High Availability with Error Correction Codes in Distributed Caching

Felix Chen *
University of Toronto
Toronto, Canada
felixj.chen@mail.utoronto.ca

Abstract

Error correction codes may be a viable option to support high availability in distributed caches. Typically, data replication is used to tolerate node failure, but data replication demands at least 50% redundancy. Using error correction codes, node failures can be tolerated while using less than 50% redundancy. A distributed cache using error correction codes may use a fraction of the memory required for a replicated cache. In this report, two approaches for distributed caching are compared, replication and error correction codes.

1 Motivation

At the moment, Redis is a popular cache [1] that provides fault tolerance by replicating data [2]. Redis uses a leader-follower architecture, where a leader node handles incoming client requests, and follower nodes strive to be replicas of the leader. When a leader fails-over, a follower will be promoted to the leader. Redis's replication strategy demands at least 50% of nodes contain redundant data. Similar replication strategies are used among other distributed caches such as Ehcache [3], Hazelcast [4] and Riak [5].

Error correction codes (ECCs) encode messages into codewords by constructing redundant data. Using the constructed redundancy, codewords can suffer from small errors and still be decoded into the original message. Importantly, when using ECCs the percentage of redundancy can be less then 50%. In practice, ECCs are used in applications such as DVDs and RAID, where physical damage or drive failures can be tolerated.

2 Replicated Cache

2.1 Introduction

Creating a replicated cache is non trivial, common difficulties in distributed systems such as fault tolerance, consistency and latency need to be considered. A replicated cache is an example of a replicated state machine and consensus algorithms such as Raft [6] or Paxos can be used to maintain a replicated state machine.

2.2 Implementation

To implement the replicated cache [7], the Raft consensus algorithm was selected for its availability and consistency, the Rust programming language was chosen for it's novelty and speed, and gRPCs were chosen for their performance. The replicated cache contains two main programs, a server and a client.

Every server is an individual Raft node that leverages the async-raft [8] implementation of the Raft consensus protocol. Async-raft requires custom implementation of storage and networking logic. Server networking uses Tonic gRPCs [9] to communicate with other raft servers using the same RequestVote, AppendEntries and InstallSnapshot RPCs as described in the Raft paper [6]. Server storage includes a replicated log, where logs entries are replicated to all other raft peers and compacted every 5000 entries with Serde [10]. Each server builds an identical replicated cache by eventually applying all committed Raft logs.

The client has two commands, set and get. An important note is that the client will first find the Raft leader before starting a transaction, if the client contacts a Raft follower, the follower will respond with a known leader address. Writes must be done through the Raft leader as per design. Reads must done through the Raft leader to prevent stale reads, as the leader is guarenteed to be up-to-date, whereas followers are not.

2.3 Consistency

Our replicated cache offers strong consistency, unlike Redis Cluster which loses completed writes [11]. In the Raft algorithm, the leader pushes log entries onto all followers, committing logs that are replicated to the majority of nodes, and applying committed logs to the state machine. Raft guarantees that all committed entries are durable and will be eventually applied to every state machine. Our implementation replies to write requests only after the log has been committed, and only elects up-to-date followers to be new leaders. Ultimately, this means the replicated cache contains all completed write transactions.

Additionally, we can say that the replicated cache is linearizable [12]. Transactions have a strict ordering because every log entry includes a transaction and log entries are applied in an identical order, by the Raft log matching property [6].

2.4 Availability

Our replicated cache is available given the majority of nodes are alive and are able to communicate. Since Raft requires majority log replication to commit logs and a majority vote to elect a new leader, Raft will stall if the majority of nodes become unavailable. Otherwise, if a leader node fails over and the majority of nodes are still available, the remaining followers will detect the lack of heartbeat RPCs and trigger an election, electing an up-to-date follower to be the new leader. The newly elected leader will handle all new client requests. If a follower fails over and the majority of nodes are available, new transactions are still committed to the majority of the cluster and progress continues. When the follower node recovers, it will be caught up with snapshots and log entries.

