# Crowd Control: A Consensus Algorithm for Read-Dominant Caches

Shubham Goel and Karthik Viswanathan
Code: <https://github.com/karthikv/crowd-control>

## 1. Introduction

Caches are a core component of modern systems, reducing the latency of expensive disk, database, or network accesses. But caches are limited in size; in order to provide low latency, they generally store data in DRAM/SRAM, which has limited capacity. Distributed caches, such as Facebook's memcached, leverage multiple machines and consistent hashing to extend the size of a cache while spreading the load.

At the core of such distributed systems is replication. Cached key-value pairs are replicated across multiple machines in an effort to provide fault-tolerance. If any one machine dies, the system can generally recover without losing the cached contents, thereby preventing expensive accesses. These additional machines are also leveraged for enhanced throughput. Ideally, operations can be distributed across machines, allowing the system to process more requests in a given amount of time.

Crowd Control is a consensus algorithm that replicates key-value pairs across machines, while optimizing for low latency and high throughput. It is aimed at read-dominant (fewer than 10% writes) workloads. Crowd Control sacrifices some consistency and durability to achieve high performance. Unlike other consensus protocols, it's natural in Crowd Control for a machine to get out-of-sync—for it to have multiple key-value pairs not present in other nodes or vice-versa. If a node misses set operations, the system can continue running, while storing minimal state. In fact, Crowd Control's consensus is focused on replicating metadata, keeping track of which nodes store which keys, and not the key-value pairs themselves. This design decision results in higher throughput and less stored state at the cost of more frequent recovery.

## 2. Design

### 2.1 Leader Election

Crowd Control takes inspiration from view-based consensus algorithms, like Viewstamped Replication and RAFT. Given that a majority of machines are running, one is elected as the primary. The primary coordinates and orders operations to keep the cluster consistent.

To prevent contention on whom the primary is, Crowd Control uses views, also called terms. When a node is elected as the primary, it forms a new view, identified by an integer view number. Upon creation of subsequent views (due to failures or network partitions), the view number is incremented. Consequently, each view number is associated with one and only one primary. A given node only successfully communicates and performs operations with other nodes in its view. If a node sees a higher view number than its own, it updates its view and primary accordingly. Hence, if two nodes both think they're the primary, the one with the higher view number wins, thereby circumventing contention.

Crowd Control elects a primary using RAFT's randomized leader election [3]. Each node maintains a random timeout between 150-300ms. If it doesn't hear from the current primary within this time, it begins an election, requesting votes from everyone in the cluster. If a majority of nodes vote for a given node, that node becomes the primary.

Nodes give out their vote for a given view to the first node that requests it. Due to the range of possible timeouts, it's likely that two nodes won't vye to be the leader at the same time, preventing split votes. Even if there are split votes that prevent any one node from getting a majority, a node can simply increment its view number and try again.

There is a safety concern with leader election: we don't want an out-of-date node to become the leader, as it could compromise the cluster metadata. We address this concern in section 3.1.

## 2.2 Bloom Filters

The primary is used to coordinate and order operations. If all requests go through the primary, however, it poses a bottleneck. Ideally, we'd want requests split across machines so we can achieve higher throughput. Since we're focusing on read-dominant workloads, we'd like to allow get requests—as they constitute the majority of work—to be handled by any machine in the cluster, without having to hit the primary.

If a node in the cluster is partitioned from the primary, or simply hasn't received the last few operations due to network congestion, it won't have the latest key-value pairs. A get operation to this node, then, could yield stale data or no data. As a result, we can't simply let any node in the cluster respond to get requests. We must have a concept of which nodes are up-to-date—which nodes have received the latest operations. More accurately, we can keep track of which keys a given node is up-to-date for.

To conserve space, we store which keys a node is not up-to-date for. If the node is functioning properly, there should be relatively few keys to store, especially considering that the workload is read-dominant. Nevertheless, if a node goes down, it could miss many set operations. The keys for these set operations may be large, taking up significant space.

