JEPSEN

Redis-Raft 1b3fbf6

Kyle Kingsbury
2020-06-23

Redis is a popular in-memory data structure server. Historically, Redis has supported a number of ad-hoc replication mechanisms, but none guaranteed stronger than causal consistency. Redis-Raft aims to bring strict serializability to Redis by means of the Raft consensus algorithm. We found twenty-one issues in development builds of Redis-Raft, including partial unavailability in healthy clusters, crashes, infinite loops on any request, stale reads, aborted reads, split brain leading to lost updates, and total data loss on any failover. All but one issue (a crash due to assertion failure around snapshots) appear addressed in recent development builds. This work was funded by Redis Labs and conducted in accordance with the Jepsen ethics policy.

1 Background

Redis is a fast single-threaded data structure server, commonly used as a cache, scratchpad, queue, or coordination mechanism between distributed applications. It offers operations over a broad array of datatypes, including binary blobs, lists, sets, sorted sets, maps, geohashes, counters, channels, streams, and more. In recent years, an increasing cohort of production users have deployed Redis as a system of record, spurring an increased focus on safety and reliability.

In addition to individual operations, Redis supports Lua scripts, and a transaction mechanism called `MULTI` which allows clients to group together operations into an atomically executed batch. `MULTI` does not provide interactive transactions; the results of operations are only realized after the transaction is committed. The `WATCH` command allows transactions to check whether a key has remained unmodified since its last read—an optimistic concurrency control primitive.

1.1 Existing Replication Mechanisms

Redis offers several replication mechanisms, each with distinct tradeoffs. Redis's initial replication mechanism sent updates asynchronously from primary to secondary nodes—secondaries overwrote their state with whatever the primary happened to have at that point in time. Failover was performed by hand, or via third-party watchdogs like Pacemaker.

Redis Sentinel, introduced in 2012, allowed nodes to automatically select new primaries with the help of external processes executing a homegrown fault-detection and leader-election algorithm. Sentinel lost data during network partitions, and continues to do so as of this writing:

> In general Redis + Sentinel as a whole are a an [sic] eventually consistent system where the merge function is last failover wins, and the data from old masters are discarded to replicate the data of the current master, so there is always a window for losing acknowledged writes.

A second replication strategy, Redis Cluster, provides transparent sharding and majority availability. Like Redis Sentinel, it uses asynchronous replication, and can lose acknowledged writes during some types of network failures:

> Usually there are small windows where acknowledged writes can be lost. Windows to lose acknowledged writes are larger when clients are in a minority partition.

Redis Enterprise, a commercial offering from Redis Labs, includes a third replication strategy called "Active-Active Geo-Distribution", which is based on Conflict-free Replicated Data Types (CRDTs). Sets use observed-removed sets, counters use a novel resettable counter implementation, and maps merge updates on a key-wise basis. Some datatypes, like strings and the values of maps, are resolved using last-write-wins, which is subject to lost updates.

Redis Sentinel, Redis Cluster, and Active-Active Geo-Distribution all allow lost updates—at least for some workloads. To mitigate this risk, Redis includes a `WAIT` command, which ensures prior writes are "durable even if a node catches on fire and never comes back to the cluster". Moreover, Redis Enterprise claims to offer "full ACID compliance with its support for `MULTI`, `EXEC`, `WAIT`, `DISCARD`, and `WATCH` commands."

So, does Redis Enterprise lose updates? Or is it "full ACID"? The Redis-Raft documentation casts doubt on ACID claims,[1] noting that `WAIT` "does not make the system strongly consistent overall". In discussions with Jepsen, Redis Labs clarified that Redis Enterprise *can* offer ACID characteristics, but only 1.) without any form of replication, 2.) the write-ahead log must be set to `fsync` on every write, and 3.) there is no way to roll back when transactions fail. These factors are not clearly documented, but Redis Labs plans to document them in the future.

In short, users who want fault-tolerance and *not* lost updates need something stronger than existing Redis replication systems. Whereupon: Redis-Raft.

1.2 Redis-Raft

The fourth Redis replication mechanism, and the focus of the present work, is Redis-Raft, which uses the Raft consensus algorithm to replicate Redis's state machine across a set of nodes.

