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

{rajeshn,hans}@fb.com, {sgrimm, marc}@facebook.com, {herman, hcli, rm, mpal, dpeek, ps, dstaff, ttung, veeve}@fb.com

Facebook Inc.

Abstract: `Memcached` is a well known, simple, in-memory caching solution. This paper describes how Facebook leverages `memcached` as a building block to construct and scale a distributed key-value store that supports the world's largest social network. Our system handles billions of requests per second and holds trillions of items to deliver a rich experience for over a billion users around the world.

1 Introduction

Popular and engaging social networking sites present significant infrastructure challenges. Hundreds of millions of people use these networks every day and impose computational, network, and I/O demands that traditional web architectures struggle to satisfy. A social network's infrastructure needs to (1) allow near real-time communication, (2) aggregate content on-the-fly from multiple sources, (3) be able to access and update very popular shared content, and (4) scale to process millions of user requests per second.

We describe how we improved the open source version of `memcached` [14] and used it as a building block to construct a distributed key-value store for the largest social network in the world. We discuss our journey scaling from a single cluster of servers to multiple geographically distributed clusters. To the best of our knowledge, this system is the largest `memcached` installation in the world, processing over a billion requests per second and storing trillions of items.

This paper is the latest in a series of works that have recognized the flexibility and utility of distributed key-value stores [1, 2, 5, 6, 12, 14, 34, 36]. This paper focuses on `memcached`—an open-source implementation of an in-memory hash table—as it provides low latency access to a shared storage pool at low cost. These qualities enable us to build data-intensive features that would otherwise be impractical. For example, a feature that issues hundreds of database queries per page request would likely never leave the prototype stage because it would be too slow and expensive. In our application, however, web pages routinely fetch thousands of key-value pairs from `memcached` servers.

One of our goals is to present the important themes that emerge at different scales of our deployment. While qualities like performance, efficiency, fault-tolerance, and consistency are important at all scales, our experience indicates that at specific sizes some qualities require more effort to achieve than others. For example, maintaining data consistency can be easier at small scales if replication is minimal compared to larger ones where replication is often necessary. Additionally, the importance of finding an optimal communication schedule increases as the number of servers increase and networking becomes the bottleneck.

This paper includes four main contributions: (1) We describe the evolution of Facebook's `memcached`-based architecture. (2) We identify enhancements to `memcached` that improve performance and increase memory efficiency. (3) We highlight mechanisms that improve our ability to operate our system at scale. (4) We characterize the production workloads imposed on our system.

2 Overview

The following properties greatly influence our design. First, users consume an order of magnitude more content than they create. This behavior results in a workload dominated by fetching data and suggests that caching can have significant advantages. Second, our read operations fetch data from a variety of sources such as MySQL databases, HDFS installations, and backend services. This heterogeneity requires a flexible caching strategy able to store data from disparate sources.

`Memcached` provides a simple set of operations (`set`, `get`, and `delete`) that makes it attractive as an elemental component in a large-scale distributed system. The open-source version we started with provides a single-machine in-memory hash table. In this paper, we discuss how we took this basic building block, made it more efficient, and used it to build a distributed key-value store that can process billions of requests per second. Hence-

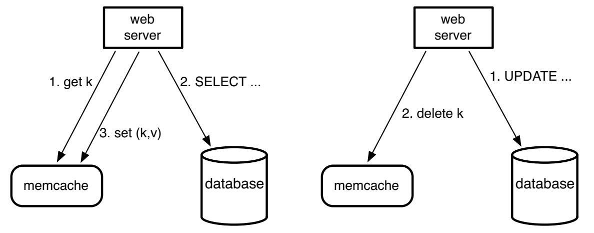

Figure 1: `Memcache` as a demand-filled look-aside cache. The left half illustrates the read path for a web server on a cache miss. The right half illustrates the write path.

forth, we use '`memcached`' to refer to the source code or a running binary and '`memcache`' to describe the distributed system.

Query cache: We rely on `memcache` to lighten the read load on our databases. In particular, we use `memcache` as a *demand-filled look-aside* cache as shown in Figure 1. When a web server needs data, it first requests the value from `memcache` by providing a string key. If the item addressed by that key is not cached, the web server retrieves the data from the database or other backend service and populates the cache with the key-value pair. For write requests, the web server issues SQL statements to the database and then sends a delete request to `memcache` that invalidates any stale data. We choose to delete cached data instead of updating it because deletes are idempotent. `Memcache` is not the authoritative source of the data and is therefore allowed to evict cached data.

While there are several ways to address excessive read traffic on MySQL databases, we chose to use `memcache`. It was the best choice given limited engineering resources and time. Additionally, separating our caching layer from our persistence layer allows us to adjust each layer independently as our workload changes.

Generic cache: We also leverage `memcache` as a more general key-value store. For example, engineers use `memcache` to store pre-computed results from sophisticated machine learning algorithms which can then be used by a variety of other applications. It takes little effort for new services to leverage the existing marcher infrastructure without the burden of tuning, optimizing, provisioning, and maintaining a large server fleet.

As is, `memcached` provides no server-to-server coordination; it is an in-memory hash table running on a single server. In the remainder of this paper we describe how we built a distributed key-value store based on `memcached` capable of operating under Facebook's workload. Our system provides a suite of configuration, aggregation, and routing services to organize `memcached` instances into a distributed system.

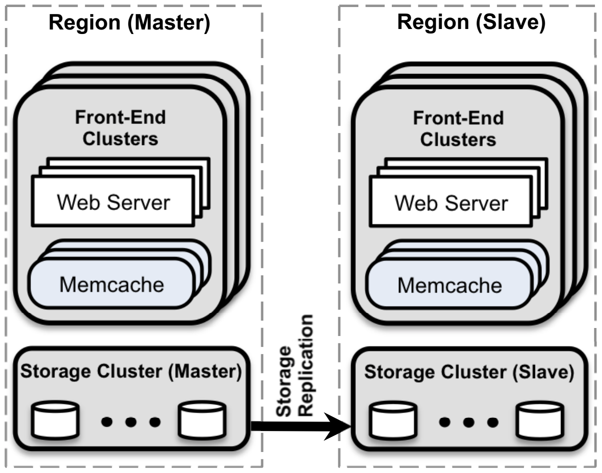

Figure 2: Overall architecture

We structure our paper to emphasize the themes that emerge at three different deployment scales. Our read-heavy workload and wide fan-out is the primary concern when we have one cluster of servers. As it becomes necessary to scale to multiple frontend clusters, we address data replication between these clusters. Finally, we describe mechanisms to provide a consistent user experience as we spread clusters around the world. Operational complexity and fault tolerance is important at all scales. We present salient data that supports our design decisions and refer the reader to work by Atikoglu *et al.* [8] for a more detailed analysis of our workload. At a high-level, Figure 2 illustrates this final architecture in which we organize co-located clusters into a region and designate a master region that provides a data stream to keep non-master regions up-to-date.

