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 replicating data demands at least 50% of nodes store redundant data. Using error correction codes, node failures can be tolerated while using less than 50% redundancy. A distributed cache using error correction codes may use significantly less memory compared to a replication cache. In this report, two approaches for distributed caching are compared, replication and error correction codes.

1 Introduction

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 all follower 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 some amount of redundant data. Using the constructed redundancy, codewords can suffer from small errors and still be decoded into their original message. In practice, ECCs are used in applications such as DVDs and RAID, where physical damage or drive failures need to be tolerated.

2 replication cache

2.1 Introduction

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

2.2 Implementation

To implement a replication 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 replication 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 RequestVote, AppendEntries and InstallSnapshot RPCs 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 using Serde [10]. Each server builds an identical replication cache by eventually applying all committed log entries in a strict order.

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. Write requests must be done through the Raft leader as per design. Read requests must done through the Raft leader to prevent stale reads, as the leader is guaranteed to be up-to-date, whereas followers are not.

2.3 Consistency

Our replication cache offers strong consistency, unlike Redis Cluster which can lose completed writes [11]. In the Raft algorithm, the leader replicates log entries onto all followers. Once a log entry is replicated to the majority of Raft nodes, the entry is considered committed and is then applied to the state machine. Raft guarantees that all committed entries are durable and will be eventually applied to every state machine. The replication cache leader only replies to set requests after the log has been committed. Ultimately, this means the replication cache leader contains all completed write transactions.

Furthermore, we can say that the replication cache is linearizable [12]. Transactions have a strict ordering because log entries are applied in an identical order, by the Raft log matching property [6]. Therefore transactions are isolated from each other, and complete atomically.

2.4 Availability

Assuming the majority of nodes are available and are able to communicate, the replication cache is available. 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. 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. As long as the majority of the nodes are available, the replication cache can make progress.

2.5 Comparison

2.5.1 Redis

Compared to our replication cache, Redis is less consistent but more performant and available. This is because transactions need to be acknowledged by only a single Redis leader, compared to the majority of our Raft cluster. Redis can survive $n-1$ outages (for each hash slot) compared to Raft which can survive $\lfloor \frac{n-1}{2} \rfloor$, this trade off comes because Raft needs the majority of nodes for voting and log replication.

The replication 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. The replication cache and Redis's memory usage similarities are important, since memory usage is the key metric the ECC cache is designed to reduce.

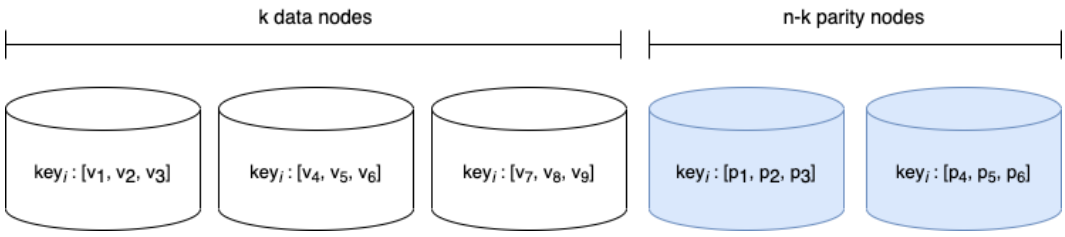


Figure 1: Block level stripping with two parity nodes

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, additionally, parity is computed and stored over the remaining nodes. If a node fails, the parity can be used to reconstruct the original data, this is used in both client reads and server reconstruction.

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] was used. Reed-Solomon codes were chosen because of flexible block length and strong correction ability as an erasure code. The ECC cache is close in design to RAID 4 [16], it uses block-level stripping with a configurable number of parity nodes. Like the replication cache, the ECC cache also contains server and client programs.

The ECC cache is designed as a distributed key-value store over n nodes. 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 n, the higher the redundancy, the more faults the cache can tolerate. One assumption we make is $k \leq n < 2k$, that is our redundancy is strictly less then 50% and that k nodes form a majority.

The server implements heartbeats, two-phase commit [15], reading values and recovery. A 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 two-phase commit's prepare phase, where the number of locks required for a transaction is in $[k, n]$, depending on cluster health. The number of locks required for a transaction involves the majority of the cluster, such that under network partitions, at most one subset of nodes may make progress. Lastly, unhealthy servers will attempt to recover, first by getting all keys from a healthy server followed by populating all values with appropriate data or parity.

The client uses Tonic gRPCs [9] and a Reed-Solomon erasure library [14] to calculate data parity, and write to the cluster. A set transaction begins with writing the new value ahead to a temporary location and acquiring a lock from every healthy server. After acquiring locks from all healthy servers, the transaction is committed, 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. The client has direct lock and write access to any server, so it is assumed to be trusted and available. Client reads use the Reed-Solomon erasure code to construct the original value given the first k server responses.

