

DSS Raft Store V1 Technical Design Document

Design: v2

Status: Review

Audience: General Public

InterUSS Github Issue: <https://github.com/interuss/dss/issues/1639>

Change Log

This section summarizes significant changes to the draft.

Date	Version	Description
2026-09-16	1	Initial version
2026-09-18	2	Requirements on similar CPU architecture have been relaxed by moving possible unreliable operations to the proposal instead of the apply path.

Summary	4
Glossary	5
Airspace and standards	5
System	5
Consensus and storage	6
Context & Objectives	8
Shared airspace and the coordination challenge	8
Historical background	8
Objective	9
Goals	9
Functional requirements	9
Non-functional requirements	10
Non-Goals	11
Alternatives considered	11
System Design & Architecture	13
High-Level Architecture Overview	13
Raft overview	15
Components Breakdown	15
Infrastructure requirements	15
Core-service detailed technical design	18
Pool Membership	27
Trust	28
Bootstrap and joining	30
Learners and voters	30
Removing a node	31
One change at a time	31
Who may propose a change	31
APIs	32
Public API	32
Consensus API	32
Storage API	33
Admin API	34
Multi-Version Interoperability	34
Behaviour changes	35
Options introducing new behaviour	37
Compatibility window and downgrade	38
Admin User Interface	39
Quality Attributes	40
ACID properties	40

Atomicity	40
Consistency	40
Isolation	40
Durability	40
Linearizability	41
Fault tolerance	41
Horizontal scalability	41
Cross-Cutting concerns	42
Security	42
Authentication and Authorization	42
Confidentiality	43
Integrity	43
Availability	43
Observability & Telemetry	44
Metrics	44
Web interface	45
Testing	45
Applicative side	45
Pooling	45
Operations, Rollout & Configuration	47
Configuration	47
Parameters	47
General configuration	48
Eviction of expired entities	48
Node lifecycle	48
Restarting more than one node	48
Rollout	49
Validation	49
Upgrade to raft	49
Upgrades	50
User documentation	50
Initial performance evaluation	50
Risks and Mitigations	51
Future Considerations	53
Conclusion	54
References	55
Standards	55
Consensus	55
Databases evaluated	55
InterUSS	55
Libraries	56

Summary

The DSS is the shared index that lets competing UTM service providers coordinate in the same airspace. It stores no flight data. It stores references, bounded in space and time. A provider queries the DSS to learn who else is active in a volume, then contacts those providers directly. Every node in a pool must return the same answer, and the ASTM standards require a write to be rejected if the state it was based on has changed. The DSS therefore needs linearizable reads and writes across a pool operated jointly by competitors trusting each other, none of which is authoritative.

Today that guarantee comes from a database cluster shared by all nodes in the pool: CockroachDB, or YugabyteDB as an open alternative. This has two problems. CockroachDB's 2024 licence change means later versions cannot be used freely and may be unaffordable for our users, which is incompatible with InterUSS's open-source basis. YugabyteDB proved unsuitable for performance-sensitive deployments and Cockroach has recently also shown some limitations.

This document proposes replacing the shared database with consensus implemented directly between DSS nodes, using the Raft protocol and the etcd Raft library. Each instance keeps its own local storage. Requests are ordered by Raft as replicated proposals and then applied independently and deterministically by every node. The only deployable component becomes the core service. The ASTM REST API that providers call is unchanged, as is the authentication model for that traffic.

A proof-of-concept implementation passes the full DSS test suite, including `uss_qualifier` and `prober`. Measured on a 9-node pool with emulated country-scale latencies, it outperformed the distributed SQL implementation in every scenario: about 95 QPS on the single-cell scenario against a 2 QPS target, about 80 QPS on the multi-cell scenario against a 45 QPS target, latency under 300 ms against a 5 s bound, and no errors against a 1% bound ([#1594](#)). The 24-node requirement has not yet been measured.

The proof of concept has passed the first two of the three gates set for this effort: functional practicability and performance fitness ([#1633](#)). This document is the design for the third, production readiness. It covers the architecture and the detailed design of the store, consensus and storage layers; the interoperability rules that allow a pool to run mixed versions during an upgrade; and the security, observability, testing and rollout implications.

Glossary

Airspace and standards

- ASTM: ASTM International, the standards body publishing the UTM standards this system implements.
- F3411-22a: ASTM standard for Network Remote Identification. Defines identification service areas and subscriptions.
- F3548-21: ASTM standard for UTM Strategic Conflict Detection. Defines operational intents, constraints and subscriptions.
- UAS: Unmanned Aircraft System
- UTM: UAS Traffic Management. The distributed coordination model for uncrewed aircraft. In federated UTM, competing commercial providers coordinate with each other rather than through a central controller.
- USS: UAS Service Supplier. A provider serving its own operators and holding its own flight data.
- USSP: U-space Service Provider. The European equivalent of a USS. Used interchangeably with USS in this document.
- DAR: DSS Airspace Representation
- RID: Network Remote Identification. The DSS service implementing F3411-22a.
- SCD: Strategic Conflict Detection. The DSS service implementing F3548-21.
- AUX: The DSS auxiliary service, exposing pool status and version information.
- ISA: Identification Service Area. A volume of space and time in which a USS holds Remote ID data.
- Operational intent reference: A record that a USS has claimed usage of a volume of airspace for a period of time. It references the managing USS and does not contain the flight plan itself.
- OVN: Opaque Version Number. The token a USS must present to the DSS when writing a relevant entity, proving it did so with an up-to-date view of the airspace. A stale OVN causes the write to be rejected by the DSS.
- Subscription: A standing request to be notified of changes in a volume.