While evolving our system we prioritize two major design goals. (1) Any change must impact a user-facing or operational issue. Optimizations that have limited scope are rarely considered. (2) We treat the probability of reading transient stale data as a parameter to be tuned, similar to responsiveness. We are willing to expose slightly stale data in exchange for insulating a backend storage service from excessive load.

3 In a Cluster: Latency and Load

We now consider the challenges of scaling to thousands of servers within a cluster. At this scale, most of our efforts focus on reducing either the latency of fetching cached data or the load imposed due to a cache miss.

3.1 Reducing Latency

Whether a request for data results in a cache hit or miss, the latency of `memcache`'s response is a critical factor in the response time of a user's request. A single user web request can often result in hundreds of individual memcache get requests. For example, loading one of our popular pages results in an average of 521 distinct items fetched from memcache.[1]

We provision hundreds of memcached servers in a cluster to reduce load on databases and other services. Items are distributed across the memcached servers through consistent hashing [22]. Thus web servers have to routinely communicate with *many* memcached servers to satisfy a user request. As a result, *all* web servers communicate with every memcached server in a short period of time. This *all-to-all* communication pattern can cause incast congestion [30] or allow a single server to become the bottleneck for many web servers. Data replication often alleviates the single-server bottleneck but leads to significant memory inefficiencies in the common case.

We reduce latency mainly by focusing on the memcache client, which runs on each web server. This client serves a range of functions, including serialization, compression, request routing, error handling, and request batching. Clients maintain a map of all available servers, which is updated through an auxiliary configuration system.

Parallel requests and batching: We structure our web-application code to minimize the number of network round trips necessary to respond to page requests. We construct a directed acyclic graph (DAG) representing the dependencies between data. A web server uses this DAG to maximize the number of items that can be fetched concurrently. On average these batches consist of 24 keys per request[2].

Client-server communication: Memcached servers do not communicate with each other. When appropriate, we embed the complexity of the system into a stateless client rather than in the memcached servers. This greatly simplifies memcached and allows us to focus on making it highly performant for a more limited use case. Keeping the clients stateless enables rapid iteration in the software and simplifies our deployment process. Client logic is provided as two components: a library that can be embedded into applications or as a standalone proxy named mcrouter. This proxy presents a memcached server interface and routes the requests/replies to/from other servers.

Clients use UDP and TCP to communicate with memcached servers. We rely on UDP for get requests to reduce latency and overhead. Since UDP is connectionless, each thread in the web server is allowed to directly communicate with memcached servers directly, bypassing mcrouter, without establishing and maintaining a connection thereby reducing the overhead. The UDP implementation detects packets that are dropped or received out of order (using sequence numbers) and treats them as errors on the client side. It does not provide any mechanism to try to recover from them. In our infrastructure, we find this decision to be practical. Under peak load, memcache clients observe that 0.25% of get requests are discarded. About 80% of these drops are due to late or dropped packets, while the remainder are due to out of order delivery. Clients treat get errors as cache misses, but web servers will skip inserting entries into memcached after querying for data to avoid putting additional load on a possibly overloaded network or server.

[1]The 95th percentile of fetches for that page is 1,740 items.

[2]The 95th percentile is 95 keys per request.

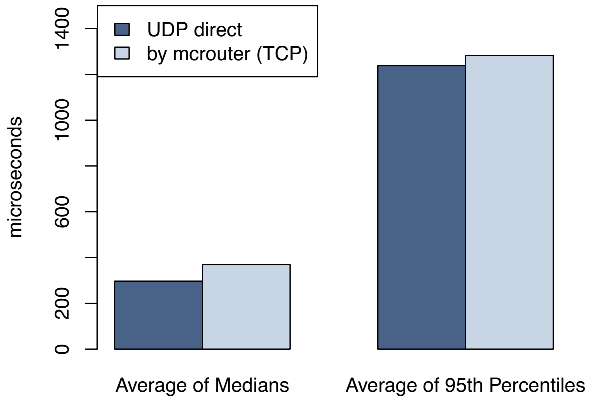

Figure 3: Get latency for UDP, TCP via mcrouter

For reliability, clients perform set and delete operations over TCP through an instance of mcrouter running on the same machine as the web server. For operations where we need to confirm a state change (updates and deletes) TCP alleviates the need to add a retry mechanism to our UDP implementation.

Web servers rely on a high degree of parallelism and over-subscription to achieve high throughput. The high memory demands of open TCP connections makes it prohibitively expensive to have an open connection between every web thread and memcached server without some form of connection coalescing via mcrouter. Coalescing these connections improves the efficiency of the server by reducing the network, CPU and memory resources needed by high throughput TCP connections. Figure 3 shows the average, median, and 95th percentile latencies of web servers in production getting keys over UDP and through mcrouter via TCP. In all cases, the standard deviation from these averages was less than 1%. As the data show, relying on UDP can lead to a 20% reduction in latency to serve requests.

Incast congestion: Memcache clients implement flow-control mechanisms to limit incast congestion. When a

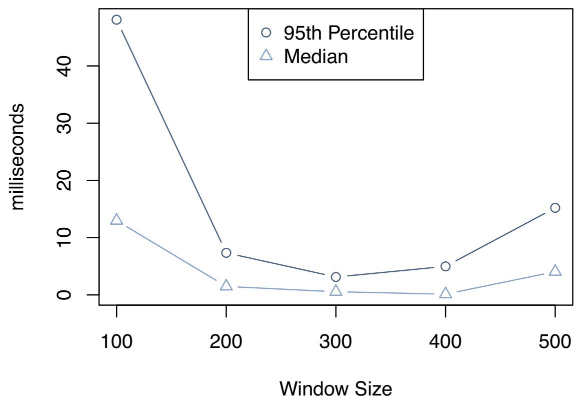

Figure 4: Average time web requests spend waiting to be scheduled

client requests a large number of keys, the responses can overwhelm components such as rack and cluster switches if those responses arrive all at once. Clients therefore use a sliding window mechanism [11] to control the number of outstanding requests. When the client receives a response, the next request can be sent. Similar to TCP's congestion control, the size of this sliding window grows slowly upon a successful request and shrinks when a request goes unanswered. The window applies to all `memcache` requests independently of destination; whereas TCP windows apply only to a single stream.

