Cluster Level of Memcache

Let's identify and solve the cluster-level inefficiencies on a large key-value store implementation.

We'll cover the following

Introduction to the cluster level

At the single server level, we didn't have to worry about routing or replication. Once we start to deal with thousands of servers, we need to understand the problems that arise with them. The Memcache server clusters' key load is managed by consistent hashing, but there are still challenges to tackle. Network congestion, too many repeating cache misses, dynamic workloads, and cluster failures are all problems that we face at the cluster level and not at the single server level.

Clusters are manageable units of a data center. The number of nodes inside a cluster is configurable. Nodes inside a cluster can communicate with each other with low latency and high throughput (because they are often near each other). After scaling our key-value store on single nodes, the next level is to utilize multiple key-value stores in a cluster.

Overview of design problems at the cluster level

At the cluster level, we attempt to solve a read-heavy workload and a wide fan-out. This wide fan-out occurs because a single web request will get routed to a few clusters and then will further trigger multiple Memcached requests.

- **Network congestion:** Why do Memcache clusters face network congestion in the first place? We can explain this by giving the example of loading a user feed of posts. One web request can trigger tens, if not hundreds, of Memcached requests that are used to construct the feed.
- **Too many repeating cache misses:** If we can't respond to a request quickly enough, the front-end servers consider it as a cache miss and the data will be fetched from a more costly path. So, we need mechanisms that reduce the rate of cache misses.
- **Diverse application needs:** Different applications have different requirements for their caches, so there might be some set of key-value items which are very expensive to compute again, like all the birthdays of a user's friends. On the other hand, another set of key-value items, such as viral images that are recommended at random, needs to be replicated quickly, and it's okay if some of the servers miss it as users aren't following the page/person that shared it.
- **Cluster failures:** Machine failure is inevitable in any distributed system. So what do we do when a request comes in and is routed to a cluster that has failed? There is a slight chance that the load might cause other clusters to fail, resulting in "cascading failures."