Redis-Raft claims to make Redis "effectively a CP system". Putting all operations through the Raft log should allow operations to be linearizable. Since operations on different keys go through the same Raft state machine, and since `MULTI` transactions are implemented as a single Raft operation, Redis-Raft should also offer strict serializability—both for individual operations and for transactions.

Redis-Raft started as a proof-of-concept in February 2018, and Redis Labs has been working towards a production release since mid-2019. During our collaboration, Redis-Raft was unavailable to the public, but Redis Labs plans to make the source available at Redisconf 20, and aims for a general-availability release as a part of Redis 7.0.

2 Test Design

We designed a test suite for Redis-Raft using the Jepsen testing library. Since Redis-Raft relies on features in the unstable branch of Redis, we ran our tests against Redis f88f866 and 6.0.3, and development builds of Redis-Raft from 1b3fbf6 through e0123a9. All tests were run on five-node Debian 9 clusters, on both LXC and EC2. We introduced a number of faults during our testing process, including process pauses, crashes, network partitions, clock skew, and membership changes.

Prior Jepsen tests have relied on a broad variety of workloads, each designed to detect different anomalies, or to compensate for performance limitations in other workloads. Over the last year, Jepsen collaborated with UC Santa Cruz' Peter Alvaro to design a new type of consistency checker, which operates in linear (rather than exponential) time, over a broad range of transactions and single-key operations, which verifies a wide range of safety properties up to strict serializability, and which provides understandable, localized counterexamples for safety property violated. We call this checker Elle.

We used Elle exclusively in this analysis, measuring Redis's safety with respect to transactions over lists. Each transaction (or singleton operation) is comprised of read and append operations over a small, evolving set of keys. Reads return the current state of the list using `LRANGE`, and appends add a distinct element to the end of the list using `RPUSH`. Elle infers dependency relationships between these transactions, including write-write, write-read, and read-write data dependencies, as well as realtime orders, and looks for cycles in that dependency graph as evidence of strict-serializability violations.

3 Results

We identified twenty-one issues in Redis-Raft, ranging from transient unavailability, to behavior that could make it difficult to write correct client programs, to complete data loss.

3.1 Infinite Loops

Raft is a leader-based protocol: requests cannot be executed by followers, but must be sent to a leader instead. Redis clients can follow redirects to submit their operations directly to a (hopefully current) leader, or, with the `follower-proxy=yes` option, Redis followers can proxy requests to a leader on the client's behalf.

With this proxy mode enabled, we found that executing *any* write against Redis-Raft 1b3fbf6 sent the cluster into an infinite loop: the operation would be applied to the log over and over again, ballooning the Raft log

[1] Redis-Raft documentation was not, as of this writing, available to the public.

and (depending on the write) Redis's in-memory and on-disk state.

This problem (#13) was caused by a missing reentrancy check. Redis-Raft works by intercepting client commands (e.g. `SET key val`) and re-writing them to a special Raft command (`RAFT SET key val`). That Raft command is then replicated through the Raft log, and, once committed, unwrapped (producing `SET key val`) and applied to the local state machine. However, the interception code would then identify that command as one that needed to be sent to the Raft log, wrap it in a `RAFT` command again, and send it *back* through the consensus system. Adding a reentrancy check to the interception logic resolved the issue, in version d589127.

3.2 Total Data Loss on Failover

When we evaluated Redis-Raft 1b3fbf6 *without* follower proxies, we found that any failover would cause the loss of all committed data. Newly elected leaders would come online with a completely fresh state. This problem was trivially reproducible at the CLI, as well as in append tests.

This issue (#14) was caused by the same missing reentrancy check as #13. When a leader processed an operation, it applied it to its local state machine. Followers, however, would intercept the operation, transform it to a `RAFT` operation, and then (proxy mode being disabled) reject the operation because they weren't the leader.

Like #13, this issue was fixed by d589127.

3.3 Split-Brain & Lost Updates

The Raft paper includes an algorithm for performing online membership changes. When nodes are added to or removed from the cluster, Raft enters a special *joint consensus* mode, in which a majority of the original members *and* a majority of the new members must agree on each operation before commit. Once a majority of the original cluster have acknowledged the new membership, the cluster resumes normal operation, requiring acknowledgement only from a majority of the new members.

