RethinkDB 2.2.3 Reconfiguration

2016-01-22

In the previous Jepsen analysis of RethinkDB, we tested single-document reads, writes, and conditional writes, under network partitions and process pauses. RethinkDB did not exhibit any nonlinearizable histories in those tests. However, testing with more aggressive failure modes, on both 2.1.5 and 2.2.3, has uncovered a subtle error in Rethink's cluster membership system. This error can lead to stale reads, dirty reads, lost updates, node crashes, and table unavailability requiring an unsafe emergency repair. Versions 2.2.4 and 2.1.6, released last week, address this issue.

Until now, Jepsen tests have used a stable cluster membership throughout the test. We typically run the system being tested on five nodes, and although the network topology between the nodes may change, processes may crash and restart, and the system may elect new nodes as leaders, we do *not* introduce or remove nodes from the system while it is running. Thus far, we haven't had to go that far to uncover concurrency errors.

Since RethinkDB passed its stable-membership partitioning tests, I offered the team a more aggressive failure model: we'd dynamically reconfigure the cluster membership during the test. This is a harder problem than consensus with fixed membership: both old and new nodes must gracefully agree on the membership change, ensure that both sets of nodes will agree on any operations performed during the handover, and finally transition to normal consensus on the new set of nodes. The delicate handoff of operations from old nodes to new provides ample opportunities for mistakes.

1 Rethink's consensus system

In order for RethinkDB to provide linearizable operations on a specific key in a table, the cluster must agree on which nodes should be replicas for a given piece of data, and which replica will be the *primary* for that data—charged with coordinating updates and linearizable reads. This is the consensus problem, and RethinkDB uses Raft to obtain that consensus.

Note that Rethink does not use Raft to obtain consensus on the data itself—only on the *table metadata*: for a given table, what replicas exist, which one should be the default primary, and so on. This table metadata is called a *configuration*. By carefully coupling Rethink's data replication algorithm to the table configuration, and the configuration to the Raft state, Rethink can offer linearizable isolation on individual keys without involving the full Raft state machine for every operation. This offers two performance benefits: first, single-key operations do not require the global order that Raft would impose, which improves concurrency, and second, RethinkDB can *shard* its data, so only a single shard's (not the entire table's) replicas must be involved in a given write or read.

The Raft paper sketches an extension to the consensus algorithm for adding and removing nodes from the cluster, and Rethink implements that algorithm for configuration changes—allowing one to add and remove replicas (and by extension, Raft nodes) on the fly. There are two phases to this membership transition.

First, the leader enters a joint-consensus mode, where it broadcasts messages to all nodes in the old *and* new configurations. In order to commit a log entry or become a leader, that request must be acknowledged by a majority of the old nodes—and independently, a majority of the new. The leader broadcasts that joint-consensus configuration as a log entry to all nodes, which apply it immediately.

Once the joint-consensus mode is committed, neither the old configuration nor the new configuration can act independently. This clears the way for the old configuration to be abandoned: a leader broadcasts a message that only the new nodes should be used, and again, the followers apply that message to their local Raft state

machines immediately.

Nodes may become isolated or fail during the transition, stranding them with out-of-date ideas about *who should even be involved* in making a decision. Their logs may be overwritten by newer leaders, leading to confusion about the authoritative state. We can exacerbate these problems by introducing network partitions, which will force larger concurrency windows for the reconfiguration procedure. With luck, Rethink will slip up during reconfiguration, and we'll be able to observe a consistency error.

2 Designing a test

For the test workload, we'll use the same reads, writes, and compare-and-set operations over a single document from the previous Rethink analysis. Because checking long histories for linearizability is expensive, we'll break up our test into operations on different documents, and check each one independently—only working with a given document for ~60 seconds. Since handoff generally takes place within 15 seconds, this should be long enough to detect consistency violations.

The main difference in this test is that we'll add a new type of *nemesis*: the special Jepsen process which introduces failures into the distributed system. We'll use RethinkDB's reconfiguration API to assign a new set of replicas and a preferred primary node for the test's table. Then, we'll design a special nemesis which chooses random replicas and primaries, and reconfigures the table accordingly.

There's only one configuration using all five nodes, five configurations using a single node, but twenty configurations which pick two or three out of five. Based on a hunch that extreme cluster sizes might be important, we'll pick a uniformly random replica count, then select a random set of replicas to create a set of that size. A randomly selected default primary rounds out the configuration. We'll open a connection to the new primary and ask it to apply our chosen configuration.