System

- DSS: Discovery and Synchronization Service. The shared index through which USSs discover each other's activity in a volume of airspace.
- DSS pool: The set of DSS instances that together serve one airspace and hold one synchronized view of the data.
- DSS instance: the whole DSS deployment one organisation operates and is responsible for. An instance runs one or more DSS nodes.
- DSS node: One core service deployment. It's a member of three Raft groups (aux, rid, scd). In each Raft group it is a voter, with its own node ID, WAL and memstore.

- Core service: The DSS server process. Under this design it is the only deployable component.
- Gate: One of the three measured milestones of the Raft V1 effort: Gate 0 functional, Gate 1 performance, Gate 2 production readiness (#1633).
- InterUSS: (InterUSS Platform) The open-source project maintaining the DSS implementation.
- TSC: InterUSS Technical Steering Committee.
- CRDB: CockroachDB. The distributed SQL database used by the current implementation.
- YugabyteDB: Distributed SQL database evaluated as an open alternative to CockroachDB.
- USS Qualifier or `uss_qualifier`: Automated conformance test tool from the InterUSS monitoring repository.
- Participant: The organisation operating one DSS instance. Used interchangeably with "DSS instance operator".
- prober: Older integration test suite that only tests a DSS deployment.
- S2: Library decomposing the surface of the Earth into cells, used to index airspace volumes.
- netem: Linux network emulation, used to reproduce realistic latency and loss in performance tests.
- CA: Certificate authority. The entity issuing and signing certificates. Each participant operates its own and issues one certificate per node.
- Mutual TLS: A TLS connection in which both ends present a certificate and each verifies the other's. Node-to-node traffic uses it; the ASTM API does not.
- Trusted CA list: The certificate authorities a node accepts on its Raft listeners. Local configuration, held by every node, not replicated.

Consensus and storage

- Raft: Consensus protocol by which a group of nodes agrees on an ordered log of entries despite failures of a minority of its members.
- etcd Raft library: The Go implementation of Raft used here. It implements leader election, log replication and safety, and leaves transport and persistence to the caller.
- Raft group: One set of nodes running one instance of the protocol over one log. This design uses three, one per service.
- Leader, follower: The node currently proposing values, and the nodes replicating from it. Leadership is temporary and carries no standing privilege.
- Voter, learner: A member that counts towards quorum, and a member that receives the log but does not vote.
- Quorum: A majority of voters. A group of $2n+1$ voters tolerates the loss of n nodes.
- Proposal: The unit of replication in this design: an encoded request plus the metadata needed to execute it identically on every node, including a timestamp and a random seed.

- Commit, apply: A proposal is committed once a majority has durably persisted the entry in the Raft log, and applied when a node executes it against its own data.
- Determinism: The requirement that applying the same proposal produces the same state on every node, so that the replicas do not drift.
- Store: The generic storage interface; implemented by the SQL backend, the Raft backend and the in-memory backend.
- raftstore: The Raft implementation of the store interface. Replicates through consensus and applies to the memstore.
- memstore: The in-memory data store holding the applied state. It performs no replication or persistence of its own.
- WAL: Write-ahead log. The on-disk record of Raft entries, written before they are applied, used to rebuild state after a restart.
- Snapshot: A serialized copy of the applied state, stored by the consensus layer to bound log growth and let new or lagging nodes catch up.
- Linearizability: The property that every operation appears to take effect at a single point between its start and its completion, so that a read always reflects all writes that completed before it.
- dssctl: Command-line tool for administrative tasks such as membership changes and option toggles.

Context & Objectives

Shared airspace and the coordination challenge

Air traffic control does not scale to uncrewed traffic. A traditional Air traffic controller manages tens of aircraft over voice radio, each with a pilot who can see and avoid. Uncrewed operations are numerous, low, short, and have nobody on board. UTM distributes the coordination across competing commercial providers instead, each serving its own customers and managing its own flights.

The DSS is the shared index that lets them coordinate. It holds no flight data. It holds references: identification service areas and subscriptions as defined under ASTM F3411-22a, operational intent references, constraint references and subscriptions as defined under F3548-21. Each reference is bounded in space and time. It also references the USS that owns it. A USS queries the DSS to find which other USSs have activity in a volume, then contacts them directly.