Figure 4 shows the impact of the window size on the amount of time user requests are in the runnable state but are waiting to be scheduled inside the web server. The data was gathered from multiple racks in one frontend cluster. User requests exhibit a Poisson arrival process at each web server. According to Little's Law [26], $L = \lambda W$, the number of requests queued in the server (L) is directly proportional to the average time a request takes to process (W), assuming that the input request rate is constant (which it was for our experiment). The time web requests are waiting to be scheduled is a direct indication of the number of web requests in the system. With lower window sizes, the application will have to dispatch more groups of `memcache` requests serially, increasing the duration of the web request. As the window size gets too large, the number of simultaneous `memcache` requests causes incast congestion. The result will be `memcache` errors and the application falling back to the persistent storage for the data, which will result in slower processing of web requests. There is a balance between these extremes where unnecessary latency can be avoided and incast congestion can be minimized.

3.2 Reducing Load

We use `memcache` to reduce the frequency of fetching data along more expensive paths such as database queries. Web servers fall back to these paths when the desired data is not cached. The following subsections describe three techniques for decreasing load.

3.2.1 Leases

We introduce a new mechanism we call *leases* to address two problems: stale sets and thundering herds. A stale set occurs when a web server sets a value in `memcache` that does not reflect the latest value that should be cached. This can occur when concurrent updates to `memcache` get reordered. A thundering herd happens when a specific key undergoes heavy read and write activity. As the write activity repeatedly invalidates the recently set values, many reads default to the more costly path. Our lease mechanism solves both problems.

Intuitively, a `memcached` instance gives a *lease* to a client to set data back into the cache when that client experiences a cache miss. The lease is a 64-bit token bound to the specific key the client originally requested. The client provides the lease token when setting the value in the cache. With the lease token, `memcached` can verify and determine whether the data should be stored and thus arbitrate concurrent writes. Verification can fail if `memcached` has invalidated the lease token due to receiving a delete request for that item. Leases prevent stale sets in a manner similar to how load-link/store-conditional operates [20].

A slight modification to *leases* also mitigates thundering herds. Each `memcached` server regulates the rate at which it returns tokens. By default, we configure these servers to return a token only once every 10 seconds per key. Requests for a key's value within 10 seconds of a token being issued results in a special notification telling the client to wait a short amount of time. Typically, the client with the lease will have successfully set the data within a few milliseconds. Thus, when waiting clients retry the request, the data is often present in cache.

To illustrate this point we collect data for all cache misses of a set of keys particularly susceptible to thundering herds for one week. Without leases, all of the cache misses resulted in a peak database query rate of 17K/s. With leases, the peak database query rate was 1.3K/s. Since we provision our databases based on peak load, our lease mechanism translates to a significant efficiency gain.

Stale values: With leases, we can minimize the application's wait time in certain use cases. We can further reduce this time by identifying situations in which returning slightly out-of-date data is acceptable. When a key is deleted, its value is transferred to a data struc-

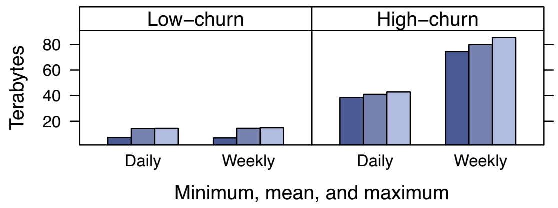

Figure 5: Daily and weekly working set of a high-churn family and a low-churn key family

ture that holds recently deleted items, where it lives for a short time before being flushed. A get request can return a lease token or data that is marked as stale. Applications that can continue to make forward progress with stale data do not need to wait for the latest value to be fetched from the databases. Our experience has shown that since the cached value tends to be a monotonically increasing snapshot of the database, most applications can use a stale value without any changes.

3.2.2 `Memcache` Pools

Using `memcache` as a general-purpose caching layer requires workloads to share infrastructure despite different access patterns, memory footprints, and quality-of-service requirements. Different applications' workloads can produce negative interference resulting in decreased hit rates.

To accommodate these differences, we partition a cluster's `memcached` servers into separate pools. We designate one pool (named *wildcard*) as the default and provision separate pools for keys whose residence in wildcard is problematic. For example, we may provision a small pool for keys that are accessed frequently but for which a cache miss is inexpensive. We may also provision a large pool for infrequently accessed keys for which cache misses are prohibitively expensive.

Figure 5 shows the working set of two different sets of items, one that is low-churn and another that is high-churn. The working set is approximated by sampling all operations on one out of every one million items. For each of these items, we collect the minimum, average, and maximum item size. These sizes are summed and multiplied by one million to approximate the working set. The difference between the daily and weekly working sets indicates the amount of churn. Items with different churn characteristics interact in an unfortunate way: low-churn keys that are still valuable are evicted before high-churn keys that are no longer being accessed. Placing these keys in different pools prevents this kind of negative interference, and allows us to size high-churn pools appropriate to their cache miss cost. Section 7 provides further analysis.

3.2.3 Replication Within Pools

Within some pools, we use replication to improve the latency and efficiency of `memcached` servers. We choose to replicate a category of keys within a pool when (1) the application routinely fetches many keys simultaneously, (2) the entire data set fits in one or two `memcached` servers and (3) the request rate is much higher than what a single server can manage.

We favor replication in this instance over further dividing the key space. Consider a `memcached` server holding 100 items and capable of responding to 500k requests per second. Each request asks for 100 keys. The difference in `memcached` overhead for retrieving 100 keys per request instead of 1 key is small. To scale the system to process 1M requests/sec, suppose that we add a second server and split the key space equally between the two. Clients now need to split each request for 100 keys into two parallel requests for $\sim$50 keys. Consequently, both servers still have to process 1M requests per second. However, if we replicate all 100 keys to multiple servers, a client's request for 100 keys can be sent to any replica. This reduces the load per server to 500k requests per second. Each client chooses replicas based on its own IP address. This approach requires delivering invalidations to all replicas to maintain consistency.

3.3 Handling Failures

The inability to fetch data from `memcache` results in excessive load to backend services that could cause further cascading failures. There are two scales at which we must address failures: (1) a small number of hosts are inaccessible due to a network or server failure or (2) a widespread outage that affects a significant percentage of the servers within the cluster. If an entire cluster has to be taken offline, we divert user web requests to other clusters which effectively removes all the load from `memcache` within that cluster.

For small outages we rely on an automated remediation system [3]. These actions are not instant and can take up to a few minutes. This duration is long enough to cause the aforementioned cascading failures and thus we introduce a mechanism to further insulate backend services from failures. We dedicate a small set of machines, named *Gutter*, to take over the responsibilities of a few failed servers. Gutter accounts for approximately 1% of the `memcached` servers in a cluster.

When a `memcached` client receives no response to its get request, the client assumes the server has failed and issues the request again to a special *Gutter* pool. If this second request misses, the client will insert the appropriate key-value pair into the Gutter machine after querying the database. Entries in Gutter expire quickly to obviate

Gutter invalidations. Gutter limits the load on backend services at the cost of slightly stale data.

