JEPSEN

etcd 3.4.3

Kyle Kingsbury
2020-01-30

The etcd key-value store is a distributed database based on the Raft consensus algorithm. In our 2014 analysis, we found that etcd 0.4.1 exhibited stale reads by default. We returned to etcd, now at version 3.4.3, to investigate its safety properties in detail. We found that key-value operations appear to be strict serializable, and that watches deliver every change to a key in order. However, etcd locks are fundamentally unsafe, and those risks were exacerbated by a bug which failed to check lease validity after waiting for a lock. The etcd developers have written a companion blog post to this report. This work was funded by The Cloud Native Computing Foundation, which is part of The Linux Foundation, and was conducted in accordance with the Jepsen ethics policy.

1 Background

The etcd key-value store is a distributed system intended for use as a coordination primitive. Like Zookeeper and Consul, etcd stores a small volume of infrequently-updated state (by default, up to 8 GB) in a key-value map, and offers strict-serializable reads, writes and micro-transactions across the entire datastore, plus coordination primitives like locks, watches, and leader election. Many distributed systems, such as Kubernetes and OpenStack, use etcd to store cluster metadata, to coordinate consistent views over data, to choose leaders, and so on.

When we evaluated etcd 0.4.1 in 2014, we found that it exhibited stale reads by default due to an optimization. While the Raft paper discusses the need to thread reads through the consensus system to ensure liveness, etcd performed reads on any leader, locally, without checking to see whether a newer leader could have more recent state. The etcd team implemented an optional quorum flag, and in version 3.0 of the etcd API, made linearizability the default for all operations except for watches.

The etcd 3.0 API centers on a flat map of keys to values, where both keys and values are opaque byte arrays. Hierarchical keys may be simulated with range queries. Users may read, write, and delete keys, or watch for a stream of updates to single or ranges of keys. Leases (transient objects with a limited lifetime, kept alive via client heartbeats), locks (exclusively held named objects, bound to leases), and leader elections round out the etcd toolkit.

In 3.0, etcd offers a limited transaction API for atomic multi-key operations. A transaction, in this model, is a single conditional expression with a predicate, a true branch, and a false branch. The predicate may be the conjunction of several per-key comparisons: equality or various inequalities, over a single key's version, the global etcd revision, or the key's current value. Both true and false branches may include multiple read and write operations, all of which are applied atomically, depending on the result of evaluating the predicate.

1.1 Consistency Documentation

As of October 2019, etcd's API guarantees documentation claimed that “all API calls exhibit sequential consistency, the strongest consistency guarantee available from distributed systems.” This is incorrect: sequential consistency is strictly weaker than linearizability, and linearizability is definitely achievable in distributed systems. The documentation goes on to claim that “etcd does not guarantee that it will return to a read the ‘most recent’ value (as measured by a wall clock when a request is completed) available on any cluster member.” This is also an overly conservative statement: if etcd provides linearizability, reads always observe the most recently committed state in the linearization order.

The documentation also claims that etcd guarantees serializable isolation: all operations, even those involving multiple keys, appear to take place in some total order. The documentation characterizes serializable isolation as "the strongest isolation level available in distributed systems". This is (depending on how one defines "isolation level"), not true either; strict serializability is stronger than serializability, and strict serializability is also achievable in distributed systems.

The documentation states that all operations (except for watches) in etcd are linearizable by default. The documentation defines linearizability as conformance with a loosely synchronized global clock. We note that this definition is not only incompatible with Herlihy & Wing's definition of linearizability, but also implies causality violations; nodes with faster clocks would be required to read the results of operations that hadn't even begun yet. We assume that etcd is not a time machine, and that as an implementation of Raft, it offers the commonly accepted definition of linearizability instead.

Since key-value operations in etcd are serializable and linearizable, we believe etcd is in fact strict serializable by default. This makes sense, because all etcd keys reside within a single state machine, and all operations on that state machine are totally ordered via Raft. In essence, the entire etcd dataset is one linearizable object.

An optional `serializable` flag *downgrades* reads from strictly serializable to serializable consistency by allowing reads of stale committed state. Note that the `serializable` flag has no impact on whether or not a history is serializable; etcd key-value operations are always serializable.

