Call me maybe: etcd and Consul

2014/06/09
Distributed Systems Jepsen Networks Etcd Consul

In the previous post, we discovered the potential for data loss in RabbitMQ clusters. In this oft-requested installation of the Jepsen series, we'll look at etcd: a new contender in the CP coordination service arena. We'll also discuss Consul's findings with Jepsen.

Like Zookeeper, etcd is designed to store small amounts of strongly-consistent state for coordination between services. It exposes a tree of logical nodes; each identified by a string key, containing a string value, and with a version number termed an *index*–plus, potentially, a set of child nodes. Everything's exposed as JSON over an HTTP API.

Etcd is often used for service discovery, distributed locking, atomic broadcast, sequence numbers, and pointers to data in eventually consistent stores. Because etcd offers atomic compare-and-set by both value and version index, it's a powerful primitive in building other distributed systems.

In this post, we'll write a Jepsen test for etcd, and see whether it lives up to its consistency claims.

Writing a client

A client, in Jepsen, applies a series of operations to one particular node in a distributed system. Our client will take invocations like `{:process 2, :type :invoke, :f :cas, :value [1 2]}`, try to change the value of a register from `1` to `2`, and return a *completion* like `{:process 2, :type :ok, :f :cas, :value [1 2]}` if etcd acknowledges the compare-and-set. If you're a little confused, now might be a good time to skim through the earlier discussion of strong consistency models.

So: first things first. We'll define a new datatype, called `CASClient`, with two fields: a key `k`, and an etcd client `client`.

```
(defrecord CASClient [k client]
```

Jepsen has a protocol–a suite of functions–for interacting with clients. We'll define how CASClient supports that protocol by declaring the protocol name `client/Client`, followed by three functions from that protocol: `setup!`, `invoke!`, and `teardown!`.

```
  client/Client
  (setup! [this test node]
```

The setup function takes three arguments: the CASClient itself (`this`), the test being run, and the name of the node this client should connect to. Think of a client like a stem cell: before the test runs, it lies latent, unspecialized. When the test starts, we'll spawn a client for each node. The setup! function *differentiates* a latent client, returning an active client bound to one particular node. Some state, like the key `k`, will be inherited by the new client. Other state, like the database connection, will be set up for each new client independently.

```
    (let [client (v/connect (str "http://" (name node) ":4001"))]
      (v/reset! client k (json/generate-string nil))
      (assoc this :client client)))
```

In this namespace, we'll use my Verschlimmbesserung etcd client and call its namespace `v`. `v/connect` creates a new Verschlimmbersserung client for the given node. We call `v/reset!` to initialize the key `k` to `nil`, json-encoded. Then, using `assoc`, we return a copy of this CASClient, but with the `:client` field replaced by the Verschlimmbersserung client.

Next, we'll implement the Client protocol's `invoke!` function, which takes a Client, a test, and an invocation to apply.

```
  (invoke! [this test op]
```

Things get a little complicated now. We often say that an unexpected exception or a timeout means an operation *failed*–but in verifying strong consistency, we need to be a little more precise. We must distinguish between three outcomes:

1. `:ok` results, where the operation definitely occurred,
2. `:fail` results, where the operation definitely did not occur, and
3. `:info` results, where the operation *might or might not* have taken place.

Indeterminate results, like timeouts, are the bane of the model checker. We never know whether those operations might complete at some point hours or weeks later–so when a timeout occurs, we consider the process *crashed* and spawn a new one. That process is still concurrent with every subsequent operation in the history, which imposes a huge cost at verification time. Wherever possible, we want to declare definitively that an operation did or did not happen.

