jetcd 0.8.2

Kyle Kingsbury
2024-08-08

Jetcd is the official Java client library for the etcd coordination service. We show that jetcd contains an improper retry mechanism which allows transactions to execute multiple times, or to appear to fail but actually succeed. To users, these behaviors may appear as lost update, circular information flow, and aborted read. These issues have been outstanding for two and a half years. No patch is available, but disabling the retry mechanism is straightforward. This work was performed independently by Jepsen without compensation, and conducted in accordance with the Jepsen ethics policy.

1 Background

Etcd is a coordination service designed to offer Strict Serializable reads, writes, and micro-transactions over a small, in-memory key-value store. In 2022, Jepsen re-tested etcd as a part of a private engagement covering an assortment of databases. We reported several issues to the etcd team during that engagement. Some of those issues remained unresolved, and the etcd team periodically speculated as to possible causes, asking for Jepsen's assistance.

In July 2024, Jepsen performed an independent investigation[1] to identify the cause of some of these issues, and traced them to a bug in jetcd, the official etcd client library for the Java programming language. In short: jetcd incorrectly retries non-idempotent requests which may have actually succeeded, leading to a variety of serious invariant violations. We present those behaviors here.

2 Test Design

We expanded upon the test harness written for etcd 3.4.3 in 2020. Our workloads and faults were essentially unchanged from that analysis. We focused on the list-append and wr-register workloads, which consist respectively of transactions reading and appending unique values to lists of integers, and transactions which read and write integer registers. Our sole fault was process crashes, induced by `kill -9`.

This work focused on the jetcd client library, not etcd itself. We evaluated jetcd 0.5.0, 0.6.0, and 0.8.2. Jepsen provides built-in support for capturing packets from database nodes using `tcpdump`. We used Wireshark, a network protocol analyzer, to trace exactly what messages the client sent over the wire. Wireshark includes HTTP2 and Protocol Buffer analyzers which understand etcd's wire protocol. The Protocol Buffers specification files are available from etcd.

3 Results

When an application makes a single call to (e.g.) `Txn.commit()`, jetcd may automatically retry, sending multiple copies of the network request. This is fine when the client can prove the first request could never execute, or that the request is idempotent. However, jetcd retries requests in unsafe contexts, like non-idempotent transactions which *have* been sent over the wire. This manifests as several kinds of safety violations when processes crash or the network loses messages.

3.1 Lost Update

Consider this this wr-register test run, which contains the following three transactions executed around the time one of the etcd servers crashed. First, transaction T_0 set key `3731` to `11`. Then transactions T_1 and T_2 both read value `11`, and overwrote it with values `20` and `17`, respectively:

```
T0: [[:w 3731 11] [:w 3732 17]]
T1: [[:r 3731 11] [:w 3731 20] [:w 3733 5]]
T2: [[:r 3731 11] [:w 3731 17] [:r 3732 19]]
```

[1] In the 2022 engagement, the client's engineers were enthusiastic about the prospect of a public analysis, and Jepsen was allowed to file public issues against systems including etcd. Following the conclusion of the contract, Jepsen independently completed a written report discussing the behaviors we'd found in etcd. However, Jepsen was unable to secure official permission from the client's legal department to disclose that the client had funded part of the work. This created an unusual state of affairs: the issues, test suite, and reproduction instructions were all public, but per Jepsen's ethics policy, the analysis itself could not be published. Jepsen shelved that analysis and it remains unpublished. The present analysis is based on entirely new work and verifies a different software system: jetcd, rather than etcd.

Writes in this workload are always unique: no other transaction wrote `11` to key `3731`. From the client's perspective, this behavior is indistinguishable from lost update:[2] T_1 and T_2 both read the version of key `3731` that T_0 wrote, and overwrote it. Since neither observed the other's write, one of their updates has effectively been silently discarded.