2 Test Design

We designed a test suite for etcd using the Jepsen testing library. We evaluated etcd version 3.4.3 (the latest release as of October 2019) running on five-node Debian Stretch clusters. We introduced a number of faults into these clusters, including network partitions isolating single nodes, separating the cluster into majority and minority components, and non-transitive partitions with overlapping majorities. We crashed and paused random subsets of nodes, as well as specifically targeting leaders for failure. We also introduced clock skew up to hundreds of seconds, both for multi-second intervals, and strobing rapidly over milliseconds. Since etcd supports dynamic membership changes, we randomly added and removed nodes during testing.

Our test workloads included registers, sets, and transactional tests to verify key-value operations, as well as specialized workloads for locks and watches.

2.1 Registers

To evaluate etcd's safety for key-value operations, we designed a register test, which performs randomized reads, writes, and compare-and-set operations over single keys. We evaluated those histories with the Knossos linearizability checker, using a model of a compare-and-set register, plus versioning information.

2.2 Sets

As a quantitative measure of stale reads, we designed a set test, which used a compare-and-set transaction to read a set of integers from a single key and append a value to that set. We concurrently read the set throughout the test. At the end of the test, we looked for cases where an element we knew *should* have been in the set failed to appear in reads, and used those cases to quantitatively measure stale reads and lost updates.

2.3 Append

To verify strict serializability, we designed an append test, where transactions concurrently read and appended to lists of unique integers. We stored each list in a single etcd key, and performed each transaction's appends by reading every key to be modified in one transaction, then writing those keys and performing any reads in a second transaction, which was guarded to ensure that no written keys had changed since the first read. At the end of the test, we constructed a dependency graph between transactions on the basis of real-time precedence, and the relationships between reads and appends. Checking that graph for cycles allowed us to determine whether the history was strict serializable.

While etcd prohibits transactions from writing the same key multiple times, we could issue transactions with up to one write per key. We also checked to ensure that reads within a transaction reflected previous writes from the same transaction.

2.4 Locks

As a coordination service, etcd touts out-of-the-box support for distributed locking. We evaluated these locks in two ways. First, we generated randomized lock and unlock requests, acquiring a lease for each lock acquisition and holding it open using the Java etcd client's keepalive feature until release. We checked these histories using Knossos to see whether they formed a linearizable implementation of a lock service.

For a more practical test (and to gain a more quantitative view of how often locks might fail), we used etcd locks to provide mutual exclusion for updates to an in-memory set, and looked for lost updates to that set. This test allowed us to directly confirm whether systems which used etcd as a mutex could update external state safely.

A third variant of the lock test used guards on the lease key to modify a set stored in etcd.

2.5 Watches

To verify that watches provide every update to a key in order, our watch test created a single key and blindly set it to unique integer values over the course of the test. Meanwhile, clients would concurrently watch that key for a few seconds at a time. Each time a client initiated a watch, it would pick up at the revision where it last left off.

At the end of this process, we verified that every client observed the exact same sequence of updates to the key.

3 Results

3.1 Watching With Revision 0

When watching a key, clients can specify a start revision, which is "an optional revision for where to inclusively begin watching". If a user wishes to observe every operation on some key, they could pass the first etcd revision. What is the first revision? The data model and glossary didn't say; revisions are characterized as monotonically incrementing 64-bit counters, but it's not clear whether etcd begins at 0 or 1. A reasonable user might assume 0, just to be safe.

This is, apparently, not correct. Asking for revision 0 causes etcd to stream updates beginning with *whatever revision the server has now, plus one*, rather than the first revision. Asking for revision 1 yields all changes. This behavior was not documented.

In practice, we think this is relatively unlikely to cause issues for production users: most clusters spend little time at revision 1, and etcd is designed to compact history over time, which means that real-world applications probably aren't relying on reading every version from revision 1 *anyway*. This behavior is justifiable, but would be less surprising if documented.