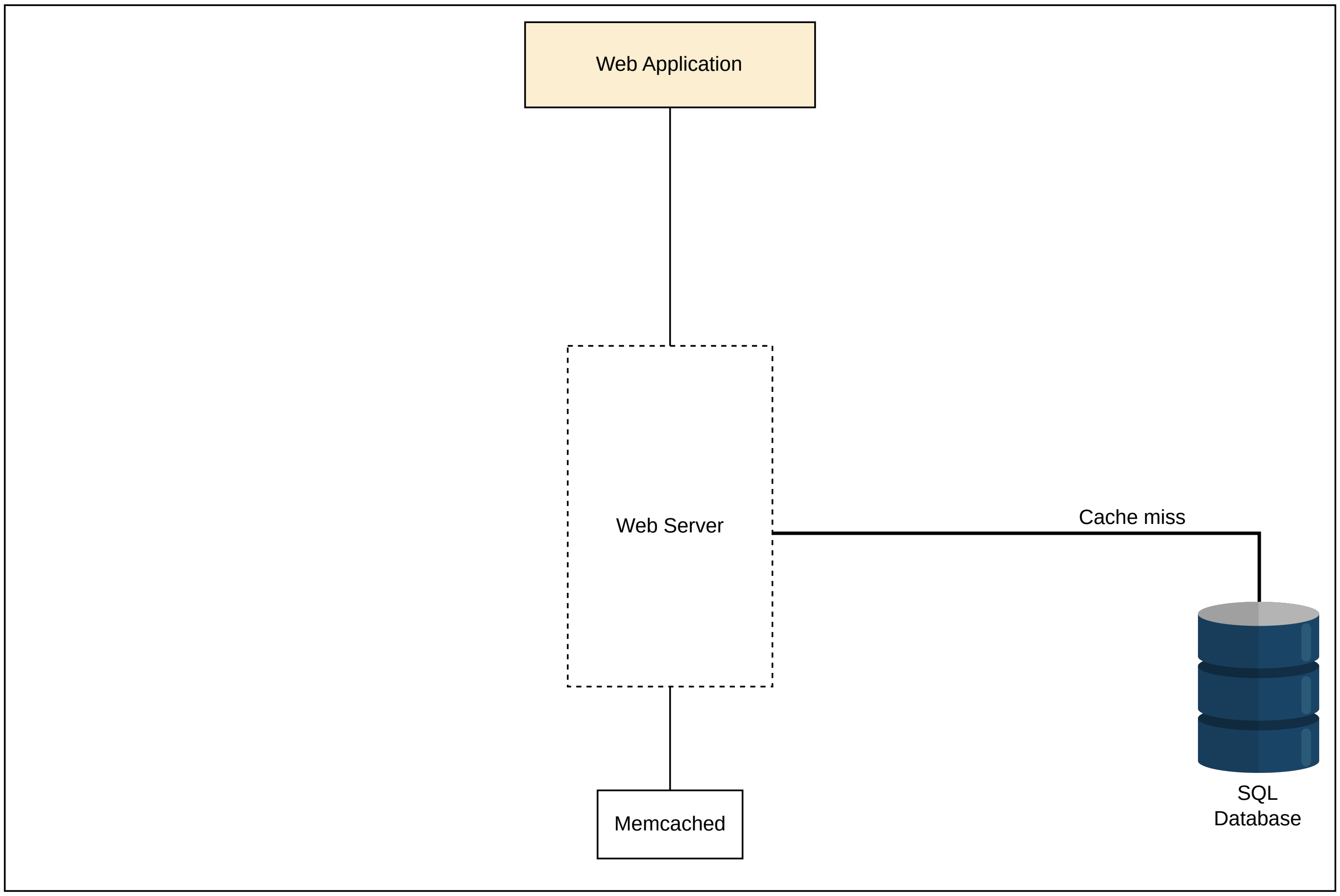
The overall design of Memcache

The individual layers of our cluster-level design are as follows:

Stateless client layer: A stateless client to the Memcached is responsible for optimal retrieval of items from Memcached servers.

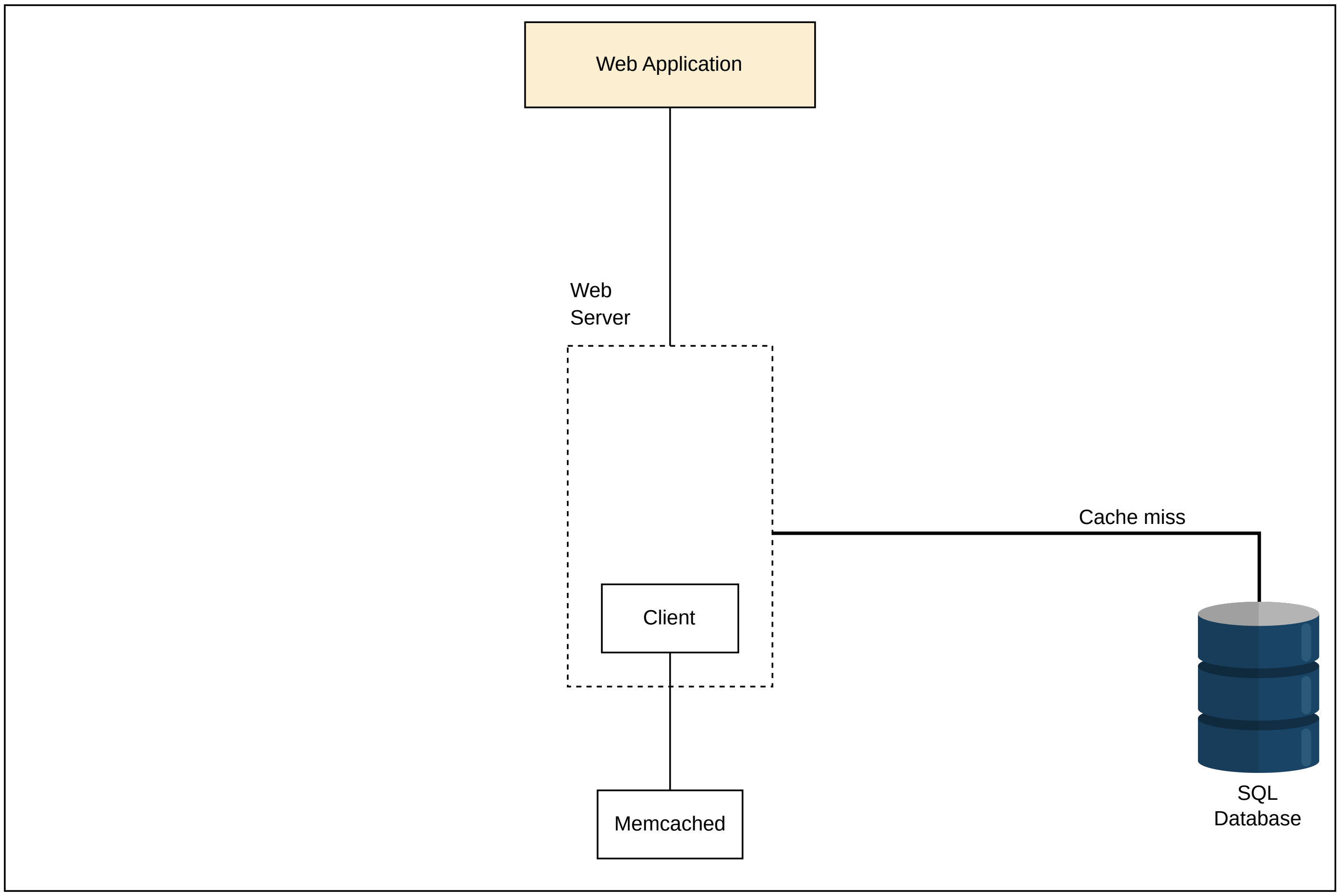
Mcrouter layer: Mcrouters are used to route requests to multiple Memcached servers using consistent hashing. To the client layer, the Mcrouter layer is the same as the Memcached layer.

Memcached server layer: Stores and serves the actual key-value items.