Each DSS instance in a pool may be operated by a different organisation, which is typically a participating USS or a DSS technology provider. A USS can query any node and must get the same answer. No organisation ever holds a privileged position in the pool. While a single central operating organization would be technically simpler, it is not an option within the federated design space. Because the DSS gates airspace access, a sole operating organization could unfairly block competitors, and a system crash would ground all providers simultaneously. As a result, the DSS operates as a decentralized network of equal participants, where data is perfectly synchronized across all nodes without relying on a central master server.

Historical background

In late 2024, Cockroach Labs changed the CockroachDB licence such that later versions are no longer released under an open-source licence. This broke InterUSS's fully open technology stack.

The Technical Steering Committee (TSC) [investigated](#) four options: accepting CockroachDB's commercial terms; migrating to another distributed SQL database (rqlite, YugabyteDB, Vitess, TiDB); adopting a distributed key-value store; and moving consensus into the DSS application itself. On the last track, CAS Paxos was rejected for its dormant ecosystem, absence of a reusable library and implementation-sensitive correctness, leaving Raft through the etcd/raft library as the only realistic algorithm. A full redesign was attempted but found no common ground, notably on the storage backing the consensus engine, and the proof-of-concept attempts of the time did not succeed, in part because etcd/raft was not yet available as a standalone module.

The TSC therefore prioritised YugabyteDB (delivered from v0.19) and agreed to revisit the decision if a contributor produced a working DSS prototype on etcd/raft. YugabyteDB has since proved unsuitable for performance-sensitive deployments.

More recently, research activities and user feedback provided valuable insights on what the DSS would require to support actual use cases and revealed performance limitations with CockroachDB as well, accelerating the need for an alternative solution.

Deep analysis of root cause of issues showed that usage of SQL (including transaction) lead to sub-optimal and very hard to improve performance wall, dues to how data/transactions have to be represented to communicate with the database, and how the database need to execute those transactions in order to guarantee required properties (at least for CockroachDB).

The revisit condition was met in February 2026 by an EPFL master thesis (see References) that delivered a working single-Raft DSS prototype on etcd/raft: writes applied only after majority acceptance, 300–400 QPS sustained on three nodes, a leader crash absorbed with a 20 % throughput dip and about 25 s of recovery, and linear latency under injected network delay. The same work prototyped a multi-Raft variant, discussed under Future Considerations. The packaging blocker of 2024 is gone: the implementation imports etcd/raft and reuses etcd's rafthttp transport, WAL and snapshotter directly.

Objective

Design and implement a synchronization protocol for the DSS such that it can handle the adoption and the traffic growth the industry will require in the coming years.

Goals

Functional requirements

The implementation must pass the full DSS test suite including the uss qualifier and the prober, which covers ASTM compliance, especially F3548-21 and F3411-22a requirements satisfied by the current SQL-backed implementation including strong consistency and linearizability guarantees, in particular DSS0215 (F3548) and DSS0070 (F3411).

The ASTM REST API interface used by the USS to interact with the DSS must remain unchanged.

Non-functional requirements

Performance targets

The targets below assume load shedding, so that the DSS operates within its intended regime with room to grow.

Across all scenarios: p95 latency under 5 seconds. This fits general ASTM standard metrics requirements. Additionally, error rate under 1%.

Test setup: All measurements to date were taken on a 9-node pool of three instances with three nodes each, all voters, with network conditions emulated to represent a country-scale deployment (table below). The 24-node requirement under Pool size and governance has not yet been measured but 24-node runs are planned for production readiness.

Path	netem configuration
Between instances	delay 16.5ms 2.5ms distribution paretonormal loss 0.025% 25%
Within an instance	delay 600us 40us 25% distribution normal loss 0.0005%

Country-scale targets

These targets match what is expected of the current CockroachDB-based deployment, defined by users of the US UTM implementation for SCD; a similar target is assumed for RID. Assuming so is conservative: for a given flight, it is expected for RID operations to be subject to less contention than for SCD operations because of the need to deconflict SCD operational intents. Meeting them establishes that this change introduces no regression.

- More than 2 QPS on a single site, 2 concurrent users, flights of 1 minute and 100 m
- More than 45 QPS across 116 sites, 40 concurrent users, flights of 4 minutes and 10 km

Pool size and governance

- Every DSS instance is an equal peer: none holds a permanent privileged position, and any node may hold the temporary Raft leadership.
- A pool functions correctly with 24 nodes.* Measurements to date cover 9 nodes (see Performance targets).
- Participants control pool membership. Adding or removing an instance is a governance decision and can be executed by any Pool participant after the new participant has been accepted by all existing participants.

* Input provided during InterUSS User Focus Group meeting in-person on July 8, 2026.

Survivability

- A pool tolerates the loss of a minority of its members without substantial latency impact.

Portability

- No dependency on any cloud provider.
- No dependency on any product under a restricted licence.