Reads are a special case: they don't affect the state of the system, so as far as the model checker is concerned, an indeterminate read can always be interpreted as *never having happened at all*–e.g., a `:fail` state. We're going to use this distinction in some error handling code later.

```
    ; Reads are idempotent; if they fail we can always assume they didn't
    ; happen in the history, and reduce the number of hung processes, which
    ; makes the knossos search more efficient
    (let [fail (if (= :read (:f op))
                 :fail
                 :info)]
```

Now, depending on the function `:f` of the invoke operation, we'll either read, write, or compare-and-set the value at key `k`.

```
      (try+
        (case (:f op)
          :read  (let [value (-> client
                                 (v/get k {:consistent? true})
                                 (json/parse-string true))]
                   (assoc op :type :ok :value value))

          :write (do (->> (:value op)
                          json/generate-string
                          (v/reset! client k))
                     (assoc op :type :ok))

          :cas   (let [[value value'] (:value op)
                       ok?            (v/cas! client k
                                              (json/generate-string value)
                                              (json/generate-string value'))]
                   (assoc op :type (if ok? :ok :fail))))
```

For a read, we'll take the client, get the key using an etcd consistent read, and parse the key as JSON. Then we'll return a copy of the invocation, but with the type `:ok` and the `:value` obtained from etcd. Note that we're using etcd's consistent read option, which claims:

> If your application wants or needs the most up-to-date version of a key then it should ensure it reads from the current leader. By using the consistent=true flag in your GET requests, etcd will make sure you are talking to the current master.

For a write, we'll take the `:value` from the operation, serialize it to JSON, and call `v/reset!` to change the register to that new value.

For a compare-and-set (`:cas`), we'll take a pair of values–old and new–and bind them to `value` and `value'`. We'll serialize both to JSON, and call `v/cas!` to atomically set `k` to the new value iff it currently has the old value. `v/cas!` returns true if the CAS succeeded, and false if the CAS failed, so we return a `:type` of `:ok` or `:fail` depending on its return value `ok?`.

Finally, we'll handle a few common error conditions, just to reduce the chatter in the logs.

```
        ; A few common ways etcd can fail
        (catch java.net.SocketTimeoutException e
          (assoc op :type fail :value :timed-out))

        (catch [:body "command failed to be committed due to node failure\n"] e
          (assoc op :type fail :value :node-failure))

        (catch [:status 307] e
          (assoc op :type fail :value :redirect-loop))

        (catch (and (instance? clojure.lang.ExceptionInfo %)) e
          (assoc op :type fail :value e))

        (catch (and (:errorCode %) (:message %)) e
          (assoc op :type fail :value e)))))
```

If we're doing a read, these error handlers will return `:fail` as a performance optimization. If we're doing a write or CAS, they'll return `:info`, letting Knossos know that those operations might or might not have taken place.

One last function from the `Client` protocol: `teardown!`, which releases any resources the clients might be holding on to. These clients are just stateless HTTP wrappers, so there's nothing to do here.

```
  (teardown! [_ test]))
```

That's it, really. We'll write a little function to create a latent instance of this datatype, and use it in our test! We'll call our key `"jepsen"`, and leave the client field blank–it'll be filled in by calls to `setup!`.

```
(defn cas-client
  "A compare and set register built around a single etcd key."
  []
  (CASClient. "jepsen" nil))
```

A singlethreaded model

We need a *model* of an etcd register to go along with this client. The model's job is to take an operation, apply it to the current state of the model, and return a new model state–or a special inconsistent state if the given operation *can't* be applied. We'll create a datatype called `CASRegister`, which has a single field called `value`.

```
(defrecord CASRegister [value]
  Model
  (step [r op]
    (condp = (:f op)
      :write (CASRegister. (:value op))
      :cas   (let [[cur new] (:value op)]
               (if (= cur value)
                 (CASRegister. new)
                 (inconsistent (str "can't CAS " value " from " cur
                                    " to " new))))
      :read  (if (or (nil? (:value op))
                     (= value (:value op)))
               r
```

```
            (inconsistent (str "can't read " (:value op)
                               " from register " value))))))
```

Just like our `invoke!` function, `CASRegister` chooses what to do based on the operation's function `:f`. For a write, it returns a new `CASRegister` wrapping the given value.

