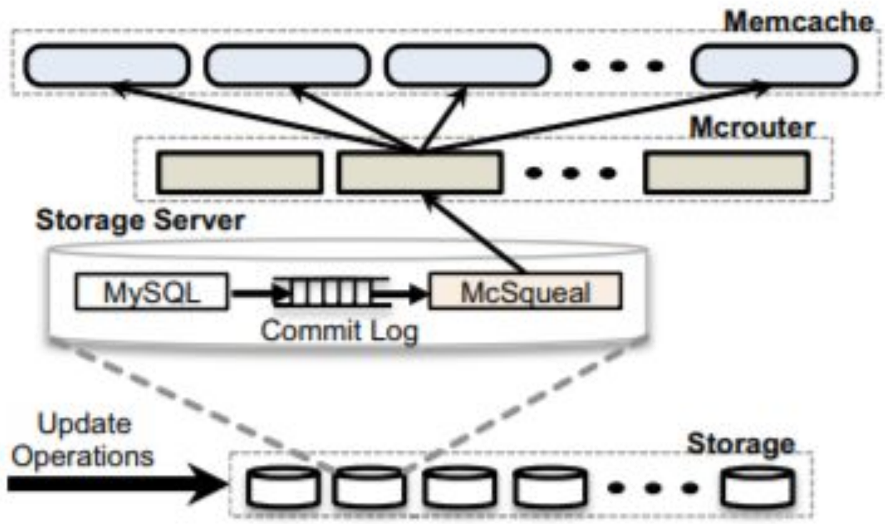


Figure 6: Invalidation pipeline showing keys that need to be deleted via the daemon (mcsqueal).

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

Deletes are batched to reduce packet rates

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

MemCache: Across Regions [1/3]

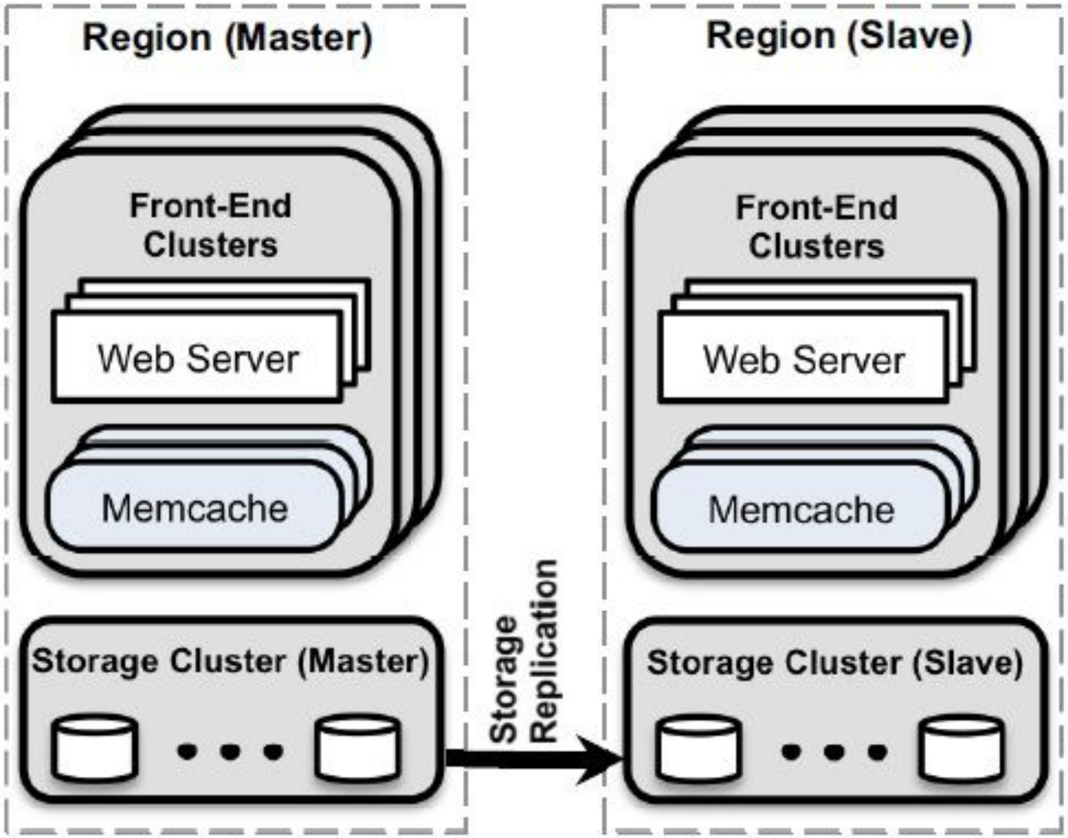


Figure 2: Overall architecture

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]

