CockroachDB mitigation approaches

Audience: general public

Purpose

Cockroach Labs is changing the license for CockroachDB such that USSs may not use later versions of CRDB without paying for a license. This is a substantial change from InterUSS's historical fully-open technology stack. This document identifies some potential mitigation approaches.

Constraints/Goals

When choosing an option below, the following design constraints and project goals will be considered.

Must-haves

1. DSS implementation meets ASTM F3548-21 requirements
    a. *DSS implementations shall (DSS0215) only respond to the USS after the transaction has been recorded in the DAR.*
    b.
2. DSS implementation meets ASTM F3411-22a requirements
    a. *The DSS shall (DSS0070) be implemented in a manner that allows a USS to access any instance of a DSS in a DSS pool and obtain the same results.*
    b. *3.2.18 DSS instance, n — for availability purposes, multiple synchronized copies of the DSS supporting a DSS region. Each copy is referred to as a DSS instance. USSs can interact with any DSS instance within a pool and switch over to any other instance in the event of a failure.*
    c. *3.2.19 DSS pool, n — a synchronized set of DSS instances where operations may be performed on any instance with the same result, and information may be queried from any instance with the same result. A DSS region will often have a production DSS pool along with one or more test or staging DSS pools.*
    d. *Whereas the DSS does not have explicit performance requirements, USS may rely on it for the following ones:*
        i. *NET0420*
3. DSS implementation can be developed and reliably maintained by the available InterUSS contributors
4. Reasonable deployment of DSS implementation is capable of handling roughly country-scale traffic in the medium term
5. The operator of a DSS instance can define which DSS instances are part of the pool in which it is participating

    a. (not necessarily by naming the other instances explicitly; accepting cryptographic certificates that validate against one or more accepted public keys would be one way of accomplishing this goal for instance)
6. Failure of one DSS instance in a pool (of at least three instances) allows the remaining instances in the pool to continue to function (continue to complete all queries without substantial increase in latency)
7. Deployment of DSS implementation is not tied to a single/specific cloud provider
8. Each DSS instance must be an equal peer, without more or less power/authority than the other DSS instances
    a. This does not preclude temporary circumstances such as Raft leaders where each instance was, a priori, equally likely (subject to some set of criteria) to take on the asymmetric role
9. Implementation can be explained to semi-technical regulators and maps to conceptual DSS functionality described in ASTM standards
10. USS use of the InterUSS DSS implementation must allow the USS to meet performance requirements

Desirable characteristics

1. DSS implementation does not depend on specific non-free/restricted-license products
2. Failure of a minority of DSS instances in a pool allows the pool to continue to function
3. Ongoing maintenance of a DSS instance deployment (including necessary coordination with maintainers of other DSS instances in the pool) does not require a large amount of effort or specialized expertise
4. Initial deployment of a DSS instance does not require a large amount of effort or specialized expertise
5. It is difficult for a member of a particular DSS instance provider to accidentally corrupt the DAR (DSS Airspace Representation – conceptual “database” of shared airspace picture)
6. A DSS instance provider can unilaterally improve DSS performance by contributing more resources (e.g., more VMs, better VMs, etc)
7. Deployment of a low-traffic DSS instance is relatively inexpensive
8. Implementation is easy to explain to semi-technical regulators and straightforwardly maps to conceptual DSS functionality described in ASTM standards

Nice-to-haves

1. DSS implementation can interoperate with non-InterUSS DSS implementations with a small-as-possible set of constraints the non-InterUSS DSS must satisfy
2. Slow failure of any number of DSS instances in a pool of at least three instances allows the pool to continue to function
    a. E.g., starting with 7 instances, 5 instances fail at a rate of 1 per week – after 6 weeks, there is still a functioning 2-instance pool

Accept commercial terms

One possibility could be to continue use of CRDB in InterUSS and acknowledge that InterUSS itself and all USS users will have to obtain, and likely pay for, one or more CRDB licenses to continue to use the InterUSS DSS implementation.

Pros

- Requires the least amount of development effort (probably)
- Likely results in easier access to paid Cockroach Labs support for users

Cons

- Increased operating cost for InterUSS (we have to buy our own licenses)
    - Indicative pricing of $450 per vCPU for persistent deployments
- Increased operating cost for all InterUSS users
    - Indicative pricing of $1800 per vCPU for production deployments + $600 per vCPU for non-prod (qual-partners) deployments + $450 per vCPU for dev deployments
- Effective endorsement of a commercial product (could be a strategic risk in interactions with regulators and prospective user support)
- License management activities will take time and resources
- Database currently exposed to the public internet
- Blast radius - single database means local mistakes/hacks become global problems
- Coordination - multiple organizations responsible for single database means less clear/more complex responsibility

Use a different database

The “store” functionality should be relatively-well abstracted from the mechanism (CockroachDB), so hypothetically it should not be too difficult to add support for an alternative database with similar capabilities.