For a compare-and-set, it binds the current and new values, then checks whether its own value is equal to the operation's current value. If it is, we return a new register with the new value. If it *isn't*, we construct a special `inconsistent` result, explaining why the CAS operation won't work.

When a read is invoked, the client may not know *what* value it's reading. We allow the read to go through–returning the same model `r`–if the client doesn't provide a value to be read, *or* if the value the client read is equal to our current value. If the client tries to read some *specific* value, and it's not the current value of the register, though–that's an inconsistent state.

Then, a quick constructor function that starts off with the value `nil`. Note that this initial value corresponds to the value we wrote to the etcd key when the clients start up; both the real system and the model have to start in the same state.

```
(defn cas-register
  "A compare-and-set register"
  ([] (cas-register nil))
  ([value] (CASRegister. value)))
```

With the client and model written, it's time to combine both into a Jepsen test:

Designing a test

We'll start with a baseline `noop-test`, and override it with etcd-specific fields. We're pulling in etcd's `db` to automate setup and teardown of the cluster, the `cas-client` we wrote earlier, and `model/cas-register`–our singlethreaded model of a compare-and-set register. We'll use two checkers: an HTML timeline visualization, and the `linearizable` checker, powered by Knossos. A special client, the `:nemesis`, introduces network failures by partitioning the cluster into randomly selected halves.

```
(deftest register-test
  (let [test (run!
              (assoc
                noop-test
                :name      "etcd"
                :os        debian/os
                :db        (db)
                :client    (cas-client)
                :model     (model/cas-register)
                :checker   (checker/compose {:html   timeline/html
                                             :linear checker/linearizable})
                :nemesis   (nemesis/partition-random-halves)
                :generator (gen/phases
                            (->> gen/cas
                                 (gen/delay 1)
                                 (gen/nemesis
                                   (gen/seq
                                     (cycle [(gen/sleep 5)
                                             {:type :info :f :start}
                                             (gen/sleep 5)
                                             {:type :info :f :stop}])))
                                 (gen/time-limit 20))
                            (gen/nemesis
                              (gen/once {:type :info :f :stop}))
                            (gen/sleep 10)
                            (gen/clients
```

```
                                  (gen/once {:type :invoke :f :read})))))]
    (is (:valid? (:results test)))
    (report/linearizability (:linear (:results test)))))
```

The generator defines the sequence and schedule of operations. This test proceeds in phases–all clients must complete a phase before any can move to the next. We'll start off with `gen/cas`, which emits a mix of random `:read`, `:write`, and `:cas` invocations.

```
(def cas
  "Random cas/read ops for a compare-and-set register over a small field of
  integers."
  (reify Generator
    (op [generator test process]
      (condp < (rand)
        0.66 {:type  :invoke
              :f     :read}
        0.33 {:type  :invoke
              :f     :write
              :value (rand-int 5)}
        0    {:type  :invoke
              :f     :cas
              :value [(rand-int 5) (rand-int 5)]}))))
```

We wrap that generator in `gen/delay`, adding an extra second of latency to each operation to slow down the test a bit. Meanwhile, the nemesis cycles through an infinite sequence of sleeping, starting a network partition, sleeping, then resolving the partition. We limit the entire phase to 20 seconds–etcd convergence times are quite fast.

In the next phase, the nemesis emits a single `:stop` operation, resolving the network partition. We sleep for 10 seconds, then ask each client to perform a final read, just to see how the system stabilized.

Running the test

Etcd starts up *fast*. It converges in a matter of milliseconds, whereas many systems take 10 seconds or even minutes to detect failures. This is really convenient for testing–and arguably a nice property in production–but it also exposed a number of serious issues in etcd's cluster state management: most notably, race conditions.