3.3 Consistency

If at least k nodes are available, then all completed write requests will be available to a client. This is because the client uses the Reed-Solomon erasure library [14] to build values using the first k responses during any read request. The ECC cache cannot lose completed writes like Redis.

Additionally, the ECC cache is linearizable because of the two-phase commit protocol . Since every transaction needs to acquire a lock from the majority of servers, a second simultaneous transaction on the same key will either abort both transactions or the transactions will proceed one at a time. Two-phase commit prevents write skew and makes transactions atomic.

3.4 Availability

Similar to the replication 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 it's data

by retrieving values from other servers. Additionally, if servers fail and the number of healthy nodes are within $[k, n]$, clients can make progress on reads and writes, as reading only requires k server responses, and writing needs at least k locks. As long as k servers are available, the ECC cache can make progress on reads, writes and server recovery.

3.5 Comparison

3.5.1 Replication Cache

Compared to the replication cache, performance and availability is comparable but the ECC cache uses less memory. Read performance is similar as the ECC cache takes the first k responses, compared to the replication cache which needs 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. The availability of both caches is similar as both caches required the majority of nodes to be available to make progress. Since the ECC cache has a $n < 2k$ constraint, the ECC cache's redundancy must be below 50%. A replication cache with 3 nodes, will have redundant replicas on two followers, that is 66% of the cache will be redundant. Overall, we expect that the ECC cache uses less memory resources while providing comparable performance and availability compared to the replication cache.

4 Benchmarks

4.1 Implementation

To measure resource usage, a Docker image capable of running either the replication cache server or ECC cache server was created. Netdata [19] was used to generate container network bandwidth, CPU usage and memory usage graphs.

The benchmarking script is written in Go, where workloads execute on a single thread and failures may be introduced. Workload A consists of 5000 set commands and 5000 get commands, and workload B consists of 9500 get commands and 500 set commands, mimizing Yahoo! Cloud Serving Benchmark (YCSB) [20]. Workload C contains 10000 set commands, intended to test the memory usage of both caches. All set commands increment a counter and create unique keys, get commands get from random keys.

During benchmarks, the ECC cache had $k = 2$ data servers, $n = 3$ total servers, and a block size of 8 bits. The replication cache had 3 total servers running Raft. The benchmarks were ran on a 2019 MacBook Pro, with a 1.4GHz Quad-Core Intel Core i5 and 8GB of 2133 MHz DDR3.

Benchmark code, configuration, timing results, and resource usage graphs can be found here [21].

4.2 Results

As expected, between the replication cache and the ECC cache, network bandwith, average transaction time, and CPU usage are similar. Memory usage is almost halved with the ECC cache on write heavy workloads, like workload A and C.

The replication cache has shorter transactions than the ECC cache on average, as seen in figure 2, this is likely due to expensive timing of the two-phase commit protocol. The ECC cache must make at least two network RPCs to the entire cluster, one round for the prepare phase and one round for the commit phase. The replication cache needs one RPC to the Raft leader, and then leader makes RPCs to the entire cluster. The prepare phase of two-phase commit likely takes more time than initiating a new write on a Raft leader, and makes the ECC cache's transactions slower.

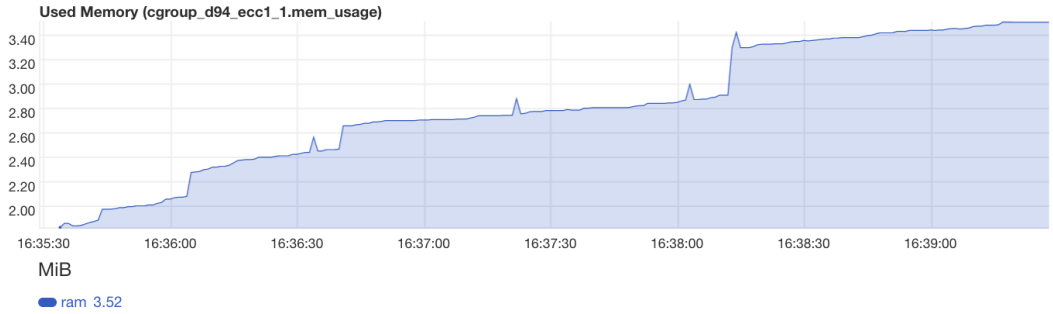
Additionally, the replication cache has slightly higher CPU usage, as seen in figure 2. This may be because the Raft library sends more frequent heartbeats then the ECC cache, or because of Raft tasks like log compaction.