That reconfiguration could fail because a server we need to contact in the reconfiguration is unavailable, or because the entire table is presently down—at least as seen from the node doing the reconfiguration. In both of these cases we'll perform a limited number of blind retries, which significantly improves our chances of finding a reconfiguration possible under the current network conditions.

To make matters more complex, we'll combine that nemesis with partition-random-halves, which divides the network into randomly selected halves and heals their connections later. We use `nemesis/compose` to combine multiple nemeses into one, routing `:reconfigure` ops to our custom reconfiguration nemesis, and `:start`/`:stop` to the network partitioner.

```
(nemesis/compose
  {#{:reconfigure} (reconfigure-nemesis "jepsen" "cas")
   #{:start :stop} (nemesis/partition-random-halves)})
```

A slight change to our generator as well: we'll emit a `:reconfigure` op between each stage of the network partitioner's `:starts` and `:stops`. We also force the nemesis to wait until the client has created the table *before* we mess with the table's replica configuration—that could lead to awkward deadlocks.

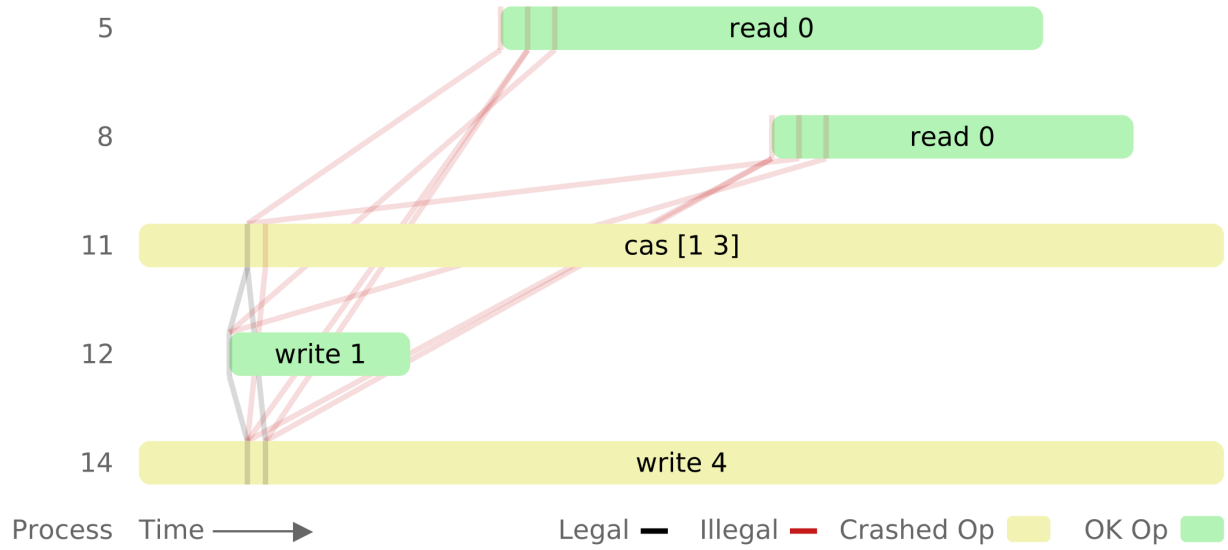
```
(gen/nemesis
  (gen/phases
    (gen/await
      (fn []
        (info "Nemesis waiting")
        (deref (:table-created? (:client t)))
        (info "Nemesis ready to go")))
    (->> (cycle [{:type :info, :f :start}
                 {:type :info, :f :stop}])
         (interpose {:type :info, :f :reconfigure})
         (gen/seq))))
```

We know from the last analysis that every consistency level lower than `read=majority` and `write=majority` will lead to nonlinearizable histories, so we'll only run this test with `majority/majority`. And indeed, this test, like those from the previous analysis, passes the linearizability checker several times in a row.

But then, *a mystery appears*.

3 A read anomaly

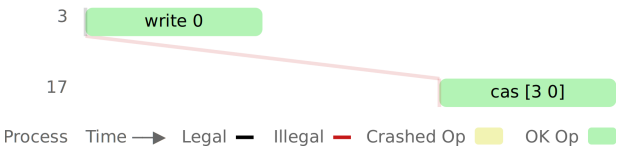
This diagram shows the operations (green and yellow bars) performed by processes (arranged vertically) over time (flowing left to right). Green operations succeeded; yellow operations crashed and may or may not take place. Black lines show the possible legal state transitions through this history, and red lines show inconsistent state transitions which would violate the rules of a compare-and-set register. For instance, it would be legal to write 1, then compare-and-set (cas) 1 to 3, because the current value would be 1. But we couldn't subsequently read 0, because the register's value along that path would be 3, not 0.