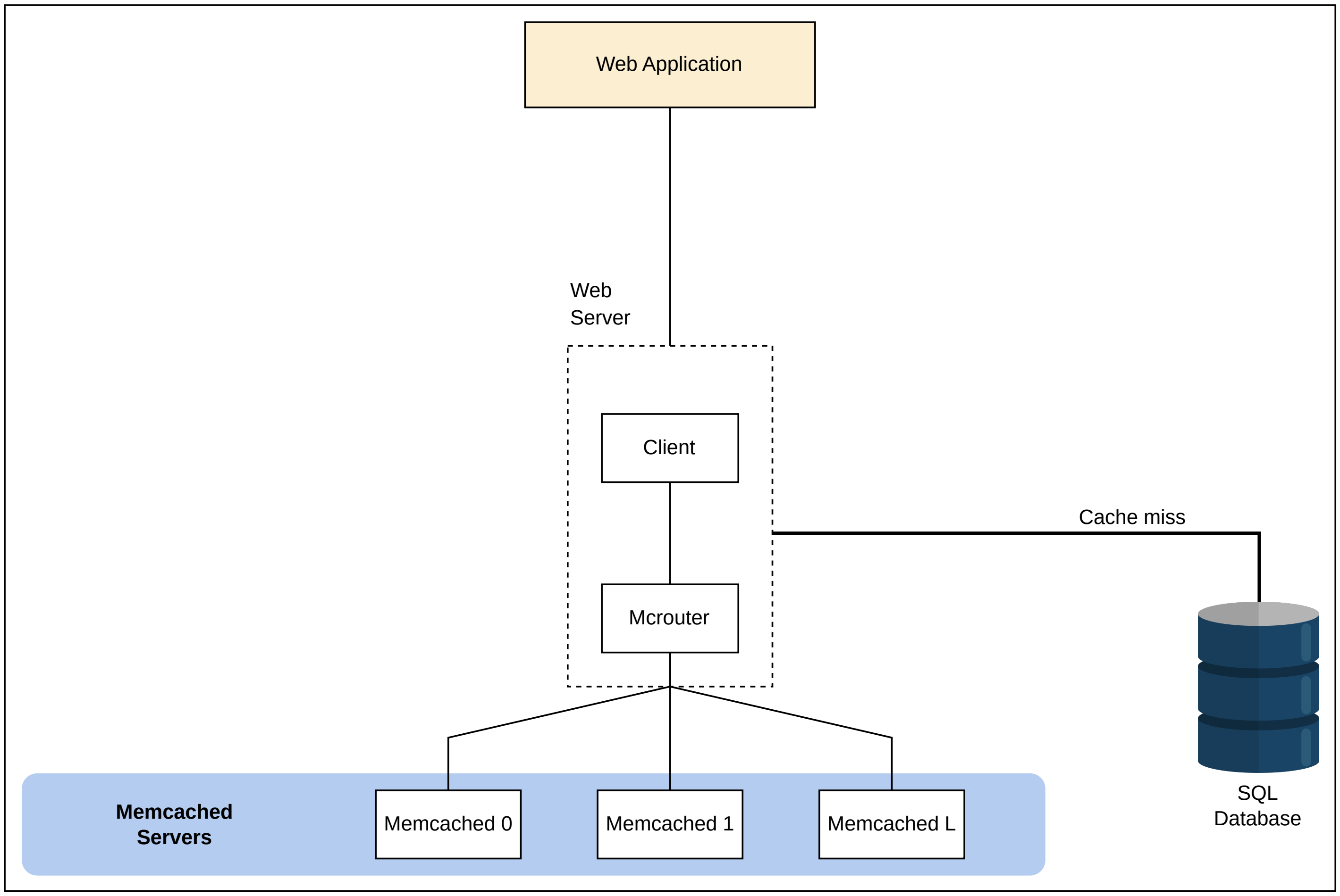
A single server and a single instance of Memcached