The ECC cache used significantly lower memory in a write heavy workload. We can see in figure 3, the ECC cache used 3.5 MiB and the replication cache used 6.0 MiB to store 10,000 unique entries. Since the ECC cache configuration has 2 data servers and 1 parity server, the ECC cache consumes 150% memory for every value. On the other hand, the replication server stores 3 copies of each value,

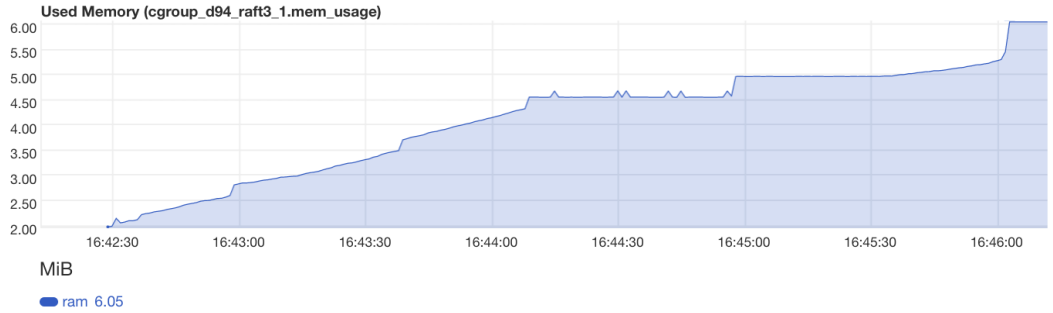
requiring 300% memory for each value. The ECC cache has significant memory savings, especially in write heavy work loads.

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	3.5	9	0.01921
C	Replication	6	10	0.01872

Figure 2: Performance over 10,000 transactions



(a) ECC



(b) Replication

Figure 3: Memory usage of workload c (note differing y-axis scale)

5 Conclusion

Implementing a distributed cache with ECC requires a fault tolerant design, an ECC implementation, and concurrency control. In this report, we chose a design inspired by RAID 4, Reed-Solomon erasure codes and two-phase commit to build a distributed key-value store. A Raft based distributed cache was also implemented to measure performance against a replication cache. Findings include slightly slower transaction times in the ECC cache, more CPU usage in the replication cache and strong memory savings in the ECC Cache.

6 Future Work

6.1 Project Implementation

Overall, the project's lack of testing is a weakness. Additionally, better error handling and logging is needed. Lastly, code review and consolidation would be needed, as this is one of my first Rust programs.

Testing is particularly needed for the ECC cache, especially for edge cases involving node failures and concurrent transactions. I know for simple cases, recovery and two-phase commit are functioning, but the project needs a proper integration test suite to find bugs.

Additionally, the project lacks proper error handling and logging, custom error types and logging libraries should be implemented.

Lastly, a code cleanup should be conducted as this is one of my first Rust programs. I believe there may be inefficient data cloning and poor usage of Rust, and there are cases of duplicate code that can be cleaned up or generalized.

6.2 Replication Cache

A couple improvements can be made to the replication cache. Firstly, the hard state needs to be written to persistent storage. Secondly, there are some bugs with client's getting from dead servers and failing fast, logic is needed to continue looking for servers while none are reached.

6.3 Error Correction Code Cache

The ECC cache could use a faster concurrency control strategy, a better server recovery design, and also edge case tests.

Currently two-phase commit is implemented for concurrency, but I suspect there may be some oversights with the heartbeat mechanism to count the number of locks required for the prepare phase. The ECC cache is unique because the client doesn't know exactly how many servers it needs to write to, only that it is somewhere in $[k, n]$.

It's possible a different concurrency control strategy is a better candidate for the ECC cache. Optimistic concurrency control could be used to increase performance as transactions rarely overlap. Snapshot Isolation could also be used, it might be able to provide sufficient guarantees about client reads.

Server recovery is also a sore spot, since a server in recovery may miss out on new writes. There are two solutions to this, first the entire cluster could block, however this would be undesirable, second the cluster can make progress and the recovering server may give new writes priority (perhaps replacing all recovered entries after recovery). Server recovery needs further exploration and inspiration should be taken by reviewing how RAID reconstructs disks, especially how transactions are allowed during recovery.

The ECC cache also needs to be tested thoroughly, there are likely overlooked edge cases that are easy to miss in a distributed system without a thorough test suite.

Lastly, a weird quirk with the Dockerfile occurs when built using a Github Actions pipeline, the container becomes unstable and crashes quickly. The image built from my Macbook is stable and the ECC cache can handle recovery without crashing. This needs to be investigated.

6.4 Scaling

Scaling either cache remains unaddressed. Adding ranges to keys is an easy solution, but then further logic needs to implemented to rebalance current nodes to new ranges. Additionally, it may be possible to scale the ECC cache in a live scenario, again inspiration from RAID could be drawn here.

6.5 Benchmarking

Visualization was done with Netdata, which presented difficulty when graphing failed nodes. Dead containers disappear from Netdata, so human intervention is needed to get graphs from containers before they disappear from Netdata. Additionally, custom graphs would create a better view of the cache as a whole, with all ECC nodes on the same graph using individual lines. CGroups could be used to collect statistics to create custom graphs here.

The benchmark script should consider multithreading, to test out performance of concurrent transactions.

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/brianfrankcooper/YCSB.

[21] https://github.com/felixjchen/Distributed-Cache/tree/main/report/benchmarks.