Redis-Raft d589127 did not correctly implement this system. Leaders could execute membership changes independently, without getting confirmation from any other node. A leader could be isolated by a network partition, process pause, or crash, successfully remove every other node in the cluster, declare itself the sole leader of the resulting single-node cluster, and proceed to execute arbitrary operations on its own. Given n nodes and a sufficiently pathological operator, Redis-Raft could split into n separate clusters, each diverging from some common prefix of the original cluster's history.

This issue (#17) was caused by a bug in the underlying Raft library: `RAFT_LOGTYPE_REMOVE_NODE` was left out of the set of log entry types considered to be voting configuration changes. Version 8da0c77 resolved the problem.

3.4 Transient Empty Reads on Startup

In version d589127, we found that killing and restarting nodes led to a short window of time following node startup where that node could return the empty state for a read, rather than committed state. This error was transient: a few seconds later, reads would observe expected values again.

For instance, a client might execute a transaction which appends 89 to key 0, and reads the resulting list:

- T_1: `[:append 0 89] [:r 0 [...86 87 89]]`

Then, following a process start, a read of key 0 on the freshly started node would return no elements:

- T_2: [:r 0 []]

The Redis team tracked this problem (#18) to a bug in the underlying Raft library. Whenever a new leader is elected, that leader should issue a no-op log entry to its followers, in order to establish what current state is committed. This behavior was missing from the Raft library packaged with Redis-Raft. Pulling a newer version helped resolve the issue, and Redis-Raft dfd91d4 no longer exhibited this behavior.

3.5 Stale Reads in Healthy Clusters

Versions d589127 and 8da0c77 also exhibited stale reads in normal operation, without any faults. For instance, consider this pair of transactions, where T_1 completed 3.25 seconds before T_2 began:

- T_1: `[:append 1 11]`
- T_2: `[:r 1 [5 8 9]]`

If Redis-Raft were linearizable, then T_1's append of 11 to key 1 should have been *visible* in T_2's read—but instead, T_2 observed only `[5 8 9]`. This is a stale read: a view into the past.

Like #18, this issue (#19) hinged on the failure of leaders to issue a no-op operation upon coming to power; it was also fixed in dfd91d4.

3.6 Spurious NOLEADER in Healthy Clusters

In versions 8da0c77 through 73ad833, we found that healthy clusters tended to experience partial outages for no apparent reason—with sub-millisecond network latencies and without fault injection, some nodes would return `NOLEADER` for hundreds of seconds, then recover as if nothing had happened. Occasionally, nodes would begin timing out new connections, rather than returning `NOLEADER`.