1 of 5



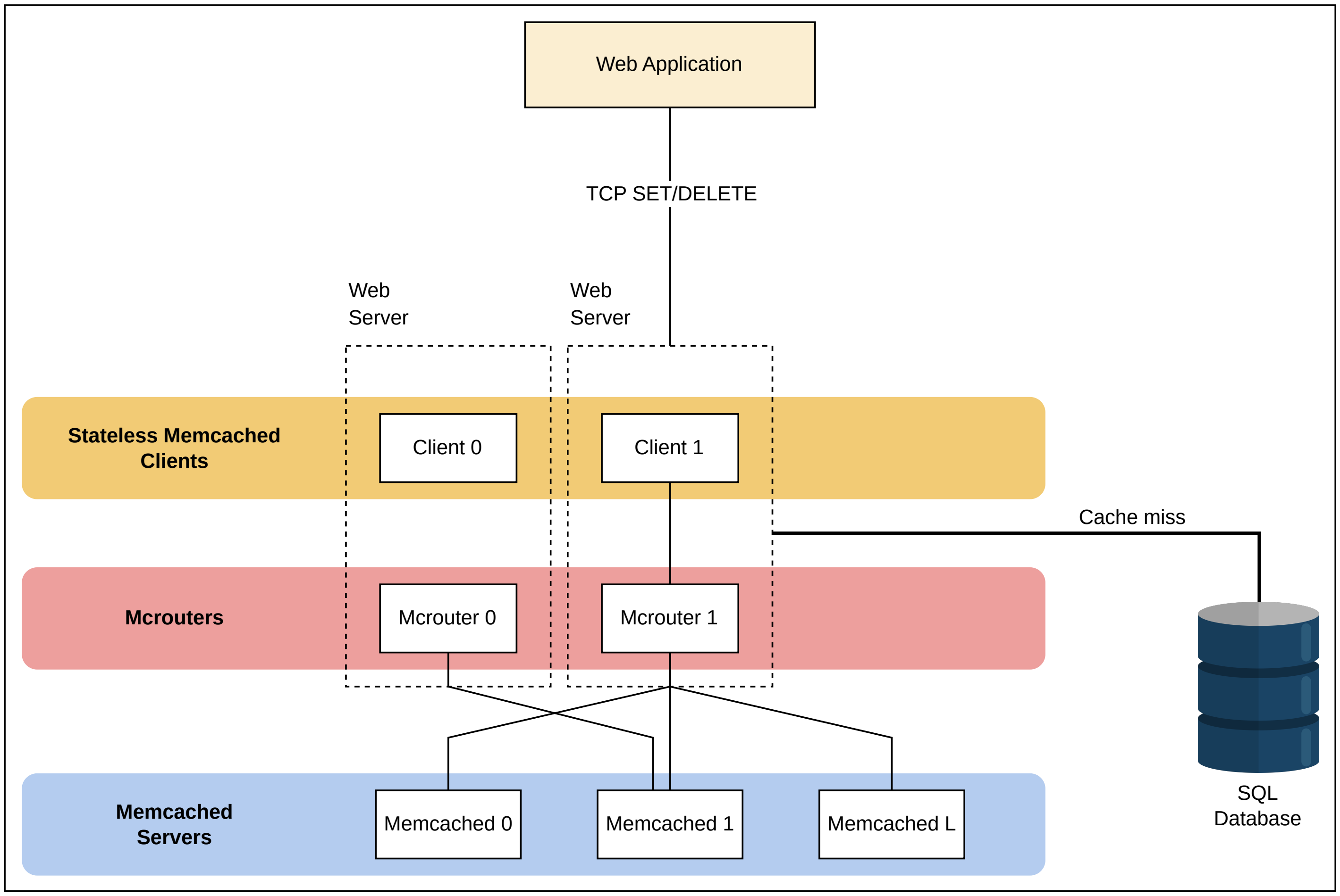
Adding a stateless client to talk to the Memcached server