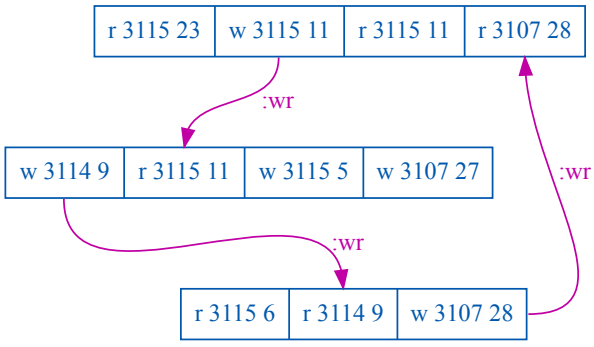
Packet capture reveals that T_0's single call to `Txn.commit()` caused two separate network requests: one 29.10 seconds into the test, and a second at 31.94 seconds. The client submitted these on separate TCP connections, and constructed different HTTP2 headers for them, but their payloads were identical. Each set key `3731`'s value to `11` (and key `3732`'s value to `17`).

Because jetcd secretly submitted T_0 twice, transactions T_1 and T_2 actually interacted with two different versions of key `3731`. The first copy of T_0 produced version 6 of key `3731` at revision 54505. Transaction T_1 read and overwrote that version at revision 54507. Two seconds later jetcd submitted a second copy of T_0, which blindly overwrote key `3731` with value `11` again—this time producing version 19, at revision 55104. Then T_2 read and overwrote this second version of value `11` with value `17`.

Lost update is prohibited by Read Committed, Snapshot Isolation, and Serializable. It should not happen in etcd, which is intended to offer at least Serializable consistency.

3.2 Circular Information Flow

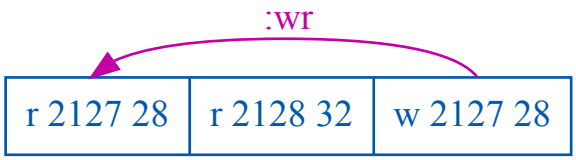
Client retries can also cause circular information flow. Take for example this wr-register run, which contains the following cycle:



Each of these transactions appears to read the other's writes. The top set key `3115` to `11`, which was read by the middle transaction. That transaction set key `3114` to `9`, which wass read by the bottom transaction. The bottom transaction set key `3107` to `28`, which was read by the top transaction. This is Adya's G1c: circular information flow. G1c should be impossible in any Read Committed system, not to mention a Serializable one. This too is a serious violation of etcd's guarantees.

This cycle occured because the client secretly submitted the middle transaction *twice*. The first execution succeeded, allowing the bottom transaction to observe its effects. However, the client opted to submit it *again*, three and a half seconds later. This second execution observed the effects of the top transaction, tying the causal loop.

Transactions can even observe their own effects from the future. In this test run, a single transaction observed a write it hadn't performed yet:



Again, packet capture shows that the client submitted this transaction twice, roughly three seconds apart. It executed both times, and the second execution observed the state from the first.

3.3 Aborted Reads

One might be inclined to add guard expressions to prevent transactions from executing more than once. Our list-append workload does just that: it performs a read-only transaction to establish current values, then performs a write transaction which commits only if the versions of any objects read are unchanged. This allows another anomaly: aborted read. For example, this list-append runs contains dozens of transactions like these:

```
T0: [[:r 4511 nil] [:append 4513 13] ...]
T1: [[:r 4513 [3 5 9 13]]]
```

Transaction T_0 aborted: its append of `13` should never have been visible to anyone. However, T_1 committed and observed value `13`. Since writes are unique, T_1 must have observed the specific version of key `4513` written by T_0: an aborted read.

The values returned from T_0's call to `Txn.commit()` indicate that its write transaction executed at revision 38343, and definitively failed: jetcd's `TxnResponse.isSucceeded()` returned `false`. But this cannot be the whole story, because T_1 ostensibly executed earlier, at revision 34777, and observed key `4513`'s value as `[3 5 9 13]`. How is this possible? Because jetcd secretly issued T_0 twice. The client did not receive an acknowledgement from the first attempt, and the second returned a failure. Jetcd returned that failure to the caller—unsafely representing an indefinite failure as if it were definite.