Note that this design differs from an approach in which a client rehashes keys among the remaining `memcached` servers. Such an approach risks cascading failures due to non-uniform key access frequency. For example, a single key can account for 20% of a server's requests. The server that becomes responsible for this hot key might also become overloaded. By shunting load to idle servers we limit that risk.

Ordinarily, each failed request results in a hit on the backing store, potentially overloading it. By using Gutter to store these results, a substantial fraction of these failures are converted into hits in the gutter pool thereby reducing load on the backing store. In practice, this system reduces the rate of client-visible failures by 99% and converts 10%–25% of failures into hits each day. If a `memcached` server fails entirely, hit rates in the gutter pool generally exceed 35% in under 4 minutes and often approach 50%. Thus when a few `memcached` servers are unavailable due to failure or minor network incidents, Gutter protects the backing store from a surge of traffic.

4 In a Region: Replication

It is tempting to buy more web and `memcached` servers to scale a cluster as demand increases. However, naïvely scaling the system does not eliminate all problems. Highly requested items will only become more popular as more web servers are added to cope with increased user traffic. Incast congestion also worsens as the number of `memcached` servers increases. We therefore split our web and `memcached` servers into multiple *frontend clusters*. These clusters, along with a storage cluster that contain the databases, define a *region*. This region architecture also allows for smaller failure domains and a tractable network configuration. We trade replication of data for more independent failure domains, tractable network configuration, and a reduction of incast congestion.

This section analyzes the impact of multiple frontend clusters that share the same storage cluster. Specifically we address the consequences of allowing data replication across these clusters and the potential memory efficiencies of disallowing this replication.

4.1 Regional Invalidations

While the storage cluster in a region holds the authoritative copy of data, user demand may replicate that data into frontend clusters. The storage cluster is responsible for invalidating cached data to keep frontend clusters consistent with the authoritative versions. As an optimization, a web server that modifies data also sends invalidations to its own cluster to provide read-after-write semantics for a single user request and reduce the amount of time stale data is present in its local cache.

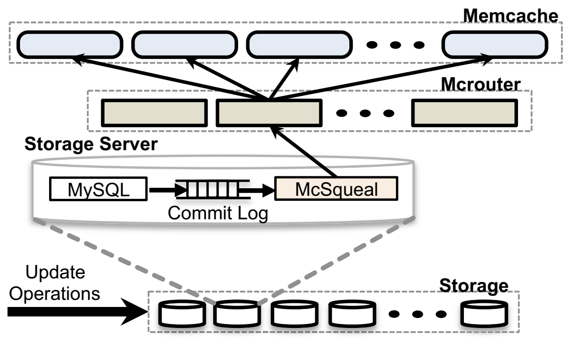

Figure 6: Invalidation pipeline showing keys that need to be deleted via the daemon (`mcsqueal`).

SQL statements that modify authoritative state are amended to include `memcache` keys that need to be invalidated once the transaction commits [7]. We deploy invalidation daemons (named `mcsqueal`) on every database. Each daemon inspects the SQL statements that its database commits, extracts any deletes, and broadcasts these deletes to the `memcache` deployment in every frontend cluster in that region. Figure 6 illustrates this approach. We recognize that most invalidations do not delete data; indeed, only 4% of all deletes issued result in the actual invalidation of cached data.

Reducing packet rates: While `mcsqueal` could contact `memcached` servers directly, the resulting rate of packets sent from a backend cluster to frontend clusters would be unacceptably high. This packet rate problem is a consequence of having many databases and many `memcached` servers communicating across a cluster boundary. Invalidation daemons batch deletes into fewer packets and send them to a set of dedicated servers running `mcrouter` instances in each frontend cluster. These `mcrouter`s then unpack individual deletes from each batch and route those invalidations to the right `memcached` server co-located within the frontend cluster. The batching results in an 18× improvement in the median number of deletes per packet.

Invalidation via web servers: It is simpler for web servers to broadcast invalidations to all frontend clusters. This approach unfortunately suffers from two problems. First, it incurs more packet overhead as web servers are less effective at batching invalidations than `mcsqueal` pipeline. Second, it provides little recourse when a systemic invalidation problem arises such as misrouting of deletes due to a configuration error. In the past, this would often require a rolling restart of the entire `memcache` infrastructure, a slow and disruptive pro-

	A (Cluster)	B (Region)
Median number of users	30	1
Gets per second	3.26 M	458 K
Median value size	10.7 kB	4.34 kB

Table 1: Deciding factors for cluster or regional replication of two item families

cess we want to avoid. In contrast, embedding invalidations in SQL statements, which databases commit and store in reliable logs, allows `mcsqueal` to simply replay invalidations that may have been lost or misrouted.

4.2 Regional Pools

Each cluster independently caches data depending on the mix of the user requests that are sent to it. If users' requests are randomly routed to all available frontend clusters then the cached data will be roughly the same across all the frontend clusters. This allows us to take a cluster offline for maintenance without suffering from reduced hit rates. Over-replicating the data can be memory inefficient, especially for large, rarely accessed items. We can reduce the number of replicas by having multiple frontend clusters share the same set of `memcached` servers. We call this a *regional pool*.

Crossing cluster boundaries incurs more latency. In addition, our networks have 40% less average available bandwidth over cluster boundaries than within a single cluster. Replication trades more `memcached` servers for less inter-cluster bandwidth, lower latency, and better fault tolerance. For some data, it is more cost efficient to forgo the advantages of replicating data and have a single copy per region. One of the main challenges of scaling `memcache` within a region is deciding whether a key needs to be replicated across all frontend clusters or have a single replica per region. Gutter is also used when servers in regional pools fail.

Table 1 summarizes two kinds of items in our application that have large values. We have moved one kind (B) to a regional pool while leaving the other (A) untouched. Notice that clients access items falling into category B an order of magnitude less than those in category A. Category B's low access rate makes it a prime candidate for a regional pool since it does not adversely impact inter-cluster bandwidth. Category B would also occupy 25% of each cluster's wildcard pool so regionalization provides significant storage efficiencies. Items in category A, however, are twice as large and accessed much more frequently, disqualifying themselves from regional consideration. The decision to migrate data into regional pools is currently based on a set of manual heuristics based on access rates, data set size, and number of unique users accessing particular items.

4.3 Cold Cluster Warmup

When we bring a new cluster online, an existing one fails, or perform scheduled maintenance the caches will have very poor hit rates diminishing the ability to insulate backend services. A system called *Cold Cluster Warmup* mitigates this by allowing clients in the "cold cluster" (*i.e.* the frontend cluster that has an empty cache) to retrieve data from the "warm cluster" (*i.e.* a cluster that has caches with normal hit rates) rather than the persistent storage. This takes advantage of the aforementioned data replication that happens across frontend clusters. With this system cold clusters can be brought back to full capacity in a few hours instead of a few days.