Simplicity

- Maintainable by the contributors InterUSS has available.
- Explainable to semi-technical regulators.
- Low deployment and operational effort. An operator should not need to run or tune a database.

Desirable

Not required, but valuable if achieved:

- Resistance to accidental corruption of the DSS Airspace Representation.
- The ability for an operator to improve its own performance unilaterally, without coordination with the rest of the pool.
- Deployment of a low-traffic DSS instance is relatively inexpensive

Non-Goals

- Changes to the ASTM REST API interfaces exposed to USSs.
- Changes to the authentication and authorization model for USS-to-DSS traffic, which is unchanged by this design.
- Challenging the correctness of the etcd implementation and the Raft protocol itself.
- Partitioning data across multiple Raft groups for horizontal scalability. The single-leader bottleneck is accepted for v1 and discussed in future considerations.

Alternatives considered

Four options were evaluated in 2024:

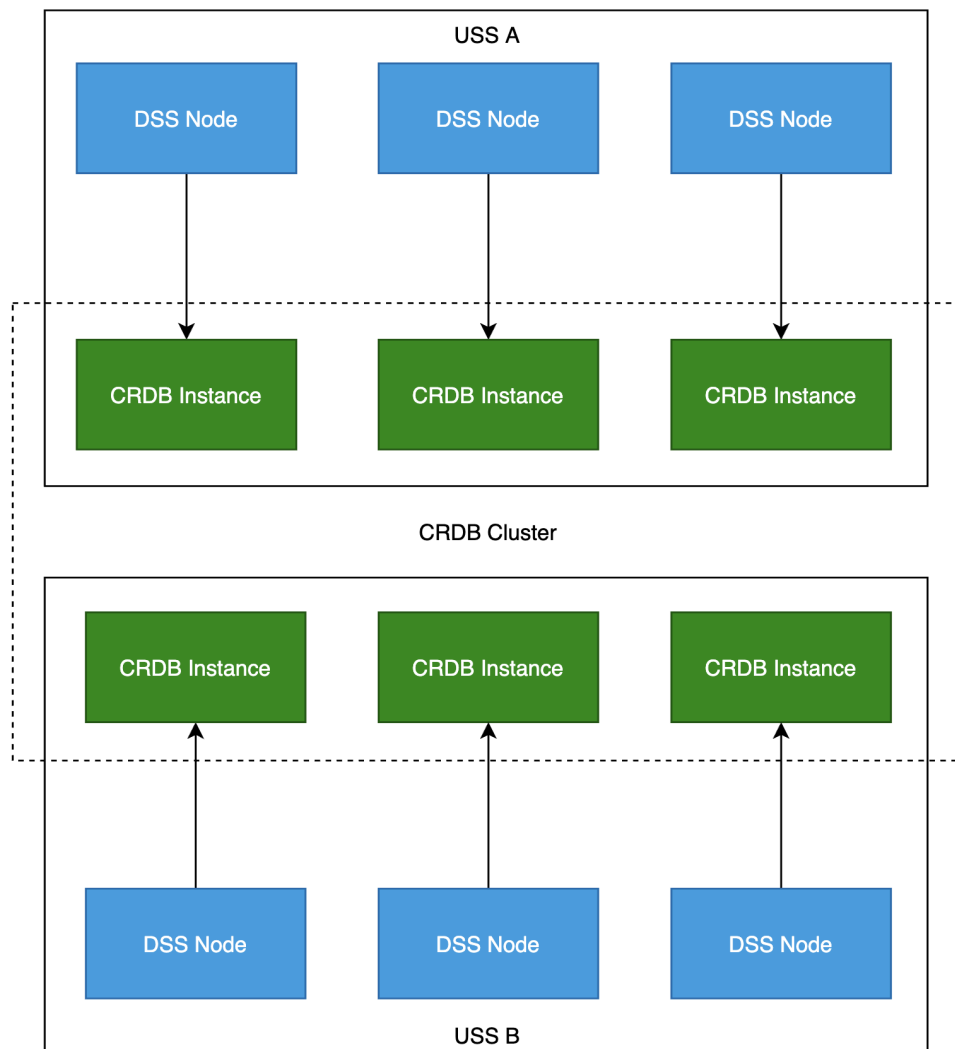
1. Accepting CockroachDB's commercial terms: rejected, a licence cost is incompatible with InterUSS's open-source basis.
2. Another distributed SQL database (rqlite, YugabyteDB, Vitess, TiDB): keeps the shared-database architecture and its root problem, that each DSS operation must be translated into a multi-statement transaction which the database then re-serialises; YugabyteDB was tried and proved unsuitable for performance-sensitive deployments and CockroachDB showed limitations since initial evaluation.

3. A distributed key-value store: in general, similar concerns as for any distributed SQL database. Strongest contender would be etcd, which however does not support multi raft groups, preventing horizontal scalability in the future.
4. Application-level consensus: removes any external product, confines each node's responsibility to its own storage, and maps one DSS request to one proposal. As part of the application-level consensus option, CAS Paxos was rejected (dormant ecosystem, no library, implementation-sensitive correctness). Raft was deferred in 2024 because etcd/raft was not yet a standalone module and the redesign found no agreement on storage; both blockers have since been removed (see Historical background).