Aborted reads are prohibited under a broad range of consistency models from Read Committed to Strict Serializable; this behavior is a serious violation of etcd's intended guarantees.

[2] Careful analysis of the revisions included in etcd's response metadata hints at the fact that from etcd's perspective, there were multiple executions of T_0. From the application's perspective T_0 only executed once, making this lost update.

4 Discussion

These problems affect jetcd 0.6.0 (released 2021-12-15) through 0.8.2 (the most recent version, released 2024-05-30). Users of these versions may observe apparent instances of aborted read, circular information flow, lost update, and other anomalies when processes crash or the network loses messages. Transactions may appear to fail but actually succeed. Transactions may be applied multiple times. Relying on conditional writes is not sufficient to obtain Serializability: aborted read occurs even when every write is guarded by a version check. These behaviors are not difficult to observe: we can reliably reproduce them in under a minute.

Jepsen reported these problems to both etcd and jetcd in 2022, filing jetcd-1072, etcd-14092, and etcd-14890. The etcd maintainers suggested a number of possible explanations, including that these were not actually anomalies, that they represented a bug in Jepsen, or that they were caused by a bug in jetcd. No one followed up on the jetcd issue, and it was automatically closed as stale. The etcd team closed 14092 as a jetcd bug, and eventually closed 14890, incorrectly concluding the report was inaccurate and most likely represented a mistake in Jepsen.

As always, we caution 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 jetcd's correctness

4.1 Recommendations

One can disable jetcd's retry mechanism by creating clients with `ClientBuilder.retryMaxAttempts(0)`. This appears to resolve the issues discussed in this report. We recommend jetcd users disable retries until a patch is available and tested.

Unsafe retries are a recurring cause of safety errors. For instance, TiDB 2.1.7 exhibited read skew and lost update thanks to an automatic retry mechanism. MongoDB 4.2.6's retry system allowed retrocausal transactions which read their own future effects. Retrying is safe when a client can prove that the operation cannot have taken place (e.g. because DNS resolution failed or the node receiving the message refused to execute it). It is also safe when the operation is idempotent (e.g. adding an element to a grow-only set, or committing only if a unique ID for that operation has not already been applied). For this reason, distributed systems engineers should take particular care to separate *definite* failures (operations which definitely did not happen) from *indefinite* failures (operations which may or may not have happened). Engineers should also be careful when classifying operations as idempotent. It is easy to assume that `set(x, 5)` is idempotent because applying it twice in a row still produces the state `x = 5`. However, this operation is not longer idempotent if its executions are interleaved with other writes—then, it leads to lost update.

In general, clients are a part of distributed systems. From the perspective of an application, the client and servers together *are* the system. Safety errors in the client library can be indistinguishable from those in servers. It is therefore especially important that distributed systems maintainers test and resolve issues in not only servers but clients as well.

Since jetcd is "the official [J]ava client for etcd v3", it is particularly worrying that it has contained a serious safety bug for two and a half years, and that these issues went unresolved for so long. We recommend that the etcd and jetcd teams find a way to jointly investigate and resolve critical safety issues in official client libraries.

4.2 Future Work

This analysis focuses on jetcd, rather than etcd itself. Other issues in etcd may remain. For example, etcd currently experiences a broad array of crashes when nodes encounter filesystem corruption or the loss of un-fsynced writes. Further research on etcd safety would likely prove fruitful.

Jepsen wishes to thank C Scotta Andreas, Tim Kordas, Ben Linsay, Nathan Taylor, and James Turnbull for their review of early drafts. This work was performed independently by Jepsen without compensation, and conducted in accordance with the Jepsen ethics policy.