2.5 Comparison

2.5.1 Redis

Compared to our replicated cache, Redis is less consistent but more performant, this is because transactions need to be acknowledged by only a single Redis leader, compared to the majority of our Raft cluster. Similarly Redis trades availability for consistency, given an n node cluster, Redis can survive $n-1$ outages (for each hash slot) compared to Raft which can survive $\lfloor \frac{n-1}{2} \rfloor$, again this trade off comes because Raft needs majority votes and replication. The replicated cache is similar to Redis in that it replicates the cache across all nodes to provide availability, resulting in $n-1$ nodes dedicated for redundancy and overall redundancy of $\frac{n-1}{n}\%$.

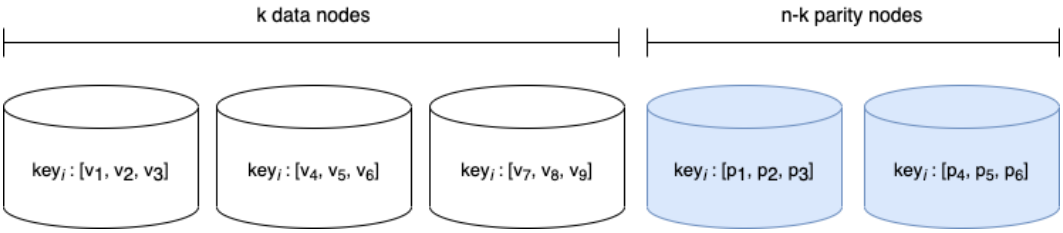

Figure 1: Block level stripping with two parity nodes

3 Error Correction Code Cache

3.1 Introduction

The ECC cache is much like RAID, but purely software and located in memory instead of disk. The ECC cache divides written data across a few nodes, storing a single copy of the data, subsequently, parity is computed and stored across the remaining nodes. If a node fails, the parity can be used to reconstruct the original data, for both client reads and node recovery.

3.2 Implementation

To build the ECC cache [13], the Rust programming language, a Reed-Solomon erasure code library [14], Tonic gRPCs [9], and two-phase commit [15] were used. The ECC Cache is close in design to RAID 4 [16], it uses block-level stripping with a chosen number of parity nodes. The ECC Cache also contains server and client programs.

The ECC cache is designed as a distributed key-value store. For each key-value pair, the value is divided into k blocks and stored on separate data nodes, the remaining $n - k$ nodes are used to store computed redundancy. Using Reed-Solomon erasure coding [17] [18], we can reconstruct any value using any k nodes. As long as we have any k nodes, we can continue reads, writes and even reconstruct failed nodes. Using this design, the amount of redundancy and fault tolerance can be chosen by selecting $k \leq n < 2k$, the higher the redundancy, the more faults the cache can tolerate.

The server implements heartbeats, two-phase commit [15], reading values and recovery. The heartbeat is used to maintain a list of healthy nodes, where nodes are considered healthy if they have at least $k - 1$ neighbours. The dynamic list of healthy servers is used in the two-phase commit prepare phase, where the number of locks required for a transaction is in $[k, n]$. We note that the number of locks required involves the majority of the cluster, such that under network partitions, only one partition may make progress. Unhealthy servers will attempt to recover, first by getting all keys from a healthy server followed by populating all values with appropriate data.

The client uses the Tonic gRPCs [9] and Reed-Solomon erasure library [14] to calculate data parity, and write to both data and parity nodes. The client is the transaction coordinator in two-phase commit, it first acquires locks from all healthy servers while temporarily writing a new value, after all locks are acquired the transaction is commuted to the key-value store. If a transaction fails to acquire sufficient locks, it is aborted. Since the client has direct write access to any server, it is assumed to be trusted and available.

3.3 Consistency

Our ECC cache is linearizable because of two-phase commit. A transaction begins with writing the new value ahead and acquiring an adequate number of locks. After acquiring locks from all healthy servers, the transaction completes, the new value is written and locks are released. If insufficient locks are obtained, then the transaction is aborted, the write ahead log is cleared and locks are released. Because every transaction needs the to lock the majority of servers, only one transaction can proceed at a time.