For instance, Issue 716 caused the primary to death-spiral almost every time I stood up a cluster, even with five or ten seconds between joining each node. The etcd team was incredibly responsive about fixing these bugs, but I'm kind of surprised to find problems like this in software that's been released for almost a year. I've heard several anecdotal reports of other concurrency issues in goraft (the implementation of the underlying consensus protocol) which makes me a little nervous about trusting it, but it's tough to turn anecdotes into reproducible failure cases, so I won't dive into those here.

With some work, I was able to reliably stand up a cluster

Here's one of the shorter cases in full. First, Jepsen stands up the etcd cluster on nodes `:n1`, `:n2`, etc, and spools up five worker threads.

```
INFO  jepsen.system.etcd - :n4 etcd ready
INFO  jepsen.system.etcd - :n1 etcd ready
INFO  jepsen.system.etcd - :n5 etcd ready
INFO  jepsen.system.etcd - :n2 etcd ready
```

```
INFO  jepsen.system.etcd - :n3 etcd ready
INFO  jepsen.core - Worker 0 starting
INFO  jepsen.core - Worker 3 starting
INFO  jepsen.core - Worker 2 starting
INFO  jepsen.core - Worker 4 starting
INFO  jepsen.core - Worker 1 starting
```

Each worker thread, representing a process, concurrently invokes a series of random operations against a single etcd register. Each process talks to a distinct etcd node for each request, but follows redirects to whatever node *that* node thinks is the current leader. They happen to start off all making reads of the initial value `nil`. Then process 0 begins a read, process 3 begins a compare-and-set from `2` to `4`, which fails since the value is `nil`, and so on.

```
INFO  jepsen.util - 2   :invoke :read   nil
INFO  jepsen.util - 4   :invoke :read   nil
INFO  jepsen.util - 0   :invoke :read   nil
INFO  jepsen.util - 1   :invoke :read   nil
INFO  jepsen.util - 3   :invoke :read   nil
INFO  jepsen.util - 0   :ok     :read   nil
INFO  jepsen.util - 3   :ok     :read   nil
INFO  jepsen.util - 4   :ok     :read   nil
INFO  jepsen.util - 2   :ok     :read   nil
INFO  jepsen.util - 1   :ok     :read   nil
INFO  jepsen.util - 0   :invoke :read   nil
INFO  jepsen.util - 3   :invoke :cas    [2 4]
INFO  jepsen.util - 4   :invoke :cas    [4 4]
INFO  jepsen.util - 2   :invoke :read   nil
INFO  jepsen.util - 1   :invoke :read   nil
INFO  jepsen.util - 0   :ok     :read   nil
INFO  jepsen.util - 2   :ok     :read   nil
INFO  jepsen.util - 1   :ok     :read   nil
INFO  jepsen.util - 4   :fail   :cas    [4 4]
INFO  jepsen.util - 3   :fail   :cas    [2 4]
```

The nemesis process initiates a network partition, isolating `:n5` from `:n4` and `:n1`, isolating `:n2` from `:n4` and `:n1`, etc. Notice that it takes some time for the nemesis to make those changes to the network.

```
INFO  jepsen.util - :nemesis    :info   :start  nil
INFO  jepsen.util - 0   :invoke :write  4
INFO  jepsen.util - 0   :ok     :write  4
INFO  jepsen.util - 3   :invoke :read   nil
INFO  jepsen.util - 3   :ok     :read   4
INFO  jepsen.util - 2   :invoke :cas    [0 4]
INFO  jepsen.util - 1   :invoke :read   nil
INFO  jepsen.util - 1   :ok     :read   4
INFO  jepsen.util - 4   :invoke :write  1
INFO  jepsen.util - 2   :fail   :cas    [0 4]
INFO  jepsen.util - 4   :ok     :write  1
INFO  jepsen.util - 0   :invoke :read   nil
INFO  jepsen.util - 0   :ok     :read   1
INFO  jepsen.util - 3   :invoke :read   nil
INFO  jepsen.util - 3   :ok     :read   1
INFO  jepsen.util - 1   :invoke :read   nil
INFO  jepsen.util - 1   :ok     :read   1
INFO  jepsen.util - :nemesis    :info   :start "Cut off {:n5 #{:n4 :n1}, :n2 #{:n4
:n1}, :n3 #{:n4 :n1}, :n1 #{:n3 :n2 :n5}, :n4 #{:n3 :n2 :n5}}"
```

