Audience: general public

Purpose

This document attempts to sketch a conceptual design for a DSS that performs application-level consensus by drilling down into the necessary details, but hopefully without diving into unnecessary technical details.

Problem outline

For people without deep knowledge of ASTM F3548 and F3411, I believe we can simplify the conceptual problem without loss of important generality:

1. The system must perform CRUD operations on “operational intents” (OIs), which are data blobs keyed by publicly-visible ID
2. The system must perform CRUD operations on “subscriptions”, which are data blobs keyed by publicly-visible ID
3. When an OI is changed, a subset of subscriptions must be changed too
4. When an OI is changed, a subset of OIs must be known (and correspond with the change)

[Josh] Do we have a distribution (even rough speculation will do) on how much of the workload is reads vs writes?

[Ben] I would expect us to care most about high-volume USSs, and I would expect high-volume USSs to mostly maintain their airspace picture via subscriptions. In that case, the workload would be nearly 100% writes.

Questions

What consensus algorithm should be used?

Raft

(leader-led consensus)

Benefits

- Very well-studied and proven
- Has high-level algorithm implementations in Go
- Easier to understand and more well-defined than, e.g., Paxos

Drawbacks

- Reads must go through leader, so are likely slower than two-phase commit

[Ben] I think the only realistic option here is Raft.

Two-phase commit

A DSS instance positively ensures all other DSS instances have an update before that update is considered accepted.

Benefits

- Fast reads
- Easy to implement(?)
    - Need to describe handshake with other instances in more detail to understand behavior in the presence of network failures at various times
        - E.g., Node 1 sends an update to Nodes 2 and 3, Nodes 2 and 3 accept and acknowledge, but the acknowledgement from Node 3 is lost. How do the nodes respond to requests before some resynchronization process happens?

Drawbacks

- Have to define pool membership via separate mechanism (Raft?)
- Zero survivability naively – any DSS instance being down prevents writes
    - But, perhaps there could be a mechanism to automatically/quickly remove a DSS instance from the pool via the pool membership mechanism designed above

Leaderless consensus

CASPaxos, EPaxos, WPaxos, Atlas

Benefits

- Often somewhat faster writes than leader-led consensus

Drawbacks

- Highly complex, probably very difficult to implement
- Slower reads than leader-led consensus under contention

What data type is contained in log entries?

In other words, what are we coming to consensus on?

[Ben] Our current database comes to consensus on rather low-level logical data objects, many of which need to be modified to serve a particular request. But, it seems like our Raft log entries

could instead consist of a CRUD operation (on either an OI or subscription) and then each individual DSS instance would work out what that means for all the low-level logical data objects (e.g., mutation of subscription due to change in OI).

This increases the risk of inconsistency as a particular DSS instance may have faulty logic that works out low-level data object values incorrectly, but I think this risk is small and very mitigatable.

It adds a benefit: a follower could reject a log entry and refuse to incorporate it if it were invalid (even though the leader thought it was ok). In this way, a log entry would only be adopted if 1) a majority of DSS instances accept it and 2) the leader does not veto it.

[Josh] It's also a core requirement of how raft/other consensus algorithms - unless we're trying to guard against Byzantine faults (we shouldn't), we have to enforce the guarantee that each node arrives at the same end state given the same set of log entries

[Ben] Yeah, there is certainly no requirement to be Byzantine fault-tolerant. I agree we should be able to verify (and therefore assume) that each node arrives at the same end state given a log set, but this would hypothetically be conceptually easier if the log entries consisted of "key=value" rather than "perform X which affects many keys and values". I think the latter is reasonably doable, but it is easier to have problems than the former.

How do log entries get purged?

When we replicate a log entry for “create X” and then later replicate a log entry for “delete X”, how do we avoid unbounded build up of log entries?
[Ben] Servers can “snapshot” independently – that is, form a single state from all log entries up to a point, and then discard the log entries included in that snapshot. When using CRUD operations as log entries, snapshots would be a database of resulting logical entities (OIs and Subscriptions).

How would shards work?

In a naive Raft implementation, everything must go through the leader, so at any given time, performance is strongly limited to the performance of the leader, and the leader is equally likely to be any node in the group. In that case, how could one participant get better performance by improving just their own DSS instance’s latency?

They wouldn't

[Ben] One option could be to have every DSS instance 302 a client to the current leader. That wouldn't fix this problem, but at least there wouldn't be ongoing man-in-the-middle forwarding latency.

Other suggestions?