2 of 5

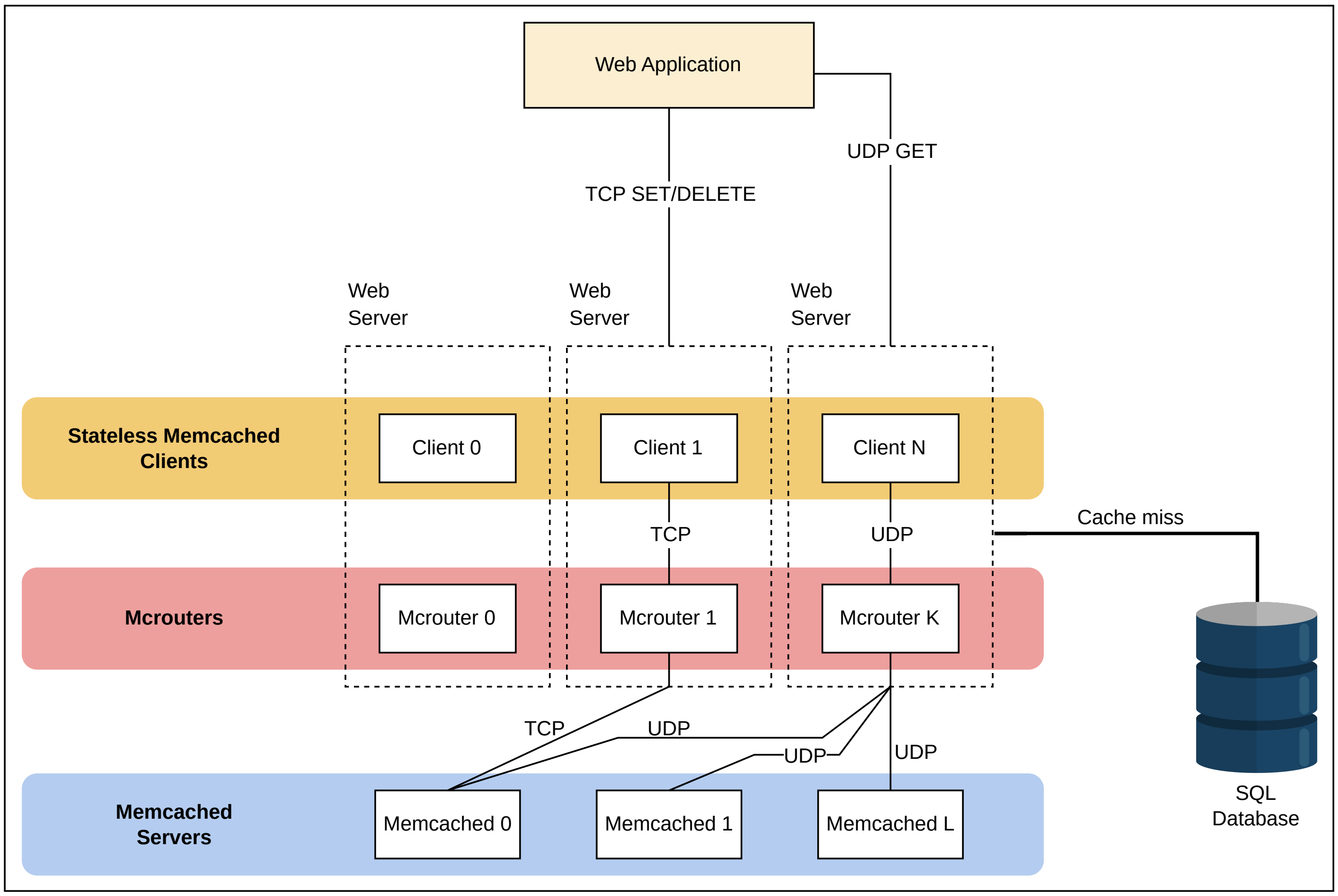


Adding a Mcrouter to allow scaling to multiple Memcached instances

3 of 5



Using multiple frontend clusters for the same Memcached server



The overall design of the scaled version of Memcached. User requests can use UDP or TCP protocol as per need

5 of 5

Let's discuss cluster-level optimizations we can make to improve our caching system.

Network efficiency problem 1: Data dependencies

Problem: How do we reduce the number of network round trips? When a lot of requests need to be handled together, we can usually batch them. In this case, we need to be careful that there might be data dependencies between the requests.

Solution: We'll use a Directed Acyclic Graph (DAG) to maximize the concurrent calls that can be made to Memcache. Rather than making individual updates serially, we can use a DAG to identify which operations are independent of the others and run them in parallel.

Using data dependencies, the web application constructs a DAG to send it to the web server. The web server uses the DAG to execute as many operations concurrently as possible to reduce network round trips for a given request.

Network efficiency problem 2: Comparing UDP and TCP

Problem: When getting data, which protocol should we use? Remember that we need to reduce latency.

UDP is much faster due to it having a connectionless protocol and a smaller header size. However, TCP is more reliable in getting data to the destination than UDP. Does using UDP instead of TCP bring more performance benefits even with the packet loss rate increase?

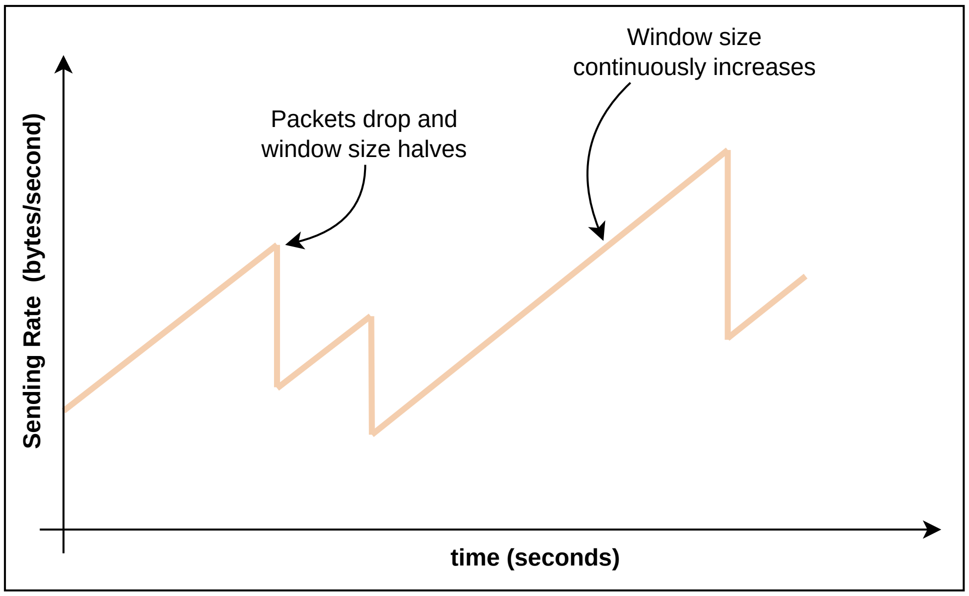
Solution: Compare the latency between the two protocols while fetching data.

To keep the system simple, we'll maintain stateless Memcache clients where possible. This allows us to deploy new versions easily. This also allows us to handle most of the complexity on the client side. We then use a Mcrouter to route data from the Memcache client side to the server. Memcache clients also have a map of all the servers they can access. This map is updated using an auxiliary service. According to a study, there is roughly a 20% advantage to using UDP for `get` requests over TCP. We can also decrease network load by directly allowing the webserver to access the Memcache servers while fetching data.