We see a few operations time out–which we expect from a CP system.

```
INFO  jepsen.util - 3   :info   :cas    :timed-out
INFO  jepsen.util - 1   :invoke :write  3
INFO  jepsen.util - 1   :ok     :write  3
INFO  jepsen.util - 2   :invoke :read   nil
INFO  jepsen.util - 2   :ok     :read   3
INFO  jepsen.util - 4   :invoke :cas    [2 3]
INFO  jepsen.util - 4   :fail   :cas    [2 3]
INFO  jepsen.util - 0   :invoke :write  0
INFO  jepsen.util - 8   :invoke :cas    [1 2]
INFO  jepsen.util - 1   :invoke :read   nil
INFO  jepsen.util - 1   :ok     :read   3
INFO  jepsen.util - 2   :invoke :write  2
INFO  jepsen.util - 2   :ok     :write  2
INFO  jepsen.util - 4   :invoke :cas    [2 3]
INFO  jepsen.util - 0   :info   :write  :timed-out
INFO  jepsen.util - 4   :ok     :cas    [2 3]
INFO  jepsen.util - 8   :info   :cas    :timed-out
```

After a few cycles of isolating and reconnecting nodes, something interesting happens just as the network is cut off: a "Raft Internal Error":

```
FO  jepsen.util - 4     :info   :cas    {:status 500, :errorCode 300, :message "Raft
Internal Error", :index 41}
INFO  jepsen.util - 10  :ok     :write  0
INFO  jepsen.util - :nemesis    :info   :start  "Cut off {:n1 #{:n2 :n5}, :n4 #{:n2
:n5}, :n3 #{:n2 :n5}, :n5 #{:n3 :n4 :n1}, :n2 #{:n3 :n4 :n1}}"
```

Another failure case: two nodes each think the other is the current leader, returning HTTP redirects to the other node in an infinite loop.

```
INFO  jepsen.util - 1   :invoke :read   nil
INFO  jepsen.util - 2   :invoke :read   nil
INFO  jepsen.util - 2   :ok     :read   0
INFO  jepsen.util - 1   :fail   :read   :redirect-loop
```

And the test completes. By and large, operations completed reliably with low latencies, and despite some failures, eyeballing the test things *look* correct. The Jepsen I wrote a year ago would have called these results A-OK.

```
INFO  jepsen.util - 18  :ok :read   2
INFO  jepsen.core - Worker 3 done
INFO  jepsen.core - Run complete, writing
INFO  jepsen.core - Analyzing
INFO  jepsen.core - Analysis complete
```

Jepsen II, however, is not quite so forgiving.