To further reduce space usage, and allow for speedy recovery, we store these keys in a counting bloom filter. A bloom filter is a space-efficient set that has a small probability of false positives, but no false negatives. In other words, for a given key X, a bloom filter can respond that X is very likely in the set, or that X is definitely not in the set. We use a counting bloom floor to allow for removals.

We've configured our filter to have n = 256 (i.e. capacity for 256 elements). Each time an element is inserted, it's hashed by k = 6 different hash functions. The results are used to index into an array of size q = 4096. Each element in the array is a counter that's incremented. Counters are represented by 4 bits, for a total space usage of 4096 * 4 bits = 2 KB of space. The false positive rate for this configuration is approximately ~0.05%.

If a key k is in a node's bloom filter, that node is not up-to-date for k, meaning it cannot respond to get requests for k. Note that a false positive from the bloom filter doesn't compromise safety. The filter may tell us a key is present when it actually isn't. In this case, a node will think that it's not up-to-date, and it won't respond to that get request, thereby keeping safety intact.

Every node in the cluster keeps track of the bloom filter not only for itself, but also for all other nodes. This allows for recovery; if a node is partitioned, the other nodes can continue handling operations, updating the partitioned node's bloom filter locally as needed. For instance, if there are set operations for keys $k_1$ and $k_2$ while a node n is partitioned, the other nodes will add $k_1$ and $k_2$ to n's bloom filter.

## 2.3 Get Operation

`get(key)` returns the value associated with the given `key`. To perform a get operation, a node first checks if the key is in its bloom filter. If it is, it responds to the client saying that it doesn't know. It then fetches the key-value pair from another node (generally the primary) in the background (we discuss this more in section 3.2). If, however, the key is not in the bloom filter, the client looks up and returns the associated value.

There could be a chance that a node's bloom filter isn't up-to-date, thereby causing it to mishandle requests. To remedy this, the node requests a lease from the primary prior to responding to get requests. It sends a hash of its bloom filter, which the primary compares with its local copy. If the hashes match, the primary grants a 10 second lease to the node to respond to get requests (i.e. the node no longer needs to consult the primary about its hash for the next 10 seconds). If the hashes don't match, the primary simply sends the correct bloom filter and grants the lease. Note that this is a cheap operation, since the bloom filter is only 1 KB.

When a client makes a get request to the cluster, it can communicate with any machine. To minimize latency, the client can send parallel get requests to multiple machines. Even if one node is slow or outdated,

another node can make up for it. The client can choose how many nodes to send requests to. More machines will result in more accurate and quick replies at the expense of potentially congesting the network.

## 2.4 Operation Log

The primary keeps an in-memory operation log to bring nodes up-to-date. Being up-to-date means that a node has the latest local bloom filters for itself and for all other nodes. It doesn't necessarily mean that a node has the same key-value pairs in its cache as another node. For instance, node 1 might have missed the set operation $k_1 \rightarrow v_1$. Node 2 might not have missed this operation. As long as node 1 and node 2 both have correct local bloom filters for themselves and for each other (i.e., node 1's filter should contain $k_1$ and node 2's filter should not), they are up-to-date.

The operation log, then, is solely used to record changes to bloom filters; it doesn't contain the key-value pairs that are passed to set operations. There are four possible operations types in the log:

1. `add(key)` adds `key` to the local bloom filters of all nodes
2. `remove(key, nodes)` removes `key` from the bloom filters of each node in `nodes`
3. `setFilter(node, filter)` sets the bloom filter of `node` to `filter`

For each node in the cluster, the primary maintains a pointer, keeping track of the first operation that node hasn't yet executed in the log. Operations that precede all pointers can be garbage collected, since they're not needed by any node.

To conserve space, the primary limits the operation log to 256 operations in length. This restriction, combined with the need for garbage collection, makes a ring buffer an ideal data structure to store the log. See section 3.7 about handling operation log overflow.

## 2.5 Set Operation

`set(key, value)` associates the given `key` with the provided `value`. To maintain a consistent ordering, set operations are only performed by the primary. If the primary receives multiple set operations concurrently, it serializes them. To perform a set, the primary appends `add(key)` to the operation log. It then executes this add operation, adding the key to all of its local bloom filters. For each node n, the primary sends a `prepare(key, ops`$_n$`)` RPC, where `ops`$_n$ is an array of operations from the operation log that node n needs to execute to get up-to-date.