Network efficiency problem 3: Incast congestion

Problem: Incast congestion occurs when a large number of responses to requests overload a data center's cluster and rack switches. This is a problem that comes when there is an "all to all" pattern in communication, which comes when web servers need to communicate with many Memcached servers to complete a user request.

Solution: Use a sliding window mechanism—similar to what TCP uses to implicitly manage congestion.



The sending rate increases when successful acknowledgments are received, and decreases when data gets lost

We do not want to overload a server or the network with multiple parallel requests. To do this, we can use a sliding window mechanism. As shown in the illustration above, the sliding window mechanism grows its window size when a successful response is received from the server. This mechanism takes into consideration Little's law to change the window size. Little's law helps us to find the average number of requests in a system in a steady state. The law is provided below.

$$L_{\text{No. of queued requests}} \propto \lambda W_{\text{Time taken to process request}}$$

$$\begin{gathered}\text{L: Average number of items in a system.}\\ \text{W: Average time an item spends in the system.}\\ \lambda\text{: Average arrival and departure rate of items in and out of the system.}\end{gathered}$$

Since incast congestion occurs due to an overload of simultaneous Memcache requests, a smaller window size would mean less throughput and a longer queue, while a larger size would mean a higher throughput and shorter queue.

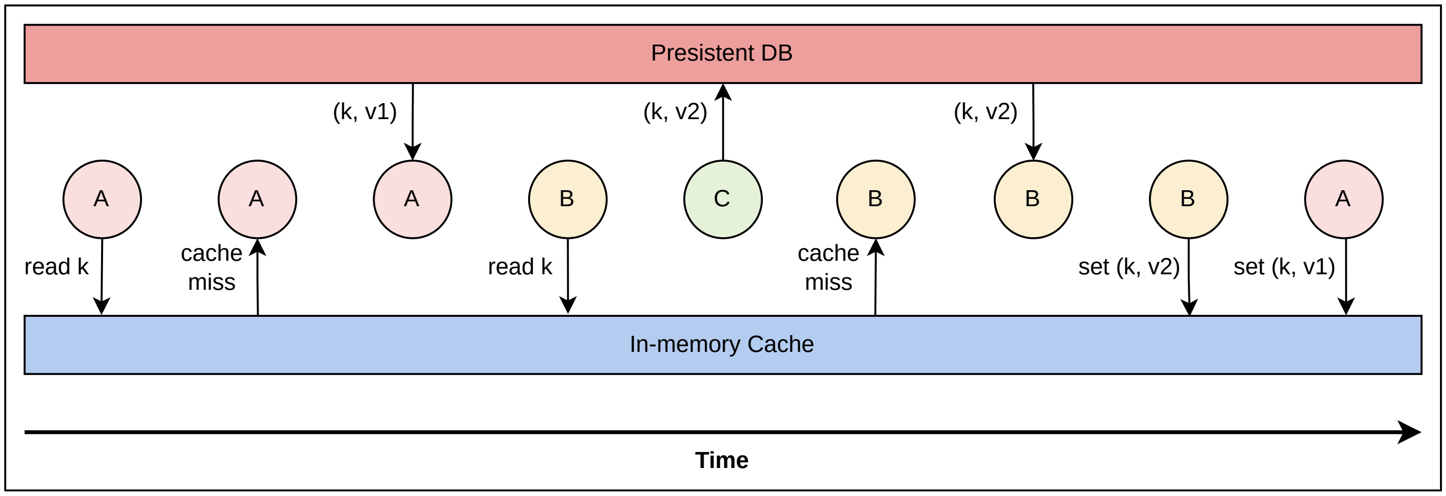
> **Note:** The windowing mechanism used here is similar to what TCP uses for flow control—the maximum amount of data that can be sent to the receiver without overwhelming the receiver.

The repeating cache miss problem

Sometimes we repeatedly face unnecessary cache misses. Let's see why this happens and what we can do to stop it.

Causes of high load

Problem: What causes the unnecessary load to build up? Stale sets and thundering herds cause excess load on the system.



Illustrating a stale set and a thundering herd

As illustrated above, we can have two problems when multiple clients are using a cache.

Stale sets

Stale sets occur when the sequence of concurrent operations gets reordered. In the example above:

1. Client A faces a cache miss for key K and fetches and stores the data from persistent storage.
2. Client B also faces a cache miss, but it fetches an updated value for key K from the persistent storage and updates it in the cache.
3. Now, when Client A sets the value it had fetched earlier after Client B had already updated it, the cache would end up storing an older version of the value for key K.