Because there are no legal paths leading to the first (or the second, for that matter) read of 0, this history is *nonlinearizable*. You can't write 1, then read 0, if the only other ops that could take effect would result in the value being 3 or 4.

4 A write anomaly

It gets much worse.



This kind of result could indicate several consistency errors. It could be a stale read: the reads of 0 could be seeing an earlier legal state, prior to the write of 1. It could be a dirty read, if a write of 0 took place, failed, but its results were somehow visible to another transaction. Or it could be a lost update: the write of 1 could have been acknowledged, then discarded. In order to rule out stale and dirty reads, we can remove reads from the workload, making every operation either a blind or conditional (cas) write.

In this history, we write 0, then compare-and-set 3 to 0—which should only succeed if the current value is 3. There are no concurrent operations, and no reads to confound our test with read anomalies. This is conclusive evidence that RethinkDB allows inconsistency in the write path: lost updates.

The full history for this test gives us deeper context. Process 3 writes 0, the nemesis reconfigures the cluster so that the sole replica is n4, and after a series of

failed operations we know did not take place, process 17 compare-and-sets 3 to 0.

```
:nemesis :info :reconfigure
  {:replicas (:n3),
   :primary  :n3,
   :grudge   {:n3 [(:n4)],
              :n4 [(:n3)]}}
12 :invoke :write 3
17 :invoke :cas   [4 2]
12 :ok     :write 3
17 :fail   :cas   [4 2]

... lots of failed ops ...

3  :invoke :write 0
3  :ok     :write 0
12 :invoke :cas   [0 0]
17 :invoke :cas   [1 4]
12 :fail   :cas   [0 0]
17 :fail   :cas   [1 4]
:nemesis :info :reconfigure
  {:replicas (:n4),
   :primary  :n4,
   :grudge   {:n4 [(:n3)],
              :n3 [(:n4)]}}

... more failed ops ...

12 :invoke :cas   [3 3]
17 :invoke :cas   [3 0]
12 :fail   :cas   [3 3]
17 :ok     :cas   [3 0] <--- Violation
```

Initially, the primary replica is n3, and n3 is isolated from n4. Because Jepsen stripes processes across nodes, process 2, 7, 12, ... all talk to node n3. Thus, process 12 writes 3 to n3, process 3 writes 0 to n4, we assign n4 as the new primary (again, isolated from n3), and process 17 CAS's 3 to 0 on n3.

So node n3 sees a write of 3, and a CAS of 3 to 0. That's legal. Node n4 sees a write of 0. Also legal. It's as if n3 and n4 were running in *independent clusters*; each one accepting writes without replicating them to the other. They are, after all, separated by a network partition.

Weeks of experimentation confirms that changing a table's replica configuration while nodes undergo network partitions can induce split-brain phenomena—but the results are hard to reproduce. It can take minutes to hours of reconfiguration, cutting, reconfiguring, healing, and so on to find one of these cases. Since most people don't reconfigure their clusters very often, I suspect these bugs are unlikely to affect many users in production.

When it *does* happen, however, the results are catastrophic. Split-brained Raft ensembles will happily assign totally or partially isolated replicas, each of which believes they have independent authority to service writes and reads. As far as I can tell, this situation can continue indefinitely, until an operator notices lost writes and takes action. The best way to reconcile the problem, I think, is to identify one of the configurations you'd like to keep, nuke the other one's nodes, and possibly perform an emergency repair on the table to establish a new final configuration. Keep in mind the emergency repair process *also* invalidates consistency guarantees—it must, in order to recover from inconsistent cluster states.

5 Invalid log windows

This split-brain behavior doesn't require any nodes to crash. But sometimes, during these tests, nodes *do* crash—and this provides a tantalizing clue:

```
2016-01-20T19:44:27.156552738 0.051520s error: Guarantee failed:
[index <= get_latest_index()] the log doesn't go forward this far
2016-01-20T19:44:27.156563122 0.051530s error: Backtrace:
...
2016-01-20T19:44:27.288763935 0.183731s error: Exiting.
```

Once this crash occurs, the node can't be restarted without crashing again. You can only recover by wiping out its persistent state and starting afresh.

```
2016-01-21T20:45:45.069903463 111.702626s error: Error in
./src/clustering/generic/raft_core.tcc at line 1040:
2016-01-21T20:45:45.069933444 111.702656s error: Guarantee failed:
[last >= first - 1]
```

This is bug 4979, originally discovered by the RethinkDB team in October 2015 using their table fuzz-tester. The apply_log_entries function, which takes a Raft log and applies some of its entries to the local state machine, ensures that the range of entries it's being asked to apply is a proper range—the last index can't be lower than the first index. The first index likely comes from the local log's committed index, and the last index probably comes from a leader's committed index—which should *never* be lower than any follower's index.