When handling a `prepare(key, ops`$_n$`)` RPC, a node executes all the operations in `ops`$_n$ to get up-to-date. Then, it responds affirmatively to the primary, indicating that it's ready to receive the value and commit. The primary collects responses until it has a majority. At this point, before committing, it responds to the client with a success message. Although the `key` hasn't been associated with `value`, the system is in a safe state. If the client performs a get operation, nodes will return an "I don't know" response, as they have the key in their filter. In the worst case scenario, this will cause the client to perform another set operation. The primary, however, detects duplicate sets and ignores them. This design decision reduces latency, allowing the set to return quickly while the commit is happening in the background.

Right after responding with a success message, the primary sends `commit(key, value)` RPCs to every node. The primary keeps track of which nodes commit. For each node that has committed, the primary removes `key` from that node's local bloom filter. It eventually adds a `remove(key, nodes)` entry to the operation log, which will get synced across nodes during the next set operation.

When handling a `commit(key, value)` RPC, a node associates `key` with `value` in its cache, and removes `key` from its filter.

## 2.6 Cache Eviction

A node will follow the standard LRU eviction policy if its cache is full. If a get request is made for an evicted key, a node will respond with no data. Evicted keys are not added to a node's filter. This circumvents the need to update the operation log.

# 3. Safety and Fault Tolerance

## 3.1 View Changes

Each operation in the log is associated with an operation number. The operation number monotonically increases as operations are added to the log. To maintain safety across view changes, nodes keep track of the latest operation number they have executed from the log. A node $n_1$ will not grant its vote to another node $n_2$ during leader election if $n_1$ has executed a later operation than $n_2$. Since a majority of nodes will always be up-to-date from the last set operation, one of these nodes must be elected the new leader (any successful election will require at least one node from this majority to vote).

When a new view is formed, the primary seeds the log with `setFilter()` operations for each node in the cluster. This ensures that nodes joining the view first synchronize their state with the primary prior to executing additional operations.

## 3.2 Recovering Individual Key-Value Pairs

On a get operation, if a node has a key in its bloom filter, it responds with "I don't know" and proceeds to recover that key-value pair. The recovery is important; without it, the node will be unable to respond to future get requests for that pair, thereby reducing throughput.

To recover a key-value pair, a node simply asks the primary for it. Once the transfer is complete, both the node and primary update their local filters. The primary appends `remove(key, nodes)` to the operation log to reflect the transaction.

## 3.3 Hard Crashes

If a machine suffers a hard crash, it loses its volatile memory, including its cache data and bloom filters (all CrowdControl data is stored in RAM, so the node needs to fully recover). When this machine rejoins the system, it broadcasts a recover request. The primary responds with its view (to disambiguate which node is the real primary), its own cache data, and its local bloom filters, bringing the machine up-to-date. During this time, the recovering node cannot respond to any requests. Once the transfer is complete, the primary then appends `setFilter(node, filter)` to the operation log, setting the recovered bloom filter of the crashed machine. The crashed machine is now up-to-date and can start responding to requests.

## 3.4 Revoking Leases

Before returning from a set operation, the primary must do one of two things for each node with an outstanding lease: (a) ensure that the node successfully responds to the `prepare()` RPC, or (b) revoke that node's lease. If neither (a) nor (b) is satisfied for a node with an outstanding lease, that node can return stale data for the next get operation, since its filter is no longer up-to-date (it wasn't involved in the latest set operation).

The primary first tries to involve the node in the `prepare()` RPC. If that doesn't work, it attempts to revoke its lease. If the node doesn't respond to the `revokeLease()` RPC, one option would be to wait for the lease to expire. This, however, is very expensive, as get leases last for 10 seconds.

To account for this case, we make a loose synchrony argument. If we aren't able to send `revokeLease()` RPCs, we very likely aren't able to send `heartbeat()` RPCs either. Hence, we keep re-trying the `revokeLease()` RPC for 300ms, the maximum election timeout. If, after 300ms, the node hasn't responded, it very likely hasn't seen `heartbeat()` RPCs for that time either. Consequently, that node would

start an election to become the new primary. Upon starting an election, a node clears its lease, so we no longer need to be concerned about it.