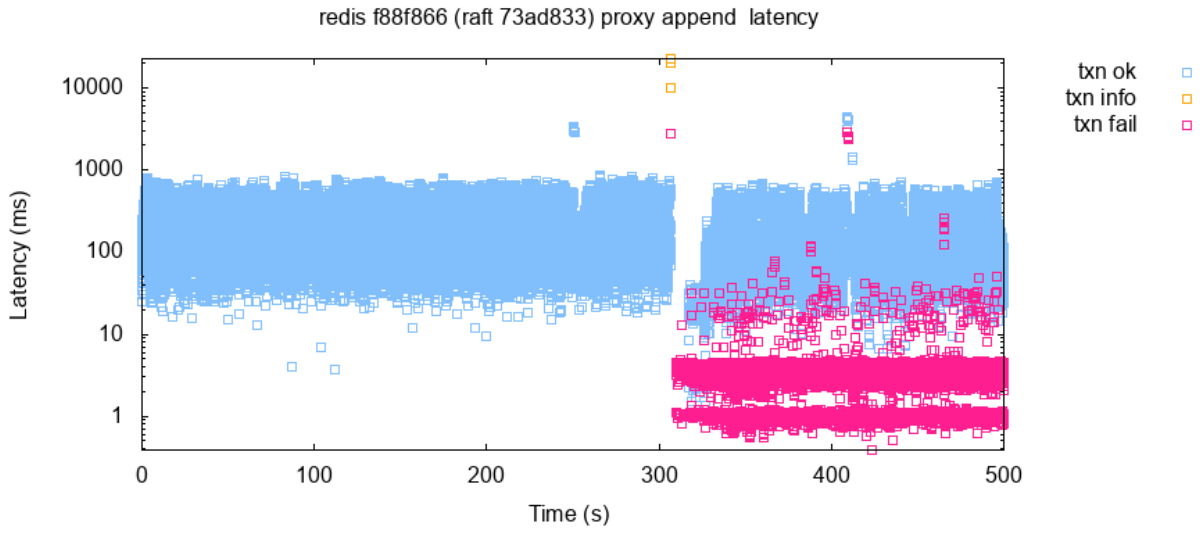

Figure 1: A plot of request latencies over time, with color indicating which nodes had failed. Just after 300 seconds, one node began returning spurious NOLEADER responses to every request.

Redis Labs traced this problem (#21 to a head-of-line blocking problem where when a new node became a leader, proxied commands from followers could delay (possibly indefinitely) Raft messages, which caused Raft operations to stall on some nodes following an election. This problem was exacerbated by aggressive default timeouts, which caused elections to occur frequently in otherwise healthy clusters.

To fix this issue, Redis Labs added a timeout mechanism to proxied commands, and adjusted the default timeouts to reduce spurious elections due to normal variability in node response times. These patches were applied in 6fca76c. As of b9ee410, Redis-Raft exhibits occasional `NOLEADER` hiccups every few minutes, but they resolve within seconds.

3.7 Aborted Reads With NOLEADER

In version dfd91d4, we observed what appeared to be aborted reads involving network partitions and process crashes. Operations would fail with the `NOLEADER` error code, but their effects would still be visible to later transactions. For instance, take this pair of transactions from an append test run:

- T_1: `[:append 295 223]`
- T_2: `... [:r 295 [223 228 229 233]] ...`

Here, T_1 failed with `NOLEADER`, but T_2 was able to observe T_1's write. Redis Labs' engineers confirmed that `NOLEADER` indicates an operation definitely failed, which means this pair of transactions constitutes an aborted read.

This issue (#23) was addressed by a package of protocol-level fixes in version e657423; we have not observed it since.

3.8 DISCARD Doesn't Always Discard

In Redis, one begins a transaction by sending `MULTI`, a sequence of commands, and a final `EXEC` to commit, or `DISCARD` to abort. This makes it straightforward for Redis clients to offer some sort of transactional flow control context, e.g. using exception handlers:

```
try {
  conn.multi();
  conn.put(k1, v1);
  conn.put(k2, v2);
  r = client.exec();
} catch Exception e {
```

```
  conn.discard();
  throw e;
}
```

This code is simple, and works correctly some of the time. However, in Redis-Raft, calls to DISCARD can (and often do!) fail, e.g. for NOLEADER, NOTLEADER, etc. This leaves the connection in the MULTI state, with operations buffered by Redis. Subsequent calls will execute in the previous MULTI context, which could lead to confusing results: transactions could be mixed together with unanticipated effects, return values from EXEC could be those for completely different operations, etc. Using MULTI correctly requires careful attention to tracking connection state.

This problem (#25) occurred because Redis-Raft performs cluster state checks on each command individually, rather than buffering commands on the local server and submitting the entire batch on EXEC. This problem was addressed in version f4bb49f by moving the MULTI state machine into the local node, allowing it to buffer operations in memory and commit them in an atomic batch.

3.9 Crossed Wires

With version dfd91d4, we found that network partitions and process crashes could lead Redis to reply to queries with answers for *different* queries. For example, a client could execute a single read…

```
LRANGE 33 0 -1
```

… And get back a MULTI reply intended for some other client altogether:

```
[3 4 ["2" "4" "21" "22"]]
```

This is a reply to a MULTI...EXEC transaction which performed two appends (resulting in lists of length 3 and 4, respectively) followed by a list read (which returned 2, 4, …). Notably, this response has nothing to do with the client's request, nor any other request this client made—in this particular case, the client performed the LRANGE request with a fresh connection. Like #25, this could lead to type errors or silent data corruption, depending on whether the response types happened to match those expected by the requester.

This issue (#26) involved multiple bugs in Redis-Raft's proxy mechanism. In the Redis command handler, replies for proxied commands were accidentally routed to the Redis-Raft context, rather than the request context. Leaders needed to apply pre-bundled MULTI transactions immediately, rather than re-bundling them. Redis Labs also added more defensive error handling to asynchronous context cleanups.