Care must be taken to avoid inconsistencies due to race conditions. For example, if a client in the cold cluster does a database update, and a subsequent request from another client retrieves the stale value from the warm cluster before the warm cluster has received the invalidation, that item will be indefinitely inconsistent in the cold cluster. `Memcached` deletes support nonzero hold-off times that reject `add` operations for the specified hold-off time. By default, all deletes to the cold cluster are issued with a two second hold-off. When a miss is detected in the cold cluster, the client re-requests the key from the warm cluster and `adds` it into the cold cluster. The failure of the `add` indicates that newer data is available on the database and thus the client will refetch the value from the databases. While there is still a theoretical possibility that deletes get delayed more than two seconds, this is not true for the vast majority of the cases. The operational benefits of cold cluster warmup far outweigh the cost of rare cache consistency issues. We turn it off once the cold cluster's hit rate stabilizes and the benefits diminish.

5 Across Regions: Consistency

There are several advantages to a broader geographic placement of data centers. First, putting web servers closer to end users can significantly reduce latency. Second, geographic diversity can mitigate the effects of events such as natural disasters or massive power failures. And third, new locations can provide cheaper power and other economic incentives. We obtain these advantages by deploying to multiple regions. Each region consists of a storage cluster and several frontend clusters. We designate one region to hold the master databases and the other regions to contain read-only replicas; we rely on MySQL's replication mechanism to keep replica databases up-to-date with their masters. In this design, web servers experience low latency when accessing either the local `memcached` servers or the local database replicas. When scaling across multiple regions, maintaining consistency between data in `memcache` and the persistent storage becomes the primary technical challenge. These challenges stem from a single problem: replica databases may lag behind the master database.

Our system represents just one point in the wide spectrum of consistency and performance trade-offs. The consistency model, like the rest of the system, has evolved over the years to suit the scale of the site. It mixes what can be practically built without sacrificing our high performance requirements. The large volume of data that the system manages implies that any minor changes that increase network or storage requirements have non-trivial costs associated with them. Most ideas that provide stricter semantics rarely leave the design phase because they become prohibitively expensive. Unlike many systems that are tailored to an existing use case, `memcache` and Facebook were developed together. This allowed the applications and systems engineers to work together to find a model that is sufficiently easy for the application engineers to understand yet performant and simple enough for it to work reliably at scale. We provide best-effort eventual consistency but place an emphasis on performance and availability. Thus the system works very well for us in practice and we think we have found an acceptable trade-off.

Writes from a master region: Our earlier decision requiring the storage cluster to invalidate data via daemons has important consequences in a multi-region architecture. In particular, it avoids a race condition in which an invalidation arrives *before* the data has been replicated from the master region. Consider a web server in the master region that has finished modifying a database and seeks to invalidate now stale data. Sending invalidations within the master region is safe. However, having the web server invalidate data in a replica region may be premature as the changes may not have been propagated to the replica databases yet. Subsequent queries for the data from the replica region will race with the replication stream thereby increasing the probability of setting stale data into `memcache`. Historically, we implemented `mcsqueal` after scaling to multiple regions.

Writes from a non-master region: Now consider a user who updates his data from a non-master region when replication lag is excessively large. The user's next request could result in confusion if his recent change is missing. A cache refill from a replica's database should only be allowed after the replication stream has caught up. Without this, subsequent requests could result in the replica's stale data being fetched and cached.

We employ a *remote marker* mechanism to minimize the probability of reading stale data. The presence of the marker indicates that data in the local replica database are potentially stale and the query should be redirected to the master region. When a web server wishes to update data that affects a key k, that server (1) sets a remote marker r_k in the region, (2) performs the write to the master embedding k and r_k to be invalidated in the SQL statement, and (3) deletes k in the local cluster. On a subsequent request for k, a web server will be unable to find the cached data, check whether r_k exists, and direct its query to the master or local region depending on the presence of r_k. In this situation, we explicitly trade additional latency when there is a cache miss, for a decreased probability of reading stale data.

We implement remote markers by using a regional pool. Note that this mechanism may reveal stale information during concurrent modifications to the same key as one operation may delete a remote marker that should remain present for another in-flight operation. It is worth highlighting that our usage of `memcache` for remote markers departs in a subtle way from caching results. As a cache, deleting or evicting keys is always a safe action; it may induce more load on databases, but does not impair consistency. In contrast, the presence of a remote marker helps distinguish whether a non-master database holds stale data or not. In practice, we find both the eviction of remote markers and situations of concurrent modification to be rare.

Operational considerations: Inter-region communication is expensive since data has to traverse large geographical distances (*e.g.* across the continental United States). By sharing the same channel of communication for the delete stream as the database replication we gain network efficiency on lower bandwidth connections.

The aforementioned system for managing deletes in Section 4.1 is also deployed with the replica databases to broadcast the deletes to `memcached` servers in the replica regions. Databases and `mcrouter`s buffer deletes when downstream components become unresponsive. A failure or delay in any of the components results in an increased probability of reading stale data. The buffered deletes are replayed once these downstream components are available again. The alternatives involve taking a cluster offline or over-invalidating data in frontend clusters when a problem is detected. These approaches result in more disruptions than benefits given our workload.

6 Single Server Improvements

The *all-to-all* communication pattern implies that a single server can become a bottleneck for a cluster. This section describes performance optimizations and memory efficiency gains in `memcached` which allow better scaling within clusters. Improving single server cache performance is an active research area [9, 10, 28, 25].

6.1 Performance Optimizations

We began with a single-threaded `memcached` which used a fixed-size hash table. The first major optimizations were to: (1) allow automatic expansion of the hash table to avoid look-up times drifting to $O(n)$, (2) make the server multi-threaded using a global lock to protect multiple data structures, and (3) giving each thread its own UDP port to reduce contention when sending replies and later spreading interrupt processing overhead. The first two optimizations were contributed back to the open source community. The remainder of this section explores further optimizations that are not yet available in the open source version.

Our experimental hosts have an Intel Xeon CPU (X5650) running at 2.67GHz (12 cores and 12 hyperthreads), an Intel 82574L gigabit ethernet controller and 12GB of memory. Production servers have additional memory. Further details have been previously published [4]. The performance test setup consists of fifteen clients generating `memcache` traffic to a single `memcached` server with 24 threads. The clients and server are co-located on the same rack and connected through gigabit ethernet. These tests measure the latency of `memcached` responses over two minutes of sustained load.