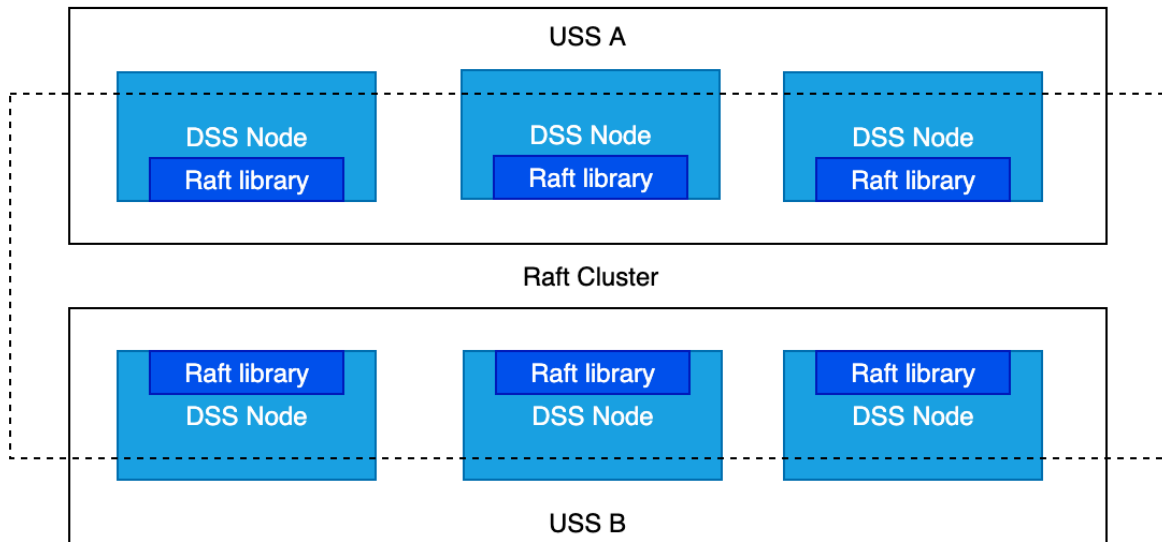
System Design & Architecture

High-Level Architecture Overview

To meet the goals above, this design replaces the shared database with consensus run directly between DSS nodes, using the Raft protocol as implemented by the etcd Raft library. Readers unfamiliar with Raft will find the terms used below under Raft overview.



Current architecture. DSS Nodes of every participant read and write to one CockroachDB cluster, whose nodes replicate among themselves over a database protocol on a dedicated port.



Proposed architecture. DSS Nodes hold their own data and replicate directly with each other over Raft. No external database.

From an external point of view, USS services calls the same ASTM REST API on whichever node it chooses, with the same authentication, and gets the same answer from every node.

The deployable unit is the core service alone. There is no database to run, tune or expose. Each node keeps its data in memory, persists its own write-ahead log and snapshots on local disk, and restores from them on restart. A participant runs one or more nodes; the pool is the set of all participants' nodes.

Nodes agree on the order of requests through Raft. A request arriving at any node is forwarded to the current Raft leader, which appends it to a log replicated to every node; once a majority has written it to disk, every node executes it, in log order, against its own copy. Each node reaches the same state because each applies the same requests in the same order, and because execution is deterministic: everything a node cannot derive for itself, such as the request time and the seed for generated identifiers, travels with the request.

Each of the three services, AUX, RID and SCD, runs its own Raft group over its own data. A DSS node is a member of all three when enabled. The groups are independent, so a service can be disabled without affecting the others.

Participants remain equal peers. Any node can be leader for a time; leadership orders requests and carries no other right. Any participant may propose adding or removing a node.

Node-to-node traffic runs over mutual TLS, authenticated against the CAs the participants have agreed to trust, on ports that replace the database port the current architecture exposes.

Raft overview

Readers familiar with Raft can skip this section. It defines the terms the rest of the document uses.

In order to achieve linearizability, nodes that hold copies of the same data have to agree on the order in which changes are applied. With the same starting state and the same changes, we get the same results. Consensus protocols settle that order despite crashed nodes. Raft is one, published in 2014, and it runs today inside etcd and inside CockroachDB and Yugabyte.

Raft keeps a log: an ordered sequence of entries, identical on every node. One voter is the leader; the others are followers. Only the leader appends. A request arriving at any node is forwarded to the leader, which assigns it the next position and sends it to all.

An entry is committed once a majority of voters have written it to disk. $2n+1$ voters tolerate n failures. A minority cannot commit, so a partitioned group cannot diverge. Learners receive the log without counting towards the majority. This design uses learner status while a node joins, so that it catches up before it counts.