Albeit unlikely, after 300ms, it's possible that some of our `heartbeat()` RPCs got through but our `revokeLease()` did not. If this occurs, the node with the lease can return stale data. We sacrifice consistency in this edge case for low latency, as we don't want to be waiting 10 seconds to finish a set operation.

### 3.5 Network partitions

Let's consider a five node cluster with nodes $n_1$ through $n_5$. Node $n_1$ is the primary. $n_1$ and $n_2$ get partitioned. $n_3$, $n_4$, and $n_5$ form a new view with a new primary and start handling requests. Note that $n_1$ is now unable to perform set operations, since it can't communicate with a majority of the cluster. The problem, however, is that $n_1$ can give leases out to itself and $n_2$, allowing $n_1$ and $n_2$ to still respond to get requests. This is problematic if sets are occurring in the newly formed view, as $n_1$ and $n_2$ will return stale data.

To prevent this case, we use yet another synchrony argument. If the primary $n_1$ cannot send heartbeats to a majority of the cluster for more than 150ms, it stops giving out leases, and revokes any leases it has outstanding. Note that 150ms is the minimum election timeout; it's likely that no other view has formed when this time elapses, thereby preventing $n_1$ and $n_2$ from serving stale data.

Of course, there is a small chance that another view formed before 150ms, or that the primary was unable to revoke the lease for a node, so we give up consistency in these edge cases to retain high uptime/performance.

### 3.6 Full Bloom Filter

Over time, the bloom filters for some nodes may become saturated at their capacity of 256. This can occur if a node is partitioned while many set operations take place. If more keys are added than the filter's capacity, the false positive rate will increase rapidly. This will prevent the node from responding to get requests in the future.

To solve this issue, as soon as a node sees that its bloom filter has saturated, it undergoes a full-recovery, as described in section 3.3.

### 3.7 Operation Log Overrun

The operation log can hold at most 256 elements. If a node has been out-of-date for long enough, the operation log will reach its capacity, as that node prevents the log from being garbage collected. To fix this, the primary will remove all operations in the log that only exist for the out-of-date node(s). During the next prep, that node will realize it's out-of-date (the primary will send a flag indicating this) and recover. Note that the node simply needs to request all filters (both for itself and for other nodes) from the primary to recover..