Results

The very first test I ran with reported a linearizability failure. I was so surprised I spent another week double-checking Knossos and Jepsen, then writing my own etcd client, to make sure I hadn't made a mistake. **Sure enough, etcd's registers are not linearizable.**

```
FAIL in (register-test) (etcd_test.clj:45)
expected: (:valid? (:results test))
  actual: false
Not linearizable. Linearizable prefix was:
2       :invoke :read   nil
```

```
4       :invoke :read   nil
0       :invoke :read   nil
1       :invoke :read   nil
3       :invoke :read   nil
0       :ok     :read   nil
3       :ok     :read   nil
4       :ok     :read   nil
2       :ok     :read   nil
1       :ok     :read   nil
0       :invoke :read   nil
3       :invoke :cas    [2 4]
4       :invoke :cas    [4 4]
2       :invoke :read   nil
1       :invoke :read   nil
0       :ok     :read   nil
2       :ok     :read   nil
1       :ok     :read   nil
4       :fail   :cas    [4 4]
3       :fail   :cas    [2 4]
0       :invoke :read   nil
0       :ok     :read   nil
2       :invoke :write  1
1       :invoke :cas    [2 3]
2       :ok     :write  1
1       :fail   :cas    [2 3]
4       :invoke :write  3
3       :invoke :write  1
4       :ok     :write  3
3       :ok     :write  1
0       :invoke :cas    [4 1]
2       :invoke :write  2
1       :invoke :write  1
0       :fail   :cas    [4 1]
3       :invoke :read   1
4       :invoke :write  0
3       :ok     :read   1
2       :ok     :write  2
1       :ok     :write  1
4       :ok     :write  0
:nemesis        :info   :start  nil
0       :invoke :write  4
0       :ok     :write  4
3       :invoke :read   4
3       :ok     :read   4
2       :invoke :cas    [0 4]
1       :invoke :read   4
1       :ok     :read   4
4       :invoke :write  1
2       :fail   :cas    [0 4]
4       :ok     :write  1
0       :invoke :read   1
0       :ok     :read   1
3       :invoke :read   1
3       :ok     :read   1
1       :invoke :read   1
1       :ok     :read   1
:nemesis        :info   :start  "Cut off {:n5 #{:n4 :n1}, :n2 #{:n4 :n1}, :n3 #{:n4
:n1}, :n1 #{:n3 :n2 :n5}, :n4 #{:n3 :n2 :n5}}"
2       :invoke :cas    [1 4]
4       :invoke :read   1
```

```
4       :ok     :read   1
2       :ok     :cas    [1 4]

Followed by inconsistent operation:
0       :invoke :read   1
```

Why aren't we allowed to read `1` from the register at this point? Knossos can provide us a litany of possible worlds, just prior to that fatal read. For instance, we might have ordered events like so: process 1 reads `nil`, process 3 reads `nil`, ⋯

```
World with fixed history:
1       :invoke :read   nil
3       :invoke :read   nil
2       :invoke :read   nil
4       :invoke :read   nil
0       :invoke :read   nil
2       :invoke :read   nil
1       :invoke :read   nil
0       :invoke :read   nil
0       :invoke :read   nil
2       :invoke :write  1
4       :invoke :write  3
3       :invoke :write  1
3       :invoke :read   1
4       :invoke :write  0
1       :invoke :write  1
2       :invoke :write  2
0       :invoke :write  4
3       :invoke :read   4
1       :invoke :read   4
4       :invoke :write  1
0       :invoke :read   1
3       :invoke :read   1
1       :invoke :read   1
4       :invoke :read   1
2       :invoke :cas    [1 4]

led to state:
{:value 4}

with pending operations:

(and 12928 more worlds, elided here)
```

But the key problem is that in all thirteen-thousand odd interpretations of this history, every one of those worlds led to a register with the value `4`.

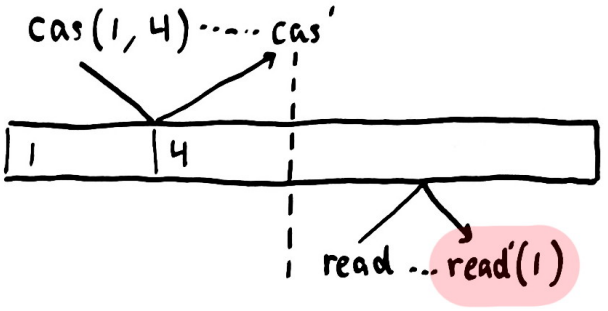
```
Inconsistent state transitions:
([{:value 4} "can't read 1 from register 4"])
```

Once that CAS goes through, a linearizable register can't return the previous value for a read. This violates linearizability.

What's going on here?