Thundering herds

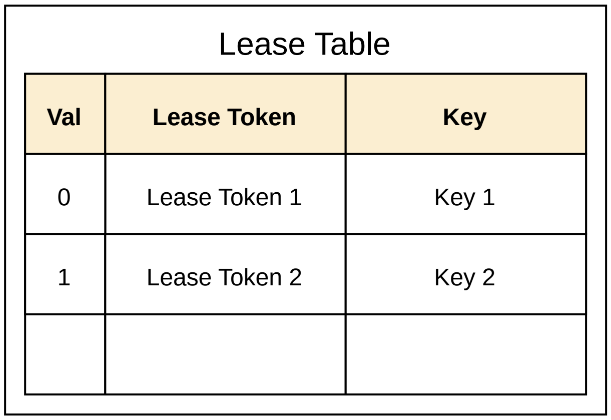
Thundering herds occur when multiple clients start to heavily read and write values to a key. In the example above:

1. Both clients A and B get a cache miss.
2. Both clients fetch data from the database.

If we scale this to millions of clients, we'll have a thundering herd problem. As in a highly concurrent system, many clients will contact the database in a very short amount of time, this can lead to database failure due to the influx of requests.

Using leases to reduce load

Solution: We can use leases to manage stale sets. We can "lease" the key-value item to a client by allocating it to a 64-bit token that is bound to that specific key. To set the data, the client has to reproduce the lease token it was given. A lease token can be invalidated if the Memcached server receives a `delete` request for its key. This leasing system is similar to how load-linked and store-conditionals work at the operating system level.


Lease Table

Val	Lease Token	Key
0	Lease Token 1	Key 1
1	Lease Token 2	Key 2

External data structure to manage concurrency on the Memcached server

Slightly modifying this leasing system allows us to handle thundering herds. Leases get allotted, by default, for 10 seconds. During those 10 seconds, other requesting clients receive a notification to try again later. Usually, a lease gets released within a few milliseconds.

Diverse application needs problem

Problem: Different workloads have different requirements. Using the same systems to handle workloads with very opposing characteristics can be inefficient.

Solution: Create pools based on the different patterns of the data being cached. In our case, we have two pools for the two major workloads. The working set is approximated by sampling items from the whole set of key-value items. We can log the minimum, average and maximum size of these items. The difference between the daily and weekly working sets can help us to identify low-churn keys from high-churn keys.

- **Low-churn keys:** These are keys that have a slower rate of leaving Memcached. For example, keys that become popular quickly end up staying in demand only for a brief period, thereby taking up memory that could've been used for other keys.
- **High-churn keys:** These are keys that have a consistent but sparse demand and get evicted quickly due to more in-demand keys. Hence, they leave Memcache too quickly.

Therefore, appropriately identifying keys based on their churn and storing them together can give us better cache utilization, since reason is that storing keys with different churn characteristics results in them interacting negatively with each other.