There's something fishy here, but for several months we couldn't figure out *how* this situation could arise. The Rethink team committed a patch which partially addressed the issue back in October, but the issue resurfaced in fuzz testing and again in Jepsen tests. The cause remained elusive.

6 Multiple Raft leaders

Meanwhile, another crash from these reconfiguration tests points to the possibility of multiple Raft leaders:

```
2016-01-21T21:02:10.488564358 494.715133s error: Error in
./src/clustering/generic/raft_core.tcc at line 971:
2016-01-21T21:02:10.488589668 494.715158s error: Guarantee failed:
[mode != mode_t::leader]
```

This assertion relies on the fact that the Raft leader election algorithm guarantees only one node will ever be the leader for a given term.

```
/* Raft paper, Section 5.2: "at most one candidate can win the election for a
particular term" If we're leader, then we won the election, so it makes no sense
for us to receive an RPC from another member that thinks it's leader. */
guarantee(mode != mode_t::leader);
```

And this crash, which occurs on a follower, appears related:

```
2016-01-21T21:02:10.400077788 494.534726s error: Error in
./src/clustering/generic/raft_core.tcc at line 986:
2016-01-21T21:02:10.400115720 494.534763s error: Guarantee failed:
[current_term_leader_id == request_leader_id]
```

This assertion double-checks the single-leader invariant by enforcing that once a follower learns who the leader is for the current term, any writes for that term should come from *the same leader*:

```
/* Raft paper, Section 5.2: "at most one candidate can win the election for a
particular term" */
guarantee(current_term_leader_id == request_leader_id);
```

This hints that somehow, two Raft nodes believed they were the leader for the same term—but again, we don't know *why*. The RethinkDB team and I reviewed their Raft core implementation, but found nothing of consequence.

7 The cause

So we have nodes being asked to apply Raft operations from *behind* their committed log index—which is supposed to be stable. We have leaders which discover other nodes are the leaders for the same term. We have followers which receive messages from multiple leaders for the same term. These all suggest some kind of rare edge case leading to a schism in the Raft cluster. Poring over various failure schedules and Jepsen

histories suggests that a combination of large (~5 node) and small (~1 node) replica configurations are involved, and the problem only manifests when network partitions isolate the small replicas from large ones. But *why*?

The Rethink team and I spent several weeks poring over the code and trying to develop a more efficient test to reproduce the problem. Rethink's engineers fixed a few tangential problems along the way, but the tests continued to fail until several members of Rethink's team identified a deliciously complex bug. Daniel Mewes' writeup of their findings is delightfully comprehensive and well worth your time, but I'll summarize it here.

As we discussed earlier, RethinkDB runs a Raft cluster across all replicas for a table (except non-voting replicas). When you add a *new* node to a table, the existing Raft cluster for that table picks a Raft node ID for the new replica, then contacts the *multi table manager* on the new node, which spins up a new instance of Raft with that ID. From there, the Raft joint-consensus protocol takes care of transitioning to the new node configuration. When you remove a replica, the multi table manager destroys the Raft instance and wipes its storage clean.

If a removed replica is later re-added, the existing cluster would pick a new node ID and send a new `ACTIVE` message to the multi table manager on that replica. The fresh node ID ensures that the cluster knows the new Raft node *has no data yet*. If we re-used a node ID on a node whose state has been wiped, then it could appear that node had *remained in the cluster* but lost all its data—which would violate Raft's assumptions about stable node storage.

Consequently, it's crucial that the multi table manager processes `ACTIVE` and `INACTIVE` messages *in order*. If a node ever applied an old `ACTIVE` message *a second time*, it'd re-use the former node ID, which would appear as data loss to other nodes who still believe that node is in the cluster. Therefore, every `ACTIVE` and `INACTIVE` message includes a monotonic logical *timestamp*, ordered by the Raft cluster itself. The multi table manager will only apply an ACTIVE or INACTIVE message if its timestamp is higher than the current state. This ensures that node IDs are not re-used.

Except.