A committed entry is then applied: each server executes it against its own data. Commit is a group property; apply is local. Raft guarantees the same entries in the same order on every server. It guarantees nothing about what applying does. That is the application's responsibility.

When the leader fails, followers miss its heartbeats and elect another within the election timeout. Only a node whose log is at least as complete as a majority's can win, so a committed entry survives any change of leader. Leadership orders proposals and nothing more; it moves on failure and any voter can hold it.

The log is bounded by snapshots. A node periodically writes its applied state and discards the entries it covers. A restarting node loads its snapshot and replays what follows. A new node, or one too far behind, receives a snapshot from the leader instead.

Components Breakdown

Infrastructure requirements

A DSS node is one process with local disk. It is not tied to a cloud-specific service. What follows is what a participant provides per node.

While establishing strict infrastructure requirements is still an open point, we can define some upper bounds for most of them. We assume that the computations complexity is necessarily below the one required by CockroachDB on the grounds that CockroachDB relies on etcd/Raft as well for its consensus layer, but additionally has to deal with all of the SQL constraints it

implements. We can then take the CockroachDB requirements as an upper bound, which roughly corresponds to the [etcd 'medium/large cluster' example configurations](#).

For disk and memory usage we can additionally compute some theoretical upper bounds given some assumptions.

Memory & disk usage

For the below calculations we use the anticipated figure of 1.6M flights per day. This shows that the memory and disk usage scales well and remains manageable even with very large loads.

Dataset

Given 1.6M flights/day and a reasonable flight duration upper bound of 20 minutes, we can assume an average of 23k active flights at any given time. With a peak-average ratio of 3.5 we can expect up to 80.5k currently active flights at any given time.

Given a reasonable flight area upper bound of 100km², and given an estimate of 1 subscription per flight per SCD/RID, we can expect a reasonable upper bound of 10kB memory usage per flight:

- RID ISA: 2kB
- RID subscription: 2kB
- SCD operational intent: 4kB
- SCD subscription: 2kB

Amounting to 805MB of memory usage, which we can round up to 1GB. We double it twice (to account for metadata and leeway for the garbage collector, and to account for the size of the in-memory snapshot) up, leading us to a safe minimum for the dataset memory usage of 4GB.

Serialized on disk, it would take up to 1GB.

Proposals

A configurable number of proposals are kept in memory and the WAL contains the ones persisted on disk. Total memory and disk usage is highly dependent on the configuration being used. Considering a flight whose geometry has an area upper bound of 100km² and is in conflict with up to 100 other flights, the following proposals are expected:

- RID create ISA: 2kB
- RID delete ISA: 1kB
- RID create subscription: 2kB
- RID delete subscription: 1kB
- SCD put operational intent x3.4 ([rationale](#)): 14kB
- SCD create subscription: 2kB
- SCD delete subscription: 1kB

Rounding this up for some margin, we can expect 30kB of proposals per flight.

Assuming the WAL is compacted every hour, and that there are up to 250k flights in the last hour (1.6M flights/day, peak-average ratio of 3.5), and rounded up for some additional margin, we can expect up to 10GB of disk usage.

And given the above distribution, a rough average of 3kB per proposal is expected. Combined with the default number of proposal to keep in memory of 100000, this amounts to 300MB of

memory usage. We round it up and double to account for metadata and leeway for the garbage collector to reach an expected 1GB of memory usage.

Total

A node therefore needs about 5GB of memory and 15GB of disk. The 16GB and 100GB retained in the recommendation below leave a margin of roughly 3x over these estimates. This margin covers the Go runtime and the operating system, and absorbs traffic exceeding the assumptions above (larger geometries, more conflicts per flight, a higher peak-average ratio). Both figures grow linearly with the number of active flights, so a deployment below the future-scale figure needs proportionally less.

Resources requirements recommendation

Resource	Requirement	Rationale
Memory	16GB	The whole dataset is held in memory. The pool's capacity is bounded by its smallest node. See estimations calculated above. Additionally matches CockroachDB sizing recommendations, which we take as an upper bound: actual requirement might be lower.
CPU	4 vCPU	Every node applies every write; The leader additionally fans out replication. Matches CockroachDB minimal sizing recommendation, which we take as an upper bound: actual requirement might be lower.
Disk	100GB, local disk or low fsync latency network disk.	The write-ahead log is written before an entry is acknowledged, so disk latency is on the commit path. Snapshots need room for one extra copy of the dataset. Network-attached volumes work if their fsync latency is acceptable. See estimations calculated above.
Network	1Gbps	Steady traffic is one proposal per request to every node. A node catching up from a snapshot receives the full dataset once. This will depend heavily on traffic shape (e.g. a lot of OVN's mean bigger requests and therefore more bandwidth required). Proposal payload size is the first bandwidth limit observed under emulated latency; encoding, compression and OVN-format work in #1587 reduces it.
Latency to	Tested at 16.5 +- 2.5ms	Commit latency is one round trip to the

peers	between participants	nearest majority. Participants far from the majority see proportionally higher write latency. Latency will also limit available bandwidth.
Ports	1 ASTM APIs 3 Raft listeners (AUX, RID, SCD)	Raft listeners carry mutual-TLS peer traffic only. They replace the database port. See Security.