Version e657423, which includes all these fixes, appears to have resolved the issue.

3.10 Split-Brain & Lost Update Redux

In version f88f866, we found another case of split-brain, this time involving membership changes and process crashes. Two nodes could diverge from a common history, or even apply the same operation to *different* local states. Consider, for example, these reads of key 81: transactions T_2 and T_4, executing on node n1, observe lists starting with 171 and 172, whereas T_1 and T_3, executing on node n5, begin with 176 instead.

- T_1: `[:r 81 [176 177 178]]`
- T_2: `[:r 81 [171 172 176 177 178]]`
- T_3: `[:r 81 [176 177 178 208]]`
- T_4: `[:r 81 [171 172 176 177 178 208]]`

Note that appends of 178 and 208 are applied to *both* sides of the split-brain, on top of different prefixes!

Redis Labs traced this issue (#28) to the cluster membership system. The underlying Raft library assumed that nodes would be demoted *then* removed from the cluster, rather than removed directly. There was also an issue in which nodes simply stepped down after being removed from the cluster, leaving their data files in place. Re-joining a removed node to the cluster could cause it to violate safety invariants. Redis Labs addressed this issue by forcing removed nodes to archive their local state before termination. As of version bc9552f, Redis-Raft no longer exhibited split-brain with membership changes.

3.11 Yet More Transient Empty Reads

In version bc9552f, we found yet another case of empty reads after startup, triggered by a combination of membership changes and process crashes. If a node started up after membership changes, it could temporarily return empty values instead of committed data. For example…

- T_1: `[[:r 37 [5 9 13 1 19 20]]]`
- T_2: `[[:r 37 []]]`
- T_3: `[[:r 37 [5 9 13 1 19 20]] ...`

Here, nodes restarted just prior to T_2. This problem was relatively infrequent—it required several hours of randomized fault injection to discover, and upwards of five minutes to reproduce with targeted faults.

Redis Labs traced this issue (#30) to two bugs in the log-loading process on startup. First, when a node's log indicated that only a single (voting) node existed,

a special case in the log loading code should have allowed Redis-Raft to treat its log as fully committed, but this codepath counted *all* nodes, rather than *voting* nodes. Second, during log loading, Redis-Raft incorrectly treated nodes which were *previously* part of the cluster as if they were still active, and could exchange messages with them. Nodes could even send messages to themselves. Both of these problems were addressed in 73ad833.

3.12 Spontaneous Snapshot Crash

With Redis-Raft b9ee410, we observed that nodes would occasionally crash under normal operation, citing a failed assertion in `callRaftPeriodic`. This issue (#42) appeared linked to an unexpected return value from `pollSnapshotStatus`. Redis Labs is investigating.

3.13 Panic! At The Raft Log

In b9ee410, we found that with process crashes, pauses, partitions, and membership changes, nodes frequently wound up with unrecoverable on-disk states where their snapshot file was taken from a log index significantly *before* the first entry in the log. Any attempt to start the node would panic, logging something like `Log initial index (1478) does not match snapshot last index (1402), aborting.` Multiple nodes encountered the problem

We did not have time to narrow down the conditions under which this bug (#43) occurs, and were unable to identify a cause. However, this issue no longer appeared in e0123a9.

3.14 Split Brain, Take III

Version b9ee410 exhibited yet another case of split-brain, which we observed repeatedly during tests with process crashes, pauses, partitions, and membership changes. As with previous split-brain issues, some nodes in the cluster would diverge from a shared prefix of the history, allowing reads and updates to proceed independently. Updates could be lost, depending on which branch(es) of the history survived.

This behavior (#44) appears to stem from a bug in the snapshot-loading process on node startup: the loader failed to mark the loading process as complete, and did not update the snapshot metadata in memory. The resulting configuration was usable by the running process, but taking a *new* snapshot from that state resulted in nodes missing from the snapshot cluster state! This bug appeared fixed as of 2d1cf30.

3.15 Crash With `raft_node_is_voting(me_)`

In version 2d1cf30, tests with process crashes, pauses, partitions, and membership changes revealed a crash due to an assertion failure when updating the local raft node's voting state. Restarting the node recovered the process, and no safety impact was detected.