Get Performance: We first investigate the effect of replacing our original multi-threaded single-lock implementation with fine-grained locking. We measured hits by pre-populating the cache with 32-byte values before issuing `memcached` requests of 10 keys each. Figure 7 shows the maximum request rates that can be sustained with sub-millisecond average response times for different versions of `memcached`. The first set of bars is our `memcached` before fine-grained locking, the second set is our current `memcached`, and the final set is the open source version 1.4.10 which independently implements a coarser version of our locking strategy.

Employing fine-grained locking triples the peak get rate for hits from 600k to 1.8M items per second. Performance for misses also increased from 2.7M to 4.5M items per second. Hits are more expensive because the return value has to be constructed and transmitted, while misses require a single static response (`END`) for the entire multiget indicating that all keys missed.

We also investigated the performance effects of using UDP instead of TCP. Figure 8 shows the peak request rate we can sustain with average latencies of less than one millisecond for single gets and multigets of 10 keys. We found that our UDP implementation outperforms our TCP implementation by 13% for single gets and 8% for 10-key multigets.

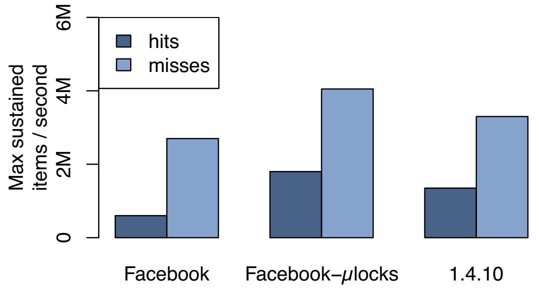

Figure 7: Multiget hit and miss performance comparison by `memcached` version

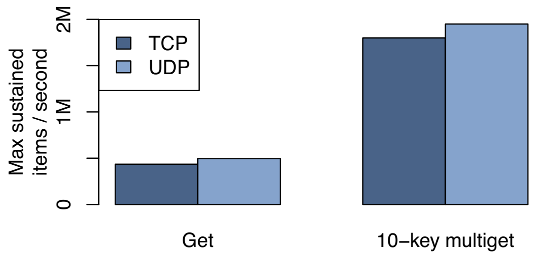

Figure 8: Get hit performance comparison for single gets and 10-key multigets over TCP and UDP

Because multigets pack more data into each request than single gets, they use fewer packets to do the same work. Figure 8 shows an approximately four-fold improvement for 10-key multigets over single gets.

6.2 Adaptive Slab Allocator

`Memcached` employs a slab allocator to manage memory. The allocator organizes memory into *slab classes*, each of which contains pre-allocated, uniformly sized chunks of memory. `Memcached` stores items in the smallest possible slab class that can fit the item's metadata, key, and value. Slab classes start at 64 bytes and exponentially increase in size by a factor of 1.07 up to 1 MB, aligned on 4-byte boundaries[3]. Each slab class maintains a free-list of available chunks and requests more memory in 1MB slabs when its free-list is empty. Once a `memcached` server can no longer allocate free memory, storage for new items is done by evicting the least recently used (LRU) item within that slab class. When workloads change, the original memory allocated to each slab class may no longer be enough resulting in poor hit rates.

[3]This scaling factor ensures that we have both 64 and 128 byte items which are more amenable to hardware cache lines.

We implemented an adaptive allocator that periodically re-balances slab assignments to match the current workload. It identifies slab classes as needing more memory if they are currently evicting items and if the next item to be evicted was used at least 20% more recently than the average of the least recently used items in other slab classes. If such a class is found, then the slab holding the least recently used item is freed and transferred to the needy class. Note that the open-source community has independently implemented a similar allocator that balances the eviction rates across slab classes while our algorithm focuses on balancing the age of the oldest items among classes. Balancing age provides a better approximation to a single global Least Recently Used (LRU) eviction policy for the entire server rather than adjusting eviction rates which can be heavily influenced by access patterns.

6.3 The Transient Item Cache

While `memcached` supports expiration times, entries may live in memory well after they have expired. `Memcached` lazily evicts such entries by checking expiration times when serving a get request for that item or when they reach the end of the LRU. Although efficient for the common case, this scheme allows short-lived keys that see a single burst of activity to waste memory until they reach the end of the LRU.

We therefore introduce a hybrid scheme that relies on lazy eviction for most keys and proactively evicts short-lived keys when they expire. We place short-lived items into a circular buffer of linked lists (indexed by seconds until expiration) – called the *Transient Item Cache* – based on the expiration time of the item. Every second, all of the items in the bucket at the head of the buffer are evicted and the head advances by one. When we added a short expiration time to a heavily used set of keys whose items have short useful lifespans; the proportion of `memcache` pool used by this key family was reduced from 6% to 0.3% without affecting the hit rate.

6.4 Software Upgrades

Frequent software changes may be needed for upgrades, bug fixes, temporary diagnostics, or performance testing. A `memcached` server can reach 90% of its peak hit rate within a few hours. Consequently, it can take us over 12 hours to upgrade a set of `memcached` servers as the resulting database load needs to be managed carefully. We modified `memcached` to store its cached values and main data structures in System V shared memory regions so that the data can remain live across a software upgrade and thereby minimize disruption.

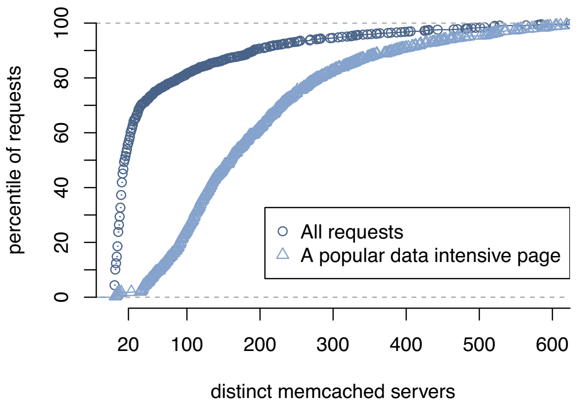

Figure 9: Cumulative distribution of the number of distinct `memcached` servers accessed

7 `Memcache` Workload

We now characterize the `memcache` workload using data from servers that are running in production.

7.1 Measurements at the Web Server

We record all `memcache` operations for a small percentage of user requests and discuss the fan-out, response size, and latency characteristics of our workload.

Fanout: Figure 9 shows the distribution of distinct `memcached` servers a web server may need to contact when responding to a page request. As shown, 56% of all page requests contact fewer than 20 `memcached` servers. By volume, user requests tend to ask for small amounts of cached data. There is, however, a long tail to this distribution. The figure also depicts the distribution for one of our more popular pages that better exhibits the all-to-all communication pattern. Most requests of this type will access over 100 distinct servers; accessing several hundred `memcached` servers is not rare.

Response size: Figure 10 shows the response sizes from `memcache` requests. The difference between the median (135 bytes) and the mean (954 bytes) implies that there is a very large variation in the sizes of the cached items. In addition there appear to be three distinct peaks at approximately 200 bytes and 600 bytes. Larger items tend to store lists of data while smaller items tend to store single pieces of content.