Placement

Availability comes from participants hosting independently: separate providers, separate regions, separate failure domains.

Clock

No change on existing requirements: ordinary NTP is required to have a similar time when needed or to agree on certificate expiration. Proposal carries the time to use in every request to be able to apply them at any time.

Certificates

Each participant operates a CA and issues a certificate per node. Every node holds the list of CAs its participant has configured it to trust. Adding a participant is a change to that list on every node before its first node can join. See Pool Membership.

Process management

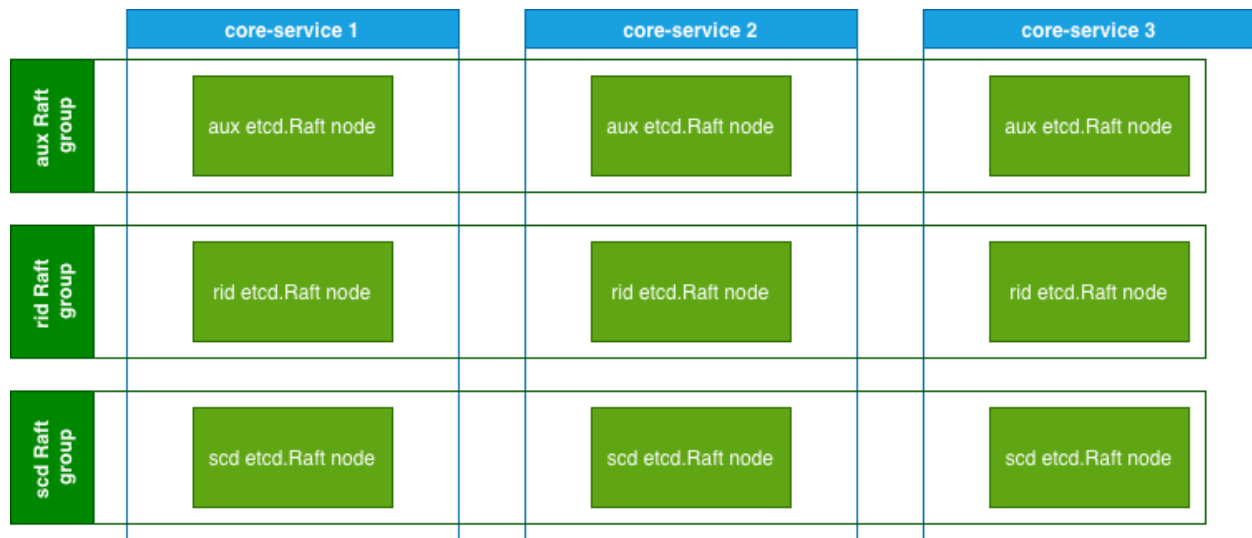
The node is a single binary and runs under any supervisor: systemd, a container, a Kubernetes StatefulSet with a persistent volume. The provided Helm chart, tanka file and Terraform modules will be updated for the new layout.

Core-service detailed technical design

A USS sends a request to whichever DSS node it chose over the unchanged ASTM API.

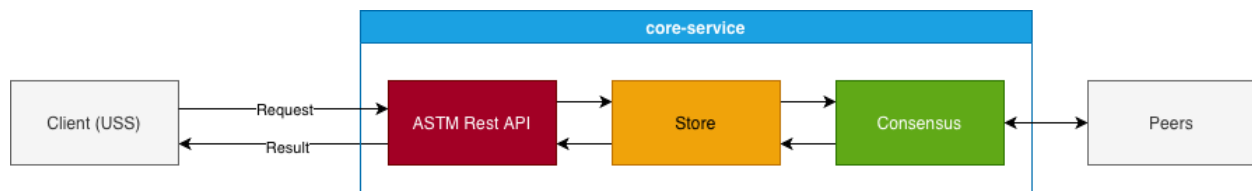
The node builds a Raft proposal out of the request encapsulating the operation and necessary values. Raft assigns that proposal a position in the shared log. Once the proposal is committed, every node executes it against its own in-memory copy of the data, in log order. The node that initially received the request may now return the result to the requesting USS.

The subsections below follow the path of a request through three layers. The store layer builds the proposal, hands it to consensus and waits for the result. The consensus instance wraps the etcd Raft library and is responsible for ordering the proposal and making it durable. The memstore holds the data and is where the proposal is finally applied. Each of the three services (aux, rid and scd) runs this stack independently, over its own Raft group and its own data.



Raft groups in core service

Each service exposes its (unchanged) ASTM Rest API and is backed by its own independent `store.Store[R]` which does Raft-based replication:



Path of a request

Alternatives considered

We considered using one shared consensus instance for all services. However, this made the initialization and the interface of the store more complex. It is also more conceptually suitable to separate the services as, for example, SCD can be disabled. In addition, multiple groups allow multiple leaders, splitting the load on different nodes.