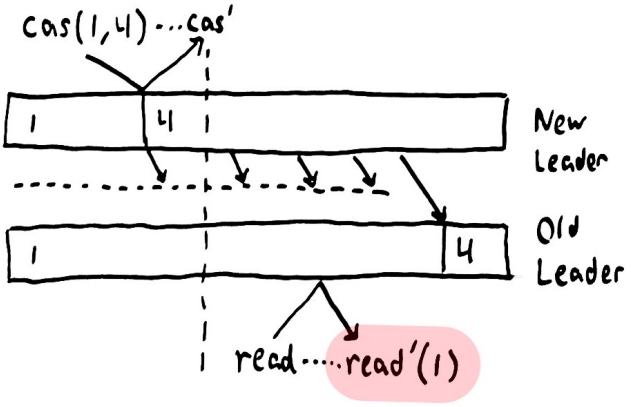
Looking at the history just prior to that failure, we see that process 4 wrote `1` to the register, and several processes read that value before the partition occurred. It looks like the value was `1`, a compare-and-set from `1` to `4` took place, but after that CAS completed, some process managed to read the *previous* value. In the consistency literature, this is called a *stale read*.

Stale reads are *bad*. They don't just violate linearizability–they violate sequential consistency, causal consistency, read-your-write, monotonic writes, monotonic reads–basically everything you'd want from a single-valued register goes out the window. This is particularly surprising because Raft, the etcd consensus algorithm, guarantees that committed log entries are linearizable.



But etcd's "consistent" reads don't go through the Raft log.

Instead, they simply return the local state if the current node considers itself a leader. But Raft says nothing about guaranteeing leader exclusivity: multiple nodes can consider themselves the leader simultaneously.



So imagine two nodes, separated by a network partition, have the value `1`. The node on top has just been elected leader for the most recent term, and accepts that CAS request, changing `1` to `4`. It's unable to propagate that change to the old leader because they're separated by a network partition. The old leader goes on happily replying to reads with the old value, until it realizes it hasn't received a heartbeat from a majority of peers in some time, and steps down.

Once the partition resolves, the old leader receives the new value `4` from the new leader, and the system continues on its way.

I want to be explicit, because some people have asserted that this behavior is "linearizable with respect to the Raft index", even if it isn't "linearizable in general". It's neither. An etcd "consistent read" can read a value from index 5, then index 4, then index 6, and so on. I think you might be able to recover sequential consistency by adding an FSM to the client that tracks the etcd index and tags all requests with a minimum-index constraint, but this is a.) optional, b.) not in any of the clients I know of, and c.) isn't linearizable anyway.

A note on Consul

Just after I found this bug in etcd, Hashicorp announced a new service-discovery project called Consul. A half-dozen people asked me what I thought of its design, and to my delight, its authors had already tested their system using Jepsen's etcd test as a template. They reported:

> As part of our Consul testing, we ran it against Jepsen to determine if any consistency issues could be uncovered. In our testing, Consul gracefully recovered from partitions without introducing any consistency issues.

This is not quite the whole story.

Jepsen actually *did* find a consistency issue. In fact, it found the same mistake that etcd made: **"consistent" reads in Consul return the local state of any node that considers itself a leader, allowing stale reads.** Their solution at the time was to change the leader timeout from 1 second to 300 milliseconds, side-stepping the race condition.

```
-       LeaderLeaseTimeout: time.Second,
+       LeaderLeaseTimeout: 300 * time.Millisecond,
```

Now, I've fought quite a few race conditions in my day, and adjusting the timeouts is a great nuclear option–but it doesn't really guarantee correctness. High IO utilization and blocking syscalls can introduce surprising delays into processes at runtime. VMWare vmotion will happily pause a process for seconds, as will garbage collection. Go's GC is not particularly sophisticated yet, and there are production reports of ten second garbage collection pauses. Bottom line: a seven-hundred millisecond pause is not gonna cut it. The *best* way to solve a race condition, in general, is to remove the time dependence from the algorithm altogether.

Future iterations of Jepsen may be somewhat more challenging with respect to clock assumptions.

Good news, everyone!