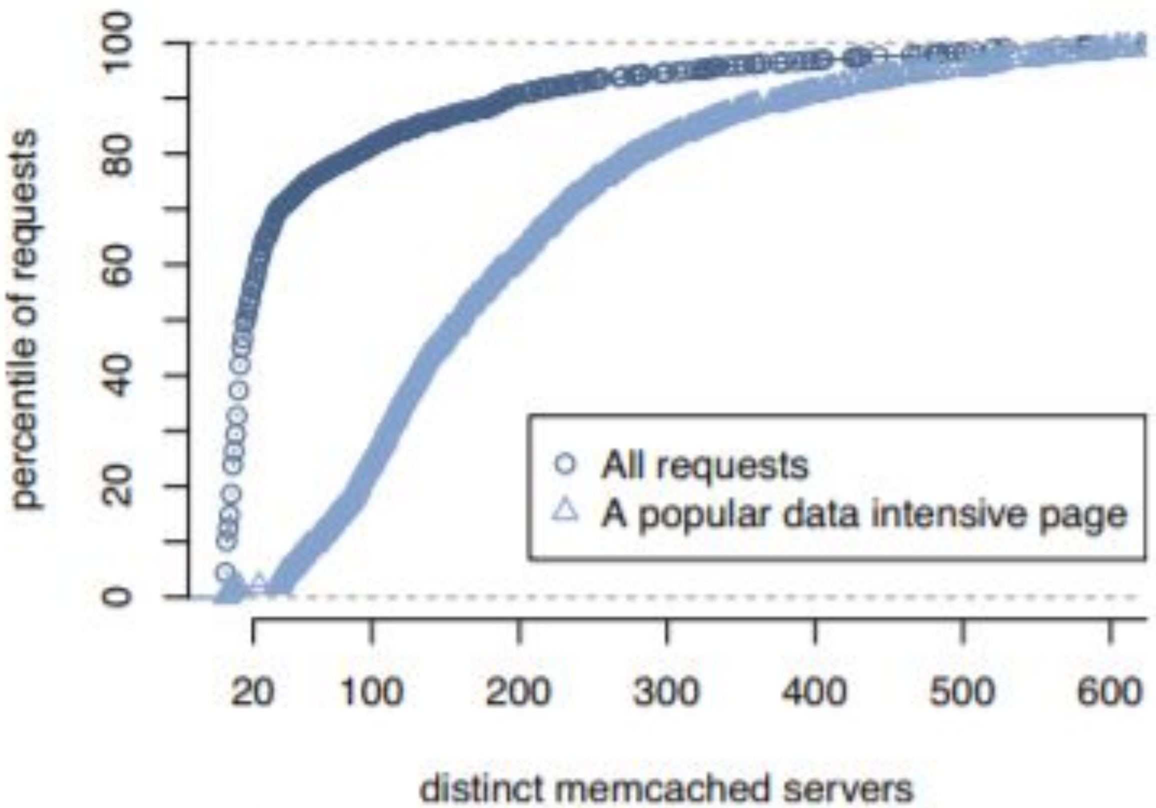
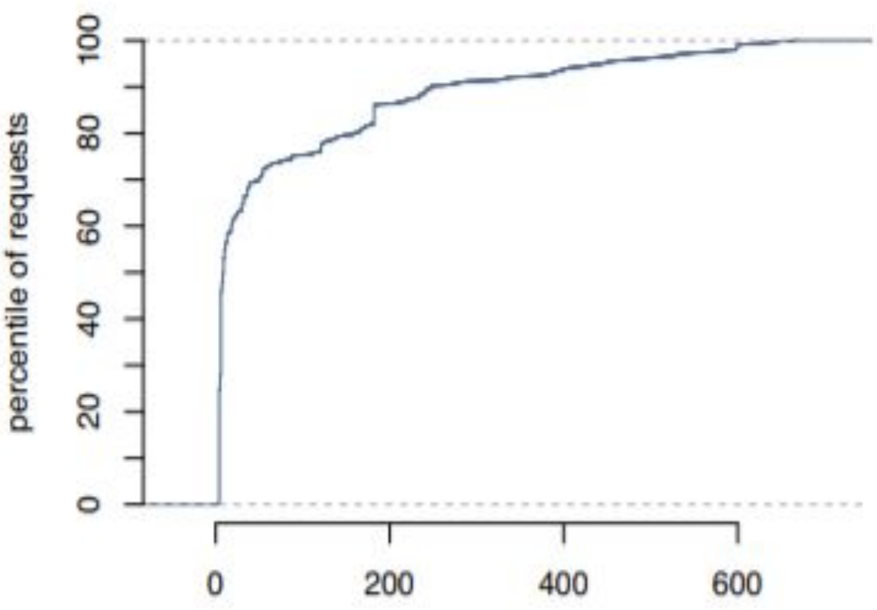
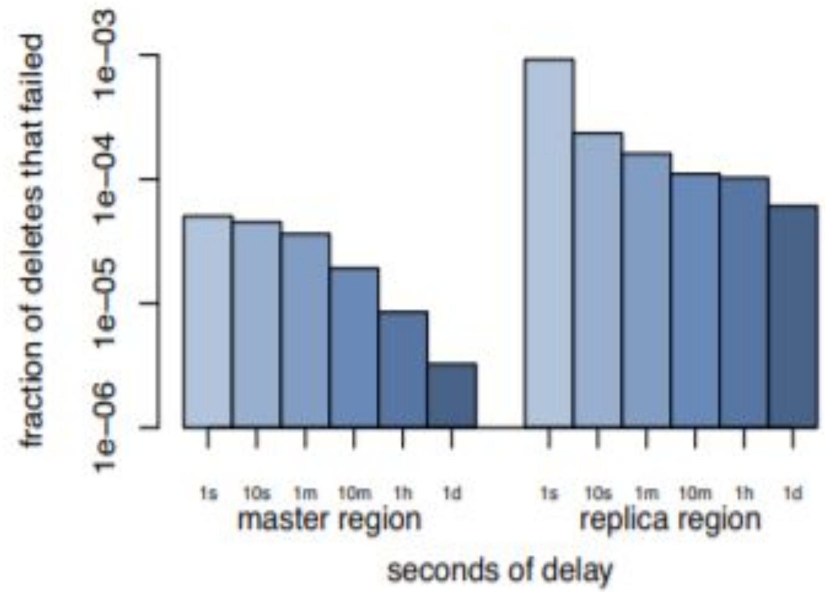


Figure 9: Cumulative distribution of the number of distinct memcached servers accessed

MemCache: Workloads [2/3]





MemCache: Workloads [3/3]

pool	miss rate	$\frac{get}{s}$	$\frac{set}{s}$	$\frac{delete}{s}$	$\frac{packets}{s}$	outbound bandwidth (MB/s)
wildcard	1.76%	262k	8.26k	21.2k	236k	57.4
app	7.85%	96.5k	11.9k	6.28k	83.0k	31.0
replicated	0.053%	710k	1.75k	3.22k	44.5k	30.1
regional	6.35%	9.1k	0.79k	35.9k	47.2k	10.8

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

pool	mean	std dev	p5	p25	p50	p75	p95	p99
wildcard	1.11 K	8.28 K	77	102	169	363	3.65 K	18.3 K
app	881	7.70 K	103	247	269	337	1.68K	10.4 K
replicated	66	2	62	68	68	68	68	68
regional	31.8 K	75.4 K	231	824	5.31 K	24.0 K	158 K	381 K

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

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?

THE UNIVERSITY OF CHICAGO