These subsections describe the implementation, down to the Go interfaces involved. Readers interested in the guarantees rather than the mechanism can continue at Pool Membership; the guarantees are restated under Quality Attributes.

Store layer (raftstore)

The DSS supports more than one storage backend, all behind one interface, `store.Store[R]`. This is a Go generic type: `R` is the repository interface of the service it serves. Each of AUX, RID and SCD defines its own `repo.Repository`, listing the operations available on its data:

GetSubscription, InsertSubscription and so on. A service programs against Store[R] and its own R, and does not know which backend it is running on.

The interface has two entry points:

- Interact(ctx) (R, error) returns the repository for the caller to use directly. Each method call on it, reads included, is one unit of work for the backend. Every call is individually consistent, but a sequence of calls is not atomic: another node's work may be ordered between two of them.
- Transact(ctx, request OperationRequest) takes an entire operation, spanning several repository calls, and runs it as a single unit of work. Operations are registered under an ID at startup, identically on every node, so that a request carrying only the operation ID and its parameters resolves to the same code everywhere.

Two backends implement the interface:

- sqlstore talks to CockroachDB or YugabyteDB: a repository call becomes a query, and Transact becomes a database transaction.
- raftstore implements the application-level consensus: a unit of work becomes one Raft proposal, replicated through consensus, then applied to a local memstore that raftstore embeds. Selecting raftstore changes no business logic.

raftstore.Store[R] treats consensus as a black box. It builds a proposal, hands it over, and waits for the result. It does not know how the ordering is reached.

Unit of work

Under the raftstore, the unit of work is one call to `Apply`. This is what makes the difference between the two entry points: an Interact call is atomic on its own; a Transact operation is atomic as a whole. Before any write proposal is applied, we mark a checkpoint at the memstore and restore it in case of failure.

Transaction vs Interaction workflow

Which entry point an operation uses follows from this. F3548-21 requires a USS to present the version of the state it read (the OVN), and the DSS to reject the write if that state has changed. Validating an OVN and then mutating is a read-modify-write, and splitting it across two proposals would allow another node's write to land in between, defeating the check the standard requires. Such operations go through Transact. Interact is for single reads and for operations with no read-modify-write dependency.

Cost

Writes go through a proposal which costs a round trip to a majority of the pool and routes every request through the current leader.

Reads are served using the library's `ReadIndex` method which, instead of proposing to consensus, has the leader confirm its leadership (one heartbeat round trip to a majority) then waits for its state to catch up to a confirmed index on which the read can be served. This way, reads don't occupy a log position and don't need any WAL write.

Payload size matters as much as round trips. Gate 1 showed that under emulated latency the bandwidth taken by proposals, not consensus, is the first limit. Proposals are currently JSON-encoded. #1587 introduces a compact encoding, compression, a shorter OVN format and removal of unused keys; these are part of the production-ready version.

Data replication

A proposal's payload is a one-to-one mapping between an HTTP request or a repo method to a set of objects needed for the execution of the request. Every node then applies that request as a function of the payload itself and arrives at the same result.

This keeps proposals independent of how the data happens to be stored, so a change to the memstore's internal structures is not a change to the replicated format. Correctness therefore depends on every node executing the request identically. That obligation is set out under Determinism below.

A proposal therefore carries the payload, and alongside it the values a node cannot derive for itself:

- Timestamp: the request time, taken by the proposing node. Every node uses this value rather than its own clock.
- Locality: the locality of the proposing node.
- Node ID: the proposing node's identity, for operations whose result depends on it.
- ID: identifies the proposal so the proposing node can match the committed result back to the waiting request.
- ReadOnly: marks a proposal that does not modify state.

Raft replicates the whole proposal and persists it as a log entry. Once committed, each node hands it to the service's store, which decodes it and runs the corresponding business logic against its own memstore.

Alternatives considered

Replication of the resulting state rather than the request: the proposing node could be precomputing the result of the request and sharing the result to the other nodes so that they can store it as is. While this approach has the benefit of eliminating the determinism concerns, it strongly couples the memstore internal representation with the DSS operation logic, making very difficult any change in both.

Determinism

The nodes' data copies remain identical because they execute the same instructions in the same order. While Raft guarantees the order, it does not handle the execution. Given the same proposal, two nodes will propose the same state only if the applying code only depends on the proposal and the state already in memory.

The apply path constraints

The apply path's code is a pure function of the current state and the proposal. It must not read the clock, generate randomness, read the environment, make network calls, or depend on which node is executing it. What it needs and cannot derive travels in the proposal.

Linters will be configured to enforce that the code respects this.

Sources of divergence

Source	Treatment
Wall clock	Proposal timestamp. Time. Now forbidden on the apply path.
Randomness, generated IDs	Pre-