I've corresponded with both the etcd and Consul teams about this, and the emerging consensus is to implement three types of reads, for varying performance/correctness needs:

- Anything-goes reads, where any node can respond with its last known value. Totally available, in the CAP sense, but no guarantees of monotonicity. Etcd does this by default, and Consul terms this "stale".
- Mostly-consistent reads, where only leaders can respond, and stale reads are occasionally allowed. This is what etcd currently terms "consistent", and what Consul does by default.
- Consistent reads, which require a round-trip delay so the leader can confirm it is still authoritative before responding. Consul now terms this consistent.

Consul has, I believe, already implemented these changes, and written comprehensive documentation for the tradeoffs involved. Etcd is still in process, but I think they'll get to it soon.

The etcd and Consul teams both take consistency seriously, and have been incredibly responsive to bug reports. I'm very thankful for their help in getting both systems running, and for their care in finding good tradeoffs between latency and consistency. I'm very excited to see a spate of strongly-consistent systems emerging in the last couple years, and I look forward to watching both etcd and Consul evolve. It's a good time to be a software engineer!

In particular, I'd like to thank Xiang Li, Armon Dadgar, Evan Phoenix, Peter Bailis, and Kelly Sommers for their help in this analysis. A big thanks as well to Comcast, whose research grant made this round of Jepsen verification achievable. Y'all rock.

Next up: Elasticsearch.

Brian Olson, on 2014/06/12

It's a buggy Raft implementation that considers itself leader while in the minority side of a partition. (I just read the Raft paper this week.) A Raft node can be a *candidate* during partition, but doesn't get to be leader until it gets a vote from the majority of the cluster. Maybe there's a bug in their leader logic or in the cluster membership change logic? The Raft paper says that all client requests should go through the leader and reads should recognize only committed results (that have been accepted by a majority of the cluster). 'read anything' mode will obviously improve availability and throughput, but then it's not strict Raft anymore.

Aphyr, on 2014/06/12

> It's a buggy Raft implementation that considers itself leader while in the minority side of a partition.

Naw, this is fine. Raft allows multiple simultaneous leaders; the exclusivity invariant only holds for a given term. Leaders with different terms can run concurrently–and indeed, this is expected behavior during a partition.

This is why the appendEntries index+term constraint is so important–it guarantees that contemporary leaders don't give rise to conflicting committed entries in the log. :)

Randall, on 2014/06/14

Far afield from the topic, but can't not note:

> Go's GC is not particularly sophisticated yet, and there are production reports of ten second garbage collection pauses.

Later in the thread, the poster upgrades to a newer (but still 2012-vintage) Go with (partly) parallel GC and says "now GC duration reduced to 1~2 seconds". Apparently on a 16GB heap too–wowza don't try this at home kids.

Of course, your point is random long pauses happen and mess up logic that depends on timing, and obviously still true. Also still true Go doesn't do, say, generational GC like the JVM and loses to it in GC-focused benchmarks. Just noting the report at the top of that thread is not the latest.

mbonaci, on 2014/06/21

It was really wonderful to see how *etcd* maintainers were so responsive to your findings. To me, as a potential *etcd* user, the fact that they decided to not only change the docs, but also implementation details is very encouraging, despite some arguable comments down the thread. I'd like to be able to say the same thing for *Redis*, though.

Keep going Aphyr, you're doing remarkably important job here.

If I may, I'd like to suggest a couple of future "Jepsen adversaries", from different families:

Datomic Solr - everyone will want this one, now that you did ES :) Neo4j Infinispan Spark HyperLevelDB

Thanks, Marko

mbonaci, on 2014/06/21

The correct link for Solr is http://lucene.apache.org/solr/.

Post a Comment

Name

Email

Http

Supports github-flavored markdown for [links](http://foo.com/), *emphasis*, _underline_, `code`, and > blockquotes. Use ```clj on its own line to start a Clojure code block, and ``` to end the block.

Post Comment