Pros

- Hypothetically not a huge amount of development effort
- Low risk that synchronization will behave and perform as intended

Cons

- Have to find and evaluate a new database
- Have to develop support for using this database in DSS application server
- Have to develop support for deploying this database

- Users will likely need to completely tear down their current DSS deployments to switch
- Database may be exposed to the public internet
- Blast radius - single database means local mistakes/hacks become global problems
- Coordination - multiple organizations responsible for single database means less clear/more complex responsibility

Historical comparison

This comparison was produced circa 2019 when first using CockroachDB as the backing database.

Database	Distributed	Consensus Protocol	Optimistic or Pessimistic Transactions	Transaction Consistency Level	License	Hosting	Management	Scalability	Future Flexibility	Price	Subscriptions
Blockchain	Yes	Blockchain	Optimistic	Eventual Consistency	Open Source	Private	Easy, P2P	High	Limited	Free	No default
Zookeeper	Yes	ZAB	Optimistic	Strong KV consistency	Apache 2.0	Private	Single Control Plane	O(10) Gb storage	Limited	Free	No default
Etcd	Yes	Raft	Pessimistic	Strong KV consistency	Apache 2.0	Private	Single Control Plane	O(10) Gb storage	Limited	Free	No default
CockroachDB	Yes	MultiRaft + Timestamps	Pessimistic	Serializable Isolation	Hybrid	Private & Managed	Single Control Plane	Horizontal Scaling	High	Free	Change Data Capture
TiDB	Yes	MultiRaft	Optimistic	Serializable Isolation	Apache 2.0	Private	Single Control Plane	Horizontal Scaling	High	Free	No default
Spanner	Yes	Paxos + TrueTime	Pessimistic	External Consistency	Closed Source	Managed Only	Single Control Plane	Horizontal Scaling	High	High	Commit Timestamps
Firestore	Yes	Paxos + TrueTime	Optimistic	External Consistency	Closed Source	Managed Only	Single Control Plane	Horizontal Scaling	High	Free Tier	Realtime Updates
PostgreSQL	No	None	Pessimistic	External Consistency (no replication)	PostgreSQL License	Private & Managed	Single Control Plane	O(100k) writes per second	High	Free	Postgres Subscriptions
DynamoDB	Yes	Paxos	Optimistic	Strong KV, enventually consistent queries	Closed Source	Managed Only	Single Control Plane	Horizontal Scaling	High	Free Tier	DyanmoDB Streams
Yugabyte	Yes	MultiRaft	Optimistic	Serializable Isolation	Apache 2.0	Private & Managed	Single Control Plane	Horizontal Scaling	High	Free	No default
FoundationDB	Yes	?	Optimistic	Serializable Isolation	Apache 2.0	Private	Single Control Plane	Horizontal Scaling	High	Free	No default
FaunaDB	Yes	Raft	Pessimistic	External Consistency	Closed Source	Managed Only	Single Control Plane	O(100k) writes per second	High	Moderate	No default

rqlite

rqlite is advertised as sqlite (a very simple database implementation that uses a single file to store the database information), but with Raft consensus (same general approach as CRDB). This means we can add an arbitrary number of instances that will automatically synchronize with each other, similar to CRDB.

Pros

- Friendly MIT license
- Simple; likely easy to understand and use
- Documentation seems excellent
- Very similar overall paradigm to CRDB, so less switching overhead

Cons

- Does not appear to support dynamic sharding
    - Therefore every node will need to contain the entire database
    - This is unlike CRDB where a single DSS instance can shard its copy of the entire database across multiple nodes (currently 3)
    - This is a potential issue for scalability in the future, though it is unclear whether this will be a binding constraint in the medium term
    - This issue can be mitigated by having separate databases for F3411-19, F3411-22a, and F3548-21 (which is probably a good thing conceptually)
- Database may be exposed to the public internet
- Blast radius - single database means local mistakes/hacks become global problems

- Coordination - multiple organizations responsible for single database means less clear/more complex responsibility

Research required

- Performance versus other options
- Support for the type of query we're using for S2 cell IDs
- Support for the multiple CAs we need in order to enable cross-organization authentication
- Verify dynamic sharding behavior/capabilities

YugabyteDB

YugabyteDB was in consideration initially as an alternative to CockroachDB. At that time, Sean Teeling indicated its only downside (see above table) as "optimistic transactions".