Additionally, if at least k nodes are available, then all completed transactions will be available to the client. The ECC cache cannot lose completed writes like Redis.

3.4 Availability

Similar to the replicated cache, the ECC cache is available as long as the majority of nodes are healthy. If the number of healthy nodes falls below k, the cache is considered invalid and drains all entries and locks. If a server fails and the number of healthy nodes are within $[k, n]$, then the failed server can reconstruct itself, it gets all keys from a healthy node and then begins to rebuild data or parity. Additionally, if servers fail and the number of healthy nodes are within $[k, n]$, ECC cache clients can make progress on reads and writes, as reading only requires k server responses, and writing needs at least k locks.

3.5 Comparison

3.5.1 Replicated Cache

Compared to the replicated cache, performance and availability is comparable but resource usage should be a fraction. Reading from the ECC cache takes the first k responses, compared to the replication cache which needs at very few responses to find the leader and get a value. Writing performance should also be similar, both the ECC cache and replication cache write to the majority of the cluster before acknowledging writes. Additionally availability is also similar as both caches required the majority of nodes to be alive to make progress. With the $k \leq n < 2k$ constraint on the ECC cache, redundancy always remains below 50%, compared to the replication cache, redundancy approaches 100% as $n \to \infty$. Overall, we expect that the ECC cache uses a fraction of memory resources will providing comparable performance and availability.

4 Benchmarks

4.1 Implementation

To measure resource usage, a Docker image to run either server was created. Netdata [19] was used to generate container network, cpu and memory usage graphs. The benchmarking script [20] is written in Go, where workloads were run on a single thread and some failures are introduced. Workload A consists of 5000 set commands and 5000 get commands, and workload B consists of 9500 get commands and 500 set commands, mimicing YCSB [21]. Workload C contains 10000 set commands, intended to test the memory usage.

4.2 Results

Workload	Type	Maximum Memory Usage (MiB)	Typical CPU Usage (% of core)	Average Transaction Time (seconds)
A	ECC	3.5	7	0.01451
A	Replication	7.5	8.5	0.01403
B	ECC	2.1	7	0.01155
B	Replication	2.5	7.5	0.01172
C	ECC	3.4	9	0.01921
C	Replication	6.5	10	0.01872

Figure 2: Usage over 10,000 transactions

5 Conclusion

6 Future Work

6.1 Sharding

6.2 Replication Cache

6.3 Error Correction Code Cache

- Recover should be blocking or OCC - Build time

References

[1] https://insights.stackoverflow.com/survey/2020#technology-databases.

[2] https://redis.io/topics/replication.

[3] https://www.ehcache.org/documentation/2.8/replication/#other-replication-means-table-of-contents.

[4] https://docs.hazelcast.com/imdg/4.2/data-structures/replicated-map.html.

[5] https://docs.riak.com/riak/kv/latest/learn/concepts/replication/index.html.

[6] https://raft.github.io/raft.pdf.

[7] https://github.com/felixjchen/Distributed-Cache/tree/main/src/raft.

[8] https://github.com/async-raft/async-raft.

[9] https://github.com/hyperium/tonic.

[10] https://serde.rs/.

[11] https://redis.io/topics/cluster-tutorial#redis-cluster-consistency-guarantees.

[12] https://www.cs.princeton.edu/courses/archive/fall16/cos418/docs/L13-strong-cap.pdf.

[13] https://github.com/felixjchen/Distributed-Cache/tree/main/src/ecc.

[14] https://github.com/darrenldl/reed-solomon-erasure.

[15] https://en.wikipedia.org/wiki/Two-phase_commit_protocol.

[16] https://en.wikipedia.org/wiki/Standard_RAID_levels#RAID_4.

[17] https://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction.

[18] https://www.backblaze.com/blog/reed-solomon/.

[19] https://www.netdata.cloud/.

[20] https://github.com/felixjchen/D94/blob/main/benchmark.go.

[21] https://github.com/brianfrankcooper/YCSB.