[Ben] I'm not sure what else we could do. Sharding is difficult because the data elements interact with each other – we can't shard on OI/subscription ownership, for instance, because a change in USS 1's OI probably requires a change in USS 2's subscription, and I can't imagine we want to attempt transactions that span multiple Raft groups.

Even if we established a good sharding mechanism, we would then need to figure out how a particular node could preferentially become the leader of that shard. I don't think that logic is included in the established Raft algorithm. Perhaps that part is as simple as setting the random leader election delay according to how much a particular node wants to be leader. In that case, how do we trigger new elections frequently enough that the preferred leader gets to be leader in sufficiently little time from an anomalous event, but not so frequently that it cuts into available cluster resources?

[Josh] One angle worth talking about would be to denormalize the data by S2 cell ID (at some level), and use that as the shard key. Similarly, establishing a consensus group per cell ID can help limit contention to geographically contentious regions.

[Ben] If we sharded by S2 cell ID, how would operations on objects that spanned multiple S2 cells work? It seems like we would need to have a second level of consensus between the two shards before the operation could complete (and be sure not to leave the DAR dirty if the operation completed successfully on one shard and not the other shard).

[Josh] Unfortunately all depends a lot on the specific mechanism as to how this plays out. Examples:

- Single-leader cluster wide - likely not a problem because the one leader is the one ensuring consistency of the S2 cells along with e.g. the flight

- Multi-leader (e.g. raft-per-shard(s)) - likely need something like a two phase commit or some other more complicated mechanism because each shard's leader needs to be coordinated. This is how spanner works under the hood when a txn spans multiple ranges.

- Leaderless - depends again on the mechanism. For example under CASPaxos (a promising option IMO) the S2 cell could be treated as "derived data" from the core data structure and "inherit" the ballot number of the most recent piece of core data to touch it, in which case there's no major implication

[Ben] Is "Single-leader cluster wide" essentially the same as Raft with no sharding?

> [Josh] Yeah, it's basically "this node is the leader for the whole DAR".

> It really only solves the "blast radius" problem, while still likely retaining the latency issues of the current system (and on average it's probably worse, and also requires bigger hardware such that one instance has to host the entire DAR)
>
> It's certainly easier to implement and reason about, so I mentioned it for completeness. I don't think it's really a viable option.
>
> > [Ben] It seems like this would be substantially worse than the current system (in terms of performance limit) since CockroachDB automatically shards for horizontal scalability.
> >
> > One DSS instance should already be hosting the entire DAR (survivability), but even just one node hosting the entire DAR is actually probably acceptable.

For Multi-leader (Raft-per-shard), isn't that effectively three layers of consensus? (Local consensus within a single DSS instance that chooses to horizontally scale by sharding its own data [like with CRDB], application-level per-shard consensus, and meta-consensus between shards)

> [Josh] It really depends on how you are wanting to define consensus, but sure.
>
> Note that the "local consensus" is solely an implementation detail of choosing to use CRDB, and is kind of orthogonal to this discussion. Furthermore, consensus algorithms (e.g. within CRDB) are very different beasts when they're within a single region with a locally connected network as compared to consensus over WAN. I'd go so far as to say they're two different "entities".
>
> Application-level WAN consensus is what I would say is the one consensus we're talking about, representing one organization's communication with another organization in a structured/controlled fashion.
>
> Finally, Two Phase Commit (2PC) is I guess a kind of consensus as well, but again it has a very different shape and profile compared to e.g. Raft - it's really more of a mechanism for implementing distributed transactions rather than reaching consensus. Depending on flight patterns and sharding schemes it may or may not come into play (e.g. imagine if the sharding key were a larger S2 cell that encompasses broader swaths of land - 2PC might only come into play on the "seams" between these big shards, but most flights in a metro area would fall within one shard or another)

For Raft, the etcd Raft implementation seems like a great starting point that we can "just" plug our message exchange and storage into – what is the simplest available similar starting point if we wanted to pursue Leaderless consensus?

> [Josh] Build it. Most of the algorithms only exist in the literature.

My current thinking is that CASPaxos can hit that sweet spot of simplicity without some of the WAN-based leader issues with Raft and co.

Here's a decent high-level read:
https://reubenbond.github.io/posts/caspaxos

FWIW I was able to (on my own time & hardware) get a working key-value store running using CASPaxos in about a days worth of effort (adapting https://github.com/peterbourgon/caspaxos) , and it's been pretty straightforward to understand.