Latency: We measure the round-trip latency to request data from `memcache`, which includes the cost of routing the request and receiving the reply, network transfer time, and the cost of deserialization and decompression. Over 7 days the median request latency is 333 microseconds while the 75^{th} and 95^{th} percentiles (p75 and p95) are 475μs and 1.135ms respectively. Our median end-to-end latency from an idle web server is 178μs while the p75 and p95 are 219μs and 374μs, respectively. The

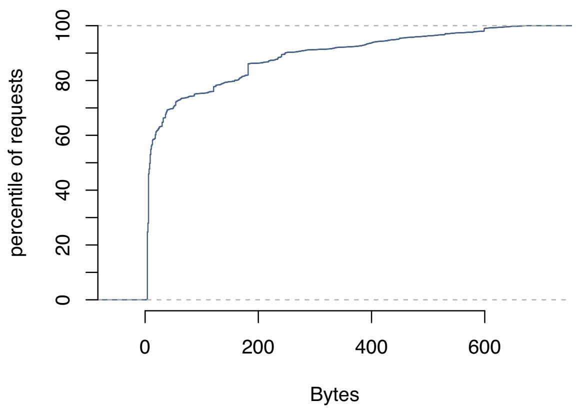

Figure 10: Cumulative distribution of value sizes fetched

wide variance between the p95 latencies arises from handling large responses and waiting for the runnable thread to be scheduled as discussed in Section 3.1.

7.2 Pool Statistics

We now discuss key metrics of four `memcache` pools. The pools are wildcard (the default pool), app (a pool devoted for a specific application), a replicated pool for frequently accessed data, and a regional pool for rarely accessed information. In each pool, we collect average statistics every 4 minutes and report in Table 2 the highest average for one month collection period. This data approximates the peak load seen by those pools. The table shows the widely different get, set, and delete rates for different pools. Table 3 shows the distribution of response sizes for each pool. Again, the different characteristics motivate our desire to segregate these workloads from one another.

As discussed in Section 3.2.3, we replicate data within a pool and take advantage of batching to handle the high request rates. Observe that the replicated pool has the highest get rate (about 2.7× that of the next highest one) and the highest ratio of bytes to packets despite having the smallest item sizes. This data is consistent with our design in which we leverage replication and batching to achieve better performance. In the *app* pool, a higher churn of data results in a naturally higher miss rate. This pool tends to have content that is accessed for a few hours and then fades away in popularity in favor of newer content. Data in the regional pool tends to be large and infrequently accessed as shown by the request rates and the value size distribution.

7.3 Invalidation Latency

We recognize that the timeliness of invalidations is a critical factor in determining the probability of exposing stale data. To monitor this health, we sample one out of a million deletes and record the time the delete was issued. We subsequently query the contents of `memcache` across all frontend clusters at regular intervals for the sampled keys and log an error if an item remains cached despite a delete that should have invalidated it.

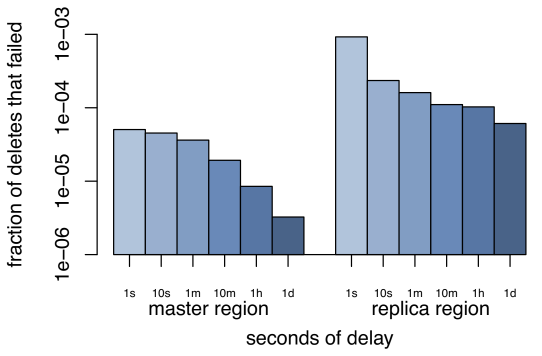

Figure 11: Latency of the Delete Pipeline

In Figure 11, we use this monitoring mechanism to report our invalidation latencies across a 30 day span. We break this data into two different components: (1) the delete originated from a web server in the master region and was destined to a `memcached` server in the master region and (2) the delete originated from a replica region and was destined to another replica region. As the data show, when the source and destination of the delete are co-located with the master our success rates are much higher and achieve four 9s of reliability within 1 second and five 9s after one hour. However when the deletes originate and head to locations outside of the master region our reliability drops to three 9s within a second and four 9s within 10 minutes. In our experience, we find that if an invalidation is missing after only a few seconds the most common reason is that the first attempt failed and subsequent retrials will resolve the problem.

8 Related Work

Several other large websites have recognized the utility of key-value stores. DeCandia *et al.* [12] present a highly available key-value store that is used by a variety of application services at Amazon.com. While their system is optimized for a write heavy workload, ours targets a workload dominated by reads. Similarly, LinkedIn uses Voldemort [5], a system inspired by Dynamo. Other major deployments of key-value caching solutions include Redis [6] at Github, Digg, and Blizzard, and `memcached` at Twitter [33] and Zynga. Lakshman *et al.* [1] developed Cassandra, a schema-based distributed key-value store. We preferred to deploy and scale `memcached` due to its simpler design.

Our work in scaling `memcache` builds on extensive work in distributed data structures. Gribble *et al.* [19]

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

present an early version of a key-value storage system useful for Internet scale services. Ousterhout *et al.* [29] also present the case for a large scale in-memory key-value storage system. Unlike both of these solutions, memcache does not guarantee persistence. We rely on other systems to handle persistent data storage.

Ports *et al.* [31] provide a library to manage the cached results of queries to a transactional database. Our needs require a more flexible caching strategy. Our use of leases [18] and stale reads [23] leverages prior research on cache consistency and read operations in high-performance systems. Work by Ghandeharizadeh and Yap [15] also presents an algorithm that addresses the stale set problem based on time-stamps rather than explicit version numbers.

While software routers are easier to customize and program, they are often less performant than their hardware counterparts. Dobrescu *et al.* [13] address these issues by taking advantage of multiple cores, multiple memory controllers, multi-queue networking interfaces, and batch processing on general purpose servers. Applying these techniques to mcrouter's implementation remains future work. Twitter has also independently developed a memcache proxy similar to mcrouter [32].

In Coda [35], Satyanarayanan *et al.* demonstrate how datasets that diverge due to disconnected operation can be brought back into sync. Glendenning *et al.* [17] leverage Paxos [24] and quorums [16] to build Scatter, a distributed hash table with linearizable semantics [21] resilient to churn. Lloyd *et al.* [27] examine causal consistency in COPS, a wide-area storage system.

TAO [37] is another Facebook system that relies heavily on caching to serve large numbers of low-latency queries. TAO differs from memcache in two fundamental ways. (1) TAO implements a graph data model in which nodes are identified by fixed-length persistent identifiers (64-bit integers). (2) TAO encodes a specific mapping of its graph model to persistent storage and takes responsibility for persistence. Many components, such as our client libraries and mcrouter, are used by both systems.