Like #42, #43, and #44, this bug (#48) arose late in our testing process; we have little information about its cause, or what circumstances are necessary to trigger it. No cause was ascertained, but we believe this issue was resolved by e0123a9.

3.16 Crash With `!uv__is_closing(handle)`

Also in version 2d1cf30, and under similar conditions to #48, we found another crash (#49) in libuv's unix/poll.c. Like #48, this crash seemed recoverable, and there was no observable safety impact. No cause was ascertained, but we believe this crash was resolved by e0123a9.

3.17 Mysterious Segfault

We found yet another crash in 2d1cf30, where nodes occasionally crashed during tests with crashes, pauses, partitions, and membership changes. The error message provided little to go on, other than citing a `SIGSEGV` crash, a blank address, no assertion, no file, no line number, and no stacktrace.

We observed no safety impact associated with this issue (#50), and the node recovered when restarted. No cause was ascertained, but we believe this crash was resolved by e0123a9.

3.18 Crash in EntryCacheAppend

In 2d1cf30, process pauses, or crashes, or network partitions, could each independently trigger an assertion crash: `EntryCacheAppend: Assertion 'cache->start_idx + cache->len == idx'`. This crash (#51) did not seem associated with any safety impact. We were unable to ascertain the cause of this issue, but it did not appear in e0123a9.

3.19 Rewriting History

Again in 2d1cf30, we found that process crashes alone were sufficient to cause Redis-Raft nodes to disagree over history: nodes could delete or duplicate operations, even operations well in the past which had been superceded by dozens of committed writes! Nodes could flip back and forth between various versions of the same history.

For instance, consider this test run, where a process appends 25 to key 297, 25 is visible to reads against three nodes, and then, after a process is killed and restarted, element 25 is retroactively erased from the history. We present just the observed values of key 297, and elide some elements for clarity:

```
[:r 297 [1 3 2 ... 26 25]]
[:r 297 [1 3 2 ... 26 25 27]]
[:r 297 [1 3 2 ... 26 25 27 ... 47]]
[:r 297 [1 3 2 ... 26 25 27 ... 48]]
[:r 297 [1 3 2 ... 26 27 29 ... 106]]
[:r 297 [1 3 2 ... 26 27 29 ... 120]]
```

We suspect this "time travel" revision of key 297's history indicates a violation of the Raft log agreement invariant, but because this issue (#52]) arose late in our testing process, we are unsure of its cause. It appeared fixed as of e0123a9.

3.20 `AppendEntries` Term Mismatch

2d1cf30, ever the font of new and exciting discoveries, offered up a new crash (#53) during process crashes and network partitions. In response to the Raft `AppendEntries` RPC call, nodes could crash, complaining that their previous term in the log didn't match that of the `AppendEntries` message.

We were unable to ascertain the cause of this issue (#53, but it appeared resolved as of e0123a9.

```
raft.c:479: <raftlib> AE term doesn't match prev_term (ie. 177 vs 184)
ci:12673 comi:12577 lcomi:12637 pli:12577
raft.c:479: <raftlib> AE prev conflicts with committed entry
redis-server: raft.c:784: callRaftPeriodic: Assertion `ret == 0' failed.
```

3.21 Another Snapshot Crash

Surprise! 2d1cf30 contained yet another crash which arose during testing with process kills. When writing snapshots, the snapshot process could could segfault during the log-rewriting process. While the main Redis-Raft process continued running, the snapshot process would log a segfault in `RaftLogWriteEntry`. This issue (#54) co-occurred with duplicate elements and incompatible reads, as reported earlier—the two might be related. We were unable to ascertain a cause for this issue, but it appeared resolved as of e0123a9.