3.2 Locks Aren't Real

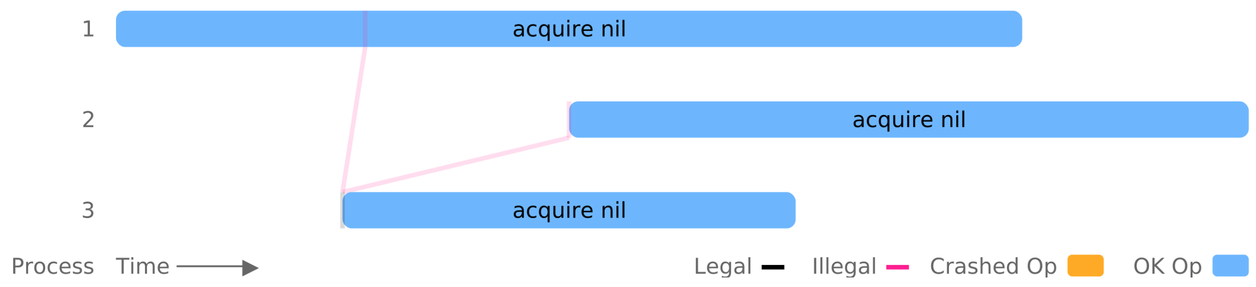
The API documentation for locks states that a locked key "can be used in conjunction with transactions to safely ensure updates to etcd only occur while holding lock ownership," but strangely does not describe any guarantees, or the intended purpose, for locks themselves.

However, other writing from the etcd maintainers *does* tell us how locks are intended to be used. For instance, the etcd 3.2 release announcement demonstrates the use of `etcdctl` to lock concurrent updates to a file on disk. A GitHub issue asking what the exact use case is for locks resulted in this response from an etcd maintainer:

> My understanding is that etcd lock is a service that can be used by users (or other systems) for protecting access to whatever resource they wanted to protect (not necessarily any resource in etcd database), something like:
>
> 1. acquire an etcd lock
> 2. do something (again, not necessarily related to etcd database)
> 3. unlock the same etcd lock

This is exactly what the etcdctl documentation showed as an example: an etcd lock was used to protect an etcd `put` command, but did not couple the lock key to the update.

Unfortunately, this is unsafe, because multiple clients may hold the same etcd lock simultaneously. While this problem is exacerbated by process pauses, crashes, or network partitions, it can also occur in healthy clusters, without any external faults. For instance, in this short test run, process 3 successfully acquires a lock, and process 1 concurrently acquires that same lock before process 3 can release it:



Mutex violation was worst with short lease TTLs: 1, 2, and 3-second TTLs generally failed to provide mutual exclusion after only a few minutes of testing, even in healthy clusters. Process pauses and network partitions created problems faster.

In a variant of our lock test, we used etcd mutexes to protect concurrent updates to a set of integers, just as the etcd documentation suggested. Each update read the current value of an in-memory collection, and, approximately one second later, wrote back that same collection plus a unique element. With two-second leases TTLs, five concurrent processes, and process pauses every five seconds, we could reliably induce the loss of ~18% of acknowledged updates.

This problem was exacerbated by the way in which etcd acquired locks internally. If a client waited for another client to release a lock, lost its lease, and then the lock was released, the server would not re-check to make sure the lease was still valid before informing the client that they now held the lock.

Adding an additional lease check, as well as choosing longer TTLs and tuning election timeouts carefully, can all help reduce the frequency of this issue. However, mutex violations cannot be eliminated altogether, because distributed locks are a fundamentally unsafe concept in asynchronous systems. As Dr. Martin Kleppmann describes eloquently in his article on distributed locking, lock services must sacrifice correctness in order to preserve liveness in asynchronous systems: if a process crashes while holding a lock, the lock service needs some way to force the release of the lock in order to make progress. However, if the process is *not* in fact dead, but merely slow or unreachable, releasing the lock could lead to it being held in multiple places at once.

Even if a distributed lock service *were* to take advantage of a magical failure detector and actually guarantee mutual exclusion, it would still, in general, be unsafe to use that lock service to guarantee the ordering of operations on some non-local resource. If process A sends a message to database D while holding a lock, A crashes, and process B acquires the lock and sends a message to D, then the message sent by A might arrive (thanks to asynchrony) *after* process B's message, violating the mutual exclusion property that the lock was intended to provide.

In order to prevent this problem, one must rely on the storage system itself to ensure transactional correctness, or, if the lock service provides one, use a fencing token of some kind, which is included with every operation a lockholder performs and used to ensure that no previous lockholder's operations interleave with the current lockholder's. Google's Chubby lock service, for instance, calls these tokens *sequencers*. In etcd, users can use the revision of their lock key as a globally ordered fencing token.

In addition, etcd lock keys can be used to protect transactional updates to etcd itself. By executing a transaction which checks to see if the lock key's version is greater than zero, users can prevent a transaction from taking effect if the lock is no longer held. In our tests, this approach safely isolated read-modify-write operations where the write was a single lock-guarded transaction. This approach provides an isolation property akin to fencing tokens, but (like fencing tokens) does not guarantee atomicity: a process could crash or lose its mutex during a multi-operation update, leaving etcd in a logically inconsistent state.