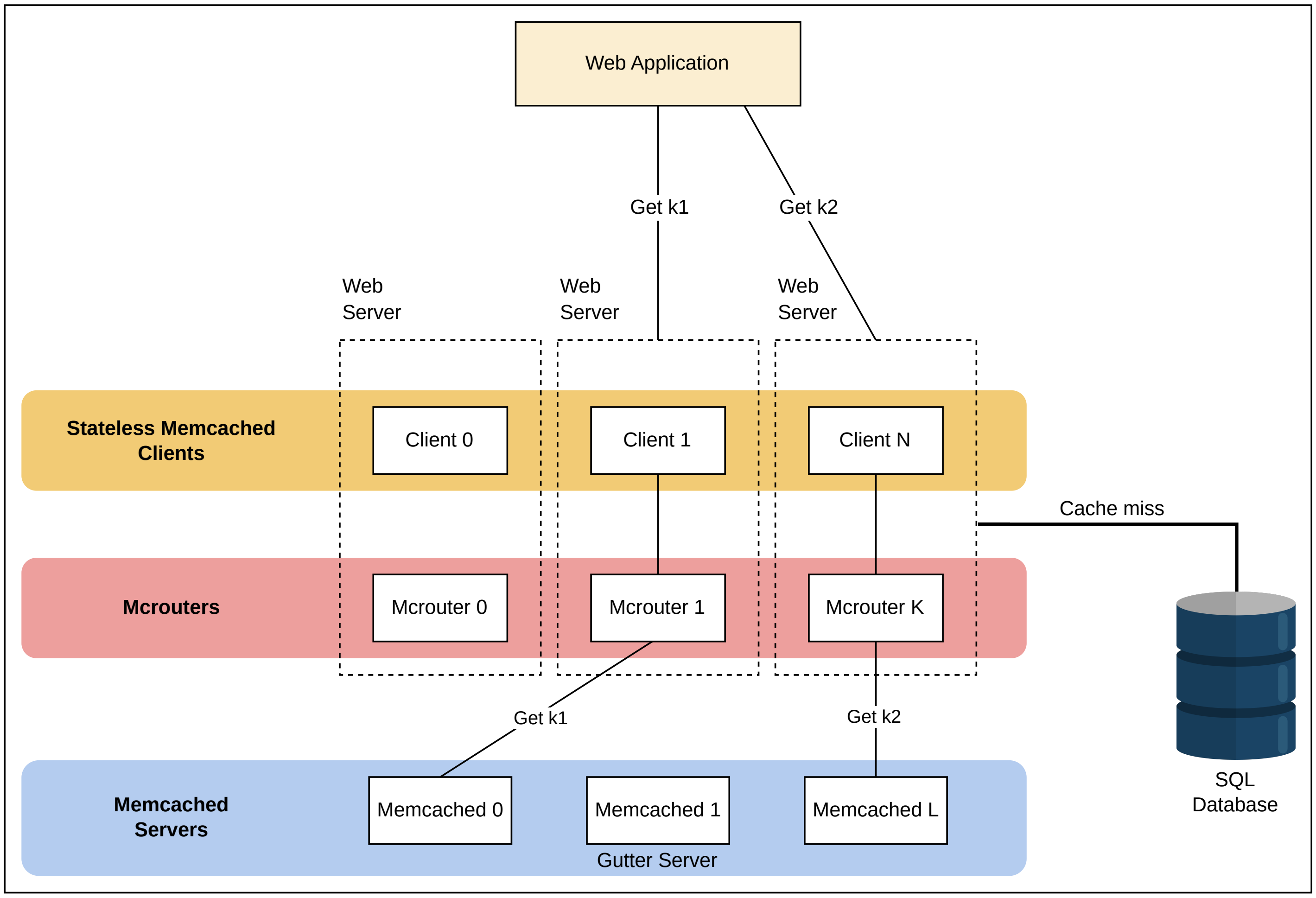
Replication within pools

A cluster can have multiple pools for different use cases, and a pool with replicated Memcached servers is called a replicated pool. We can use replication within pools to improve latency and divide the load when the request rate is too high and the size of the data is small enough to be replicated easily (that is, it can fit onto one or two Memcached servers).

So, when the request rate is too high for one server to manage alone, replicating the key-value set is better than splitting the key-value set between two or more Memcached servers, as this means that a request doesn't have to check both servers. If we have two Memcached servers in a pool, both can serve 50k requests from a total of 100k requests.

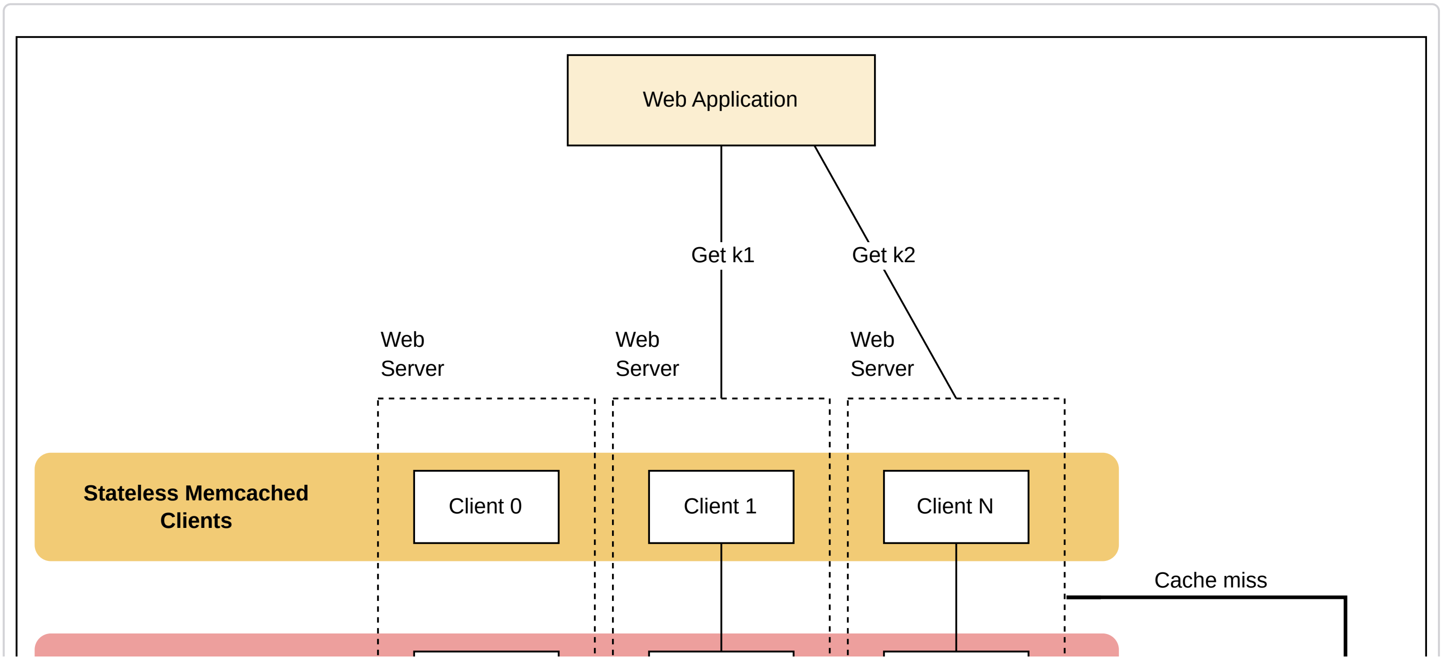
Handling failures

Failures are common when dealing with large distributed systems. Some node in some data center is always crashing. Let's see what we can do to mitigate it's adverse affects.



Getting items from Memcached

1 of 4


Web Application
Get k1
Get k2
Web Server
Web Server
Web Server
Stateless Memcached Clients
Client 0
Client 1
Client N
Cache miss