A bug in 2.1.0 allowed replicas to generate timestamps of 263: the maximum integer size. This prevented the multi table manager from ever applying any newer configuration, effectively locking replicas in an inactive state. As a workaround, the multi table manager's message ordering code has an escape hatch: it always allows `INACTIVE -> ACTIVE` transitions *regardless of their timestamp*.

```
/* If we are inactive and are told to become active, we ignore the
timestamp. The `base_table_config_t` in our inactive state might be
a left-over from an emergency repair that none of the currently active
servers has seen. In that case we would have no chance to become active
again for this table until another emergency repair happened
(which might be impossible, if the table is otherwise still available). */
bool ignore_timestamp =
    table->status == table_t::status_t::INACTIVE
    && action_status == action_status_t::ACTIVE;
```

So, if messages are delayed or reordered during a cluster reconfiguration (say, due to a network partition), *and* a replica is removed from a table, *and* that message delay allows a duplicate `ACTIVE` message to be delivered to that replica *after* it's received an `INACTIVE`, it's possible for that replica to rejoin the cluster using its *old* node ID, but *with all its data missing*.

Suffering from retrograde amnesia, that replica can induce chaos in the Raft cluster. For example, it can re-cast votes for elections it already participated in. That allows two leaders to win the election for a single term—which explains the leader invariant violations we saw earlier. With two leaders comes inconsistency about committed log offsets, which explains the log index crashes as well. Both leaders can elect independent RethinkDB primaries, which can go on to independently satisfy reads and writes—causing the anomalies we saw in the test.

With this special case removed, RethinkDB appears to pass Jepsen's linearizability tests during network partitions and reconfigurations. There may be other bugs lurking in the depths, but after dozens of hours of testing, we think things are relatively safe.

8 Discussion

RethinkDB passed initial Jepsen tests, providing linearizable single-key operations through network partitions. However, long-running tests, involving randomized network partitions *and* reconfiguring the cluster membership, resulted in nonlinearizable histories: stale reads, illegal compare-and-sets, and the loss of acknowledged operations.

These faults stem from a violation of the Raft algorithm's assumptions around stable storage for each Raft node—which occurred because the system responsible for creating and destroying Raft nodes could, with the right order of message deliveries, apply config changes out of order, causing a node to re-use an old Raft node ID. That reordering was only possible because of a special workaround for a bug in an older version of RethinkDB.

The RethinkDB team has identified the error and has released version 2.2.4 with a patch. They've also backported the fix to 2.1.6.

What are the risks to users? The RethinkDB team and I suspect it's unlikely this bug will occur outside of stress testing. Cluster reconfiguration is typically infrequent, and users would need a specific series of network failures or other message delays which happen to cut the replicas apart—in a way which allows both network components to find independent majorities for their respective table configurations. In Jepsen tests, it usually takes tens to hundreds of partition/reconfigure rounds to trigger this bug.

RethinkDB's engineers noted three takeaways from tracking down this bug. First, fuzz-testing—at both the functional and integration-test level, can be a powerful tool for verifying systems with complex order dependence. Second, runtime invariant assertions were key in identifying the underlying cause. A test like Jepsen can tell you that the cluster can exhibit split-brain behavior, but can't tell you anything about *why*. The error messages from those leader and log order assertions were key hints in tracking down the bug. Rethink plans to introduce additional runtime assertions to help identify future problems. Finally, they plan to devote more attention to issues which suggest—even tangentially—consistency errors.

More generally, this adventure illustrates that time-honored aphorism: "distributed systems are hard". Managing cluster transitions—leader elections, recovering from crashes, adding nodes, removing nodes… these processes involve complex and subtle protocols. Even when a peer-reviewed consensus algorithm is employed, and the implementation carefully tested, problems can arise at the *boundary* around the consensus algorithm's core. We saw this in etcd and Consul, which ordered writes safely, but allowed stale reads by improperly coupling reads to local leader state. In Rethink's case, the Raft implementation's assumptions were compromised by allowing the system to re-use node identifiers. This is where static and runtime invariant checking, verified by generative testing techniques, can help identify subtle bugs.

My thanks to the entire RethinkDB team, especially Daniel Mewes, Jeroen Habraken, Michael Glukhovsky, Michael Lucy, Slava Akhmechet, and Tim Maxwell, for their enthusiastic support of this research. I am also indebted to Jared Morrow, Marc Hedlund, and Kelly Norton for their comments and questions. This report comprises the second half of a two-part contract with RethinkDB, and was conducted in accordance with the Jepsen ethics policy.