№	Summary	Event Required	Fixed in
11496	Watches beginning at revision 0 start later	None	Unresolved
11456	Locks return after blocking without checking ownership	None	Master
11457	Locks are not documented as unsafe	None	Unresolved

4 Discussion

In our tests, etcd 3.4.3 lived up to its claims for key-value operations: we observed nothing but strict-serializable consistency for reads, writes, and even multi-key transactions, during process pauses, crashes, clock skew, network partitions, and membership changes. Strict-serializable behavior was the default for key-value operations; performing reads with the `serializable` flag allowed stale reads, as documented.

Watches appear correct, at least over single keys. So long as compaction does not destroy historical data while a watch isn't running, watches appear to deliver every update to a key in order.

However, etcd locks (like all distributed locks) do not provide mutual exclusion. Multiple processes can hold an etcd lock concurrently, even in healthy clusters with perfectly synchronized clocks. The lock API documentation failed to mention this issue, and the only examples of locks provided were unsafe. Locking issues should be somewhat reduced after this patch is released.

The etcd team has made several changes to their documentation as a result of our collaboration, which should be published in upcoming versions of the etcd website.[1] The API guarantees page on GitHub now says etcd is strict serializable by default, and no longer claims that sequential and serializable are the strongest consistency levels achievable in distributed systems. Revisions are now documented to start at 1, though the watch API documentation still does not discuss that passing a revision of 0 means "return events after the current revision, plus one", rather than "return all events". Documentation on lock safety issues is under development.

Some documentation changes, such as documenting the special behavior of revision 0 in watches, still await attention.

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 make extensive efforts to find problems, we cannot prove etcd's correctness.

4.1 Recommendations

If you use etcd locks, consider whether those locks are used to ensure safety, or simply to improve performance by probabilistically limiting concurrency. It's fine to use etcd locks for performance, but using them for safety might be risky.

Specifically, if you use an etcd lock to protect a shared resource like a file, database, or service, that resource should guarantee safety *without* a lock involved. One way to accomplish this is to ensure that every interaction with the shared resource includes a monotonic fencing token—for instance, the etcd revision associated with the currently-held lock key. The shared resource should ensure that once a client has used a token y to perform some operation, any operations using a lower token $x < y$ must fail. This approach does not ensure atomicity, but it does ensure operations performed under a lock are contiguous, rather than interleaved.

We suspect that users are unlikely to encounter this—but if you *do* rely on reading all changes from etcd, starting at the first revision, remember to pass 1, not 0, for the first watch revision. As far as we can determine experimentally, revision 0 means "the current revision", not "the earliest revision".

Finally, etcd's locks, like all distributed locks, are dangerously named: users might be tempted to use them like normal locks, and be surprised when they fail to provide mutual exclusion. The API documentation, blog posts, and GitHub issues fail to note this risk. We recommend that the etcd documentation inform users that locks do not provide mutual exclusion, and show examples of using fencing tokens to update shared state, rather than examples which could lose updates.

4.2 Future Work

The etcd project has been stable for some years now: the Raft algorithm at its core is well-established, etcd's key-value API is simple and well-understood, and while tangential features like locks and watches have

[1] As of this writing, documentation fixes are in master on GitHub, but have not yet been published to the etcd documentation site.

gained new APIs recently, their semantics are relatively straightforward. We believe we have reasonable coverage for basic gets, puts, transactions, locks, and watches. However, there are some additional tests we could perform.

Our tests do not evaluate deletes rigorously, and there might be edge cases around versions vs revisions when objects are repeatedly created and deleted. Future tests could investigate deletions in more depth. We have also not tested range queries, or multi-key watch operations, though we suspect their semantics are similar to single-key operations.

While we test with process pauses, crashes, clock skew, partitions, and membership changes, we have not investigated disk corruption or other byzantine single-node faults. Future research could investigate these possibilities.

This work was funded by The Cloud Native Computing Foundation, which is part of The Linux Foundation, and was conducted in accordance with the Jepsen ethics policy. We wish to thank the etcd team for their invaluable assistance, especially Chris Aniszczyk, Gyuho Lee, Xiang Li, Hitoshi Mitake, Jingyi Hu, and Brandon Philips.