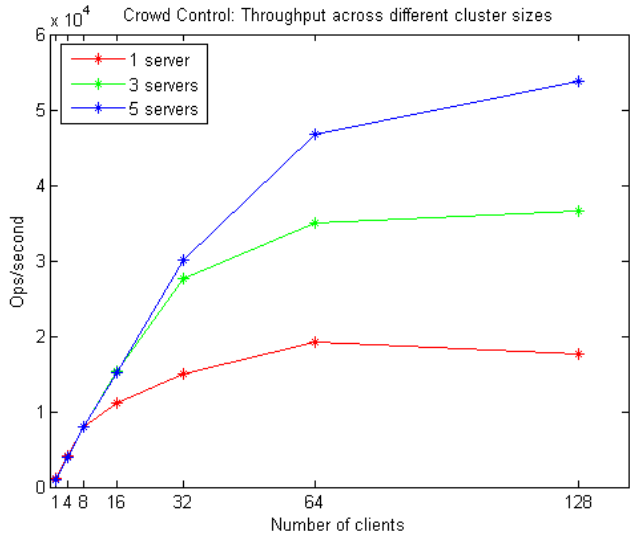
## 4. Testing

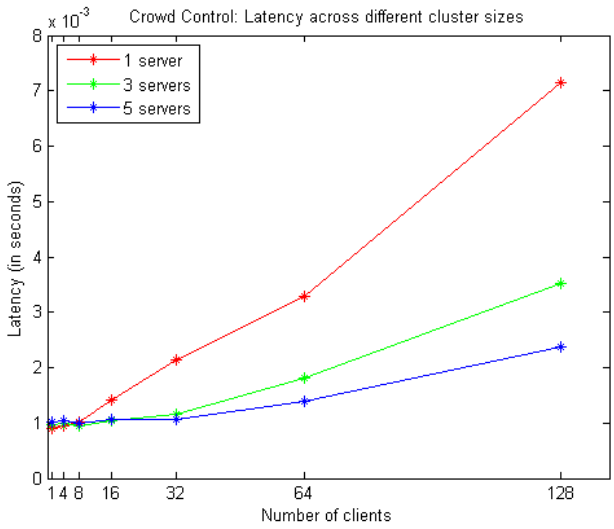
To test our implementation, we wrote an RPC abstraction layer that lets us selectively partition and kill nodes. This helped in assessing the correctness of leader election, get leases, set operation, and recovery. We also had test benches for many submodules within our implementation, such as the counting bloom filter, operation log, and LRU cache.

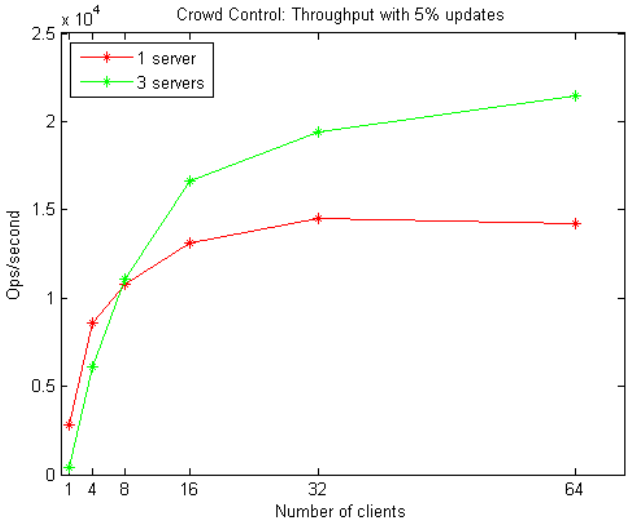
## 5. Measurements

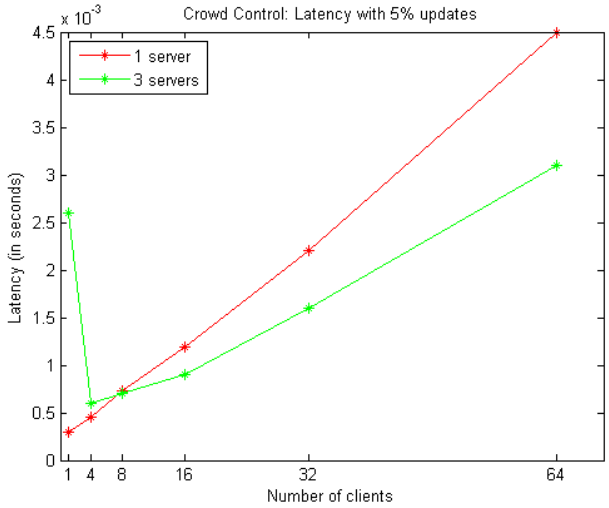
We ran clusters of 1, 3, and 5 machines using Amazon EC2 m3.medium (1 core, 3.75 GB RAM) instances. We used a c3.2xlarge (8 cores, 15 GB RAM) instance for our client to allow for many simultaneous requests.

As the cluster size increases, the read throughput goes up and the latency goes down, as load is spread across the cluster. With more updates, the system slows down, as set requests are expensive. For larger cluster sizes, set latency increases, as prepare messages need to be sent to more machines.




## 6. Conclusion

Crowd Control is a consensus algorithm for caches that replicates key-value pairs. It optimizes for performance on read-dominant workloads by sacrificing some consistency and having rather frequent recovery. To increase throughput, individual nodes obtain leases to handle get requests without involving the primary. To curb latency, the primary responds to set requests before they're fully committed. Using counting bloom filters and an in-memory operation log, Crowd Control replicates space-efficient cluster metadata to keep nodes up-to-date. Many of the recovery and fault tolerance cases only touch the metadata so that nodes can recover quickly.