Pros

- Appears to be a more fully-featured database (like CRDB) than rqlite
- Matching Apache 2 license to InterUSS

Cons

- Appears to be more complex than rqlite (more room for unknowns or unexpected behavior, like the issues we have historically had with CRDB)
- Database may be exposed to the public internet
- Blast radius - single database means local mistakes/hacks become global problems
- Coordination - multiple organizations responsible for single database means less clear/more complex responsibility

Research required

- Determine whether optimistic transactions would prevent us from satisfying any of the ASTM requirements, or any desirable characteristics it's important for the system to have
- Verify that a single logical copy of the database can be sharded across multiple nodes within a single failure zone (DSS instance)
    - Appears to be the case

Vitess

Vitess is a database solution for deploying, scaling and managing large clusters of open-source database instances.

Pros

- Supports MySQL
- Can run public or private cloud architecture
- It includes compliant JDBC and Go database drivers using a native query protocol.
- License: Apache 2.0 - Graduated project of the Cloud Native Computing Foundation

Cons

- Isolation level: No serializable transactions. It only offers REPEATABLE READ for single-shard transactions and multi-shard transactions the semantics change to READ COMMITTED.
- Multiple services are required to run the database in operations which makes the system complex to operate and support. Indeed, Vitess is an overlay on top of MySQL servers with multiple services used for query dispatch and orchestration. Based on our experience with CockroachDB, we are looking for simplicity to limit frictions for new users.
- Cross-region sharding isn't a well-lit path (their model thinks in cell terms)
- Database may be exposed to the public internet
- Blast radius - single database means local mistakes/hacks become global problems
- Coordination - multiple organizations responsible for single database means less clear/more complex responsibility

TiDB

TiDB is an open-source distributed SQL database that supports Hybrid Transactional and Analytical Processing (HTAP) workloads.

It is built on top of TiKV (graduated CNCF project). The project is an open source product (Apache 2.0) by PingCap, similar to CockroachDB.

Pros

- Either Optimistic or Pessimistic transactions

Cons

- Non-negligible chance of licensing change at some point similarly to CockroachDB.
- TiDB supports the following isolation levels: `READ COMMITTED` and `REPEATABLE READ`. `No SERIALIZABLE`.
- Database may be exposed to the public internet
- Blast radius - single database means local mistakes/hacks become global problems
- Coordination - multiple organizations responsible for single database means less clear/more complex responsibility

Use a key-value store

Instead of using a full traditional database, we could switch (back) to using a distributed key-value store since we may not need all the features of a full database. Before 2019, this is exactly how the predecessor to the current DSS implementation worked, using Zookeeper as the key-value store. Going forward, TiKV might be a good option.

Pros

- Less complexity than, e.g., CockroachDB and YugabyteDB, so perhaps fewer opportunities for unexpected behavior that we have observed with CockroachDB
- Conceptually, a distributed key-value store better matches the DSS Airspace Representation model than a single database (which prompts most readers/viewers to think of a monolithic central entity)
- Potentially more performant that databases if databases have overhead to support features we don't need

Cons

- More questionable scalability than database products designed for the kind of scale that we need (and more) – would require research to verify scaled performance
- May require a larger rewrite of behavior to eliminate the database paradigm
- Underlying key-value store may be exposed to the public internet
- Blast radius - single "database" means local mistakes/hacks become global problems
- Coordination - multiple organizations responsible for single "database" means less clear/more complex responsibility

Distributed NoSQL candidates to be evaluated:

- TiKV
- Etcd
- RethinkDB

Application-level consensus

See DSS: Application-level consensus

InterUSS - Application-level consensus alternative to Cockroach DB using Raft

Raft

Raft is a well-studied and proven consensus algorithm and we could use a Raft implementation library directly to synchronize information between DSS instances.

Pros

- Conceptually, the fact that we have a single database leads people to think about the DSS as a monolithic central entity. If we replaced that database with a mere synchronization protocol, the resulting system would be easier for people to understand correctly
- Establishing a synchronization protocol could allow that protocol to be standardized in F3548 and eliminate any concerns about the practicality of DSS instance interoperability
- Having a standard synchronization protocol would enable co-operation between very diverse DSS instance implementations (even ones with different within-instance data stores)
- The boundary of each DSS instance's own copy of the data is clearer than when sharing a single DB
- Potentially lower cross-organization coordination
- Backing datastore not exposed to public internet
- Blast radius: issues with a single organization's DSS instance are somewhat less likely to propagate to the rest of the cluster (i.e., SQL commands affecting consensus group are not available)

Cons

- Large development effort with relatively high risk of failure or suboptimal outcome compared to other options
- End result is unlikely to be as performant as a purpose-built database solution