№	Summary	Event Required	Fixed In
13	Infinite loops with follower-proxy	None	d589127
14	Total data loss	Failover	d589127
17	Split-brain & lost updates	Partition & membership change	8da0c77
18	Transient empty reads	Node startup	dfd91d4
19	Stale reads	None	dfd91d4
21	Partial failure returning NOLEADER	None	6fca76c
23	Aborted reads with NOLEADER	Partition & crash	e657423
25	DISCARD doesn't always discard	None	f4bb49f
26	Crossed wires with follower-proxy	Partition & crash	e657423
28	Split-brain & lost updates	Crash & membership change	bc9552f
30	More transient empty reads	Crash & membership change	73ad833
42	Crash with callRaftPeriodic: `ret == 0`	None	Unresolved
43	Panic on startup due to Raft log index mismatch	Partition, crash, pause & membership?	e0123a9
44	Split-brain & lost updates	Partition, crash, pause & membership?	2d1cf30
48	Crash with `raft_node_is_voting(me_)`	Partition, crash, pause & membership?	e0123a9
49	Crash with `!uv__is_closing(handle)`	Partition, crash, pause & membership?	e0123a9
50	Mystery Segfault	Partition, crash, pause & membership?	e0123a9

51	Crash in `EntryCacheAppend`	Pause	e0123a9
52	Rewriting history, write loss, duplicate elements	Crash	e0123a9
53	`AppendEntries` term mismatch	Partition & crash	e0123a9
54	Snapshot crash in `RaftLogWriteEntry`	Crash	e0123a9

4 Discussion

Redis-Raft is an unreleased project under development; we expect to find bugs at this stage. Indeed, we found twenty-one issues, including long-lasting unavailability in healthy clusters, eight crashes, three cases of stale reads, one case of aborted reads, five bugs resulting in the loss of committed updates, one infinite loop, and two cases where logically corrupt responses could be sent to clients. The first version we tested (1b3fbf6) was essentially unusable: depending on the choice of `follower-proxy`, it either entered an infinite loop on receiving any write, or lost all data on any failover.

We emphasize, again, that these are all internal development builds: Redis-Raft has no production users, so the real-world impact of these issues is negligible.

As of version e0123a9, all but one issue (#42) we observed in previous builds appeared resolved. We should note that we have not tested e0123a9 as extensively as other builds—it may contain additional bugs.

As always, we note that Jepsen takes an experimental approach to safety verification: we can prove the presence of bugs, but not their absence. While we try hard to find problems, we cannot prove the correctness of any distributed system.

Tangentially, we were surprised to discover that Redis Enterprise's claim of "full ACID compliance" only applied to specially-configured non-redundant deployments, rather than replicated clusters. While Jepsen has not experimentally confirmed data loss in Redis Enterprise, our discussions with Redis Labs suggest that users should not, in general, assume that multi-node Redis deployments offer ACID guarantees. We agree with Redis Labs that the documentation should make this clear.

4.1 Future Work

Redis Labs plans to continue development of Redis-Raft, with an aim towards general availability in 2021.

We designed only a single workload for Redis-Raft: the append test. This test is powerful, efficient, general, and captures a broad range of safety criteria. However, it is limited to evaluating just two Redis data commands: `LRANGE` and `RPUSH`. We believe this approach should still give good coverage, since Redis-Raft treats many commands alike. However, there could be other behaviors given different commands. In future work, we believe it be prudent to explore other types of operations: `GET` and `SET`, perhaps, or operations on sets.

Jepsen's membership tests are fragile, partly due to the asynchronous nature of Redis's node add/remove commands. Under certain circumstances, it appears that Jepsen could delete the data files for running nodes when performing membership tests. While this does not appear to have impacted our findings, it deserves revisiting, especially once Redis-Raft supports synchronous node removal.

We have not explored single-node faults, such as filesystem corruption or the loss of un-fsynced data written to disk. Both might be of interest for Redis-Raft, whose correctness hinges (like most consensus systems) on single-node durability.

Our work in this analysis was limited to Redis-Raft—Jepsen has not recently evaluated Redis Sentinel, and has never evaluated Redis Cluster, Active-Active Geo-Distribution, or Redis Enterprise behavior. It might be interesting to compare these systems: each has documented drawbacks, but we have no qualitative view of how serious these problems might be, or what undocumented behaviors might be present.

We look forward to Redis-Raft's eventual release.

This work was funded by Redis Labs, and conducted in accordance with the Jepsen ethics policy. Jepsen wishes to thank the Redis team for their invaluable assistance—especially Yossi Gottlieb and Yiftach Shoolman. Our thanks as well to Coda Hale for his review.