9 Conclusion

In this paper, we show how to scale a memcached-based architecture to meet the growing demand of Facebook. Many of the trade-offs discussed are not fundamental, but are rooted in the realities of balancing engineering resources while evolving a live system under continuous product development. While building, maintaining, and evolving our system we have learned the following lessons. (1) Separating cache and persistent storage systems allows us to independently scale them. (2) Features that improve monitoring, debugging and operational efficiency are as important as performance. (3) Managing stateful components is operationally more complex than stateless ones. As a result keeping logic in a stateless client helps iterate on features and minimize disruption. (4) The system must support gradual rollout and rollback of new features even if it leads to temporary heterogeneity of feature sets. (5) Simplicity is vital.

Acknowledgements

We would like to thank Philippe Ajoux, Nathan Bronson, Mark Drayton, David Fetterman, Alex Gartrell, Andrii Grynenko, Robert Johnson, Sanjeev Kumar, Anton Likhtarov, Mark Marchukov, Scott Marlette, Ben Maurer, David Meisner, Konrad Michels, Andrew Pope, Jeff Rothschild, Jason Sobel, and Yee Jiun Song for their contributions. We would also like to thank the anonymous reviewers, our shepherd Michael Piatek, Tor M. Aamodt, Remzi H. Arpaci-Dusseau, and Tayler Hetherington for their valuable feedback on earlier drafts of the paper. Finally we would like to thank our fellow engineers at Facebook for their suggestions, bug-reports, and support which makes memcache what it is today.

References

[1] Apache Cassandra. http://cassandra.apache.org/.

[2] Couchbase. http://www.couchbase.com/.

[3] Making Facebook Self-Healing. https://www.facebook.com/note.php?note_id=10150275248698920.

[4] Open Compute Project. http://www.opencompute.org.

[5] Project Voldemort. http://project-voldemort.com/.

[6] Redis. http://redis.io/.

[7] Scaling Out. https://www.facebook.com/note.php?note_id=23844338919.

[8] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. *ACM SIGMETRICS Performance Evaluation Review 40*, 1 (June 2012), 53–64.

[9] BEREZECKI, M., FRACHTENBERG, E., PALECZNY, M., AND STEELE, K. Power and performance evaluation of memcached on the tilepro64 architecture. *Sustainable Computing: Informatics and Systems 2*, 2 (June 2012), 81 – 90.

[10] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design & Implementation* (2010), pp. 1–8.

[11] CERF, V. G., AND KAHN, R. E. A protocol for packet network intercommunication. *ACM SIGCOMM Compututer Communication Review 35*, 2 (Apr. 2005), 71–82.

[12] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review 41*, 6 (Dec. 2007), 205–220.

[13] FALL, K., IANNACCONE, G., MANESH, M., RATNASAMY, S., ARGYRAKI, K., DOBRESCU, M., AND EGI, N. Routebricks: enabling general purpose network infrastructure. *ACM SIGOPS Operating Systems Review 45*, 1 (Feb. 2011), 112–125.

[14] FITZPATRICK, B. Distributed caching with memcached. *Linux Journal 2004*, 124 (Aug. 2004), 5.

[15] GHANDEHARIZADEH, S., AND YAP, J. Gumball: a race condition prevention technique for cache augmented sql database management systems. In *Proceedings of the 2nd ACM SIGMOD Workshop on Databases and Social Networks* (2012), pp. 1–6.

[16] GIFFORD, D. K. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles* (1979), pp. 150–162.

[17] GLENDENNING, L., BESCHASTNIKH, I., KRISHNAMURTHY, A., AND ANDERSON, T. Scalable consistency in Scatter. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), pp. 15–28.

[18] GRAY, C., AND CHERITON, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review 23*, 5 (Nov. 1989), 202–210.

[19] GRIBBLE, S. D., BREWER, E. A., HELLERSTEIN, J. M., AND CULLER, D. Scalable, distributed data structures for internet service construction. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design & Implementation* (2000), pp. 319–332.

[20] HEINRICH, J. *MIPS R4000 Microprocessor User's Manual*. MIPS technologies, 1994.

[21] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12*, 3 (July 1990), 463–492.

[22] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. Consistent Hashing and Random trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the 29th annual ACM Symposium on Theory of Computing* (1997), pp. 654–663.

[23] KEETON, K., MORREY, III, C. B., SOULES, C. A., AND VEITCH, A. Lazybase: freshness vs. performance in information management. *ACM SIGOPS Operating Systems Review 44*, 1 (Dec. 2010), 15–19.

[24] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.

[25] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. Silt: a memory-efficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), pp. 1–13.

[26] LITTLE, J., AND GRAVES, S. Little's law. *Building Intuition* (2008), 81–100.

[27] LLOYD, W., FREEDMAN, M., KAMINSKY, M., AND ANDERSEN, D. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), pp. 401–416.

[28] METREVELI, Z., ZELDOVICH, N., AND KAASHOEK, M. Cphash: A cache-partitioned hash table. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (2012), pp. 319–320.

[29] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., ONGARO, D., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for ramcloud. *Communications of the ACM 54*, 7 (July 2011), 121–130.

[30] PHANISHAYEE, A., KREVAT, E., VASUDEVAN, V., ANDERSEN, D. G., GANGER, G. R., GIBSON, G. A., AND SESHAN, S. Measurement and analysis of tcp throughput collapse in cluster-based storage systems. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008), pp. 12:1–12:14.

[31] PORTS, D. R. K., CLEMENTS, A. T., ZHANG, I., MADDEN, S., AND LISKOV, B. Transactional consistency and automatic management in an application data cache. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design & Implementation* (2010), pp. 1–15.

[32] RAJASHEKHAR, M. Twemproxy: A fast, light-weight proxy for memcached. https://dev.twitter.com/blog/twemproxy.

[33] RAJASHEKHAR, M., AND YUE, Y. Caching with twemcache. http://engineering.twitter.com/2012/07/caching-with-twemcache.html.

[34] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. *ACM SIGCOMM Computer Communication Review 31*, 4 (Oct. 2001), 161–172.

[35] SATYANARAYANAN, M., KISTLER, J., KUMAR, P., OKASAKI, M., SIEGEL, E., AND STEERE, D. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers 39*, 4 (Apr. 1990), 447–459.

[36] Stoica, I., Morris, R., Karger, D., Kaashoek, M., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review 31*, 4 (Oct. 2001), 149–160.

[37] Venkataramani, V., Amsden, Z., Bronson, N., Cabrera III, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Hoon, J., Kulkarni, S., Lawrence, N., Marchukov, M., Petrov, D., and Puzar, L. Tao: how facebook serves the social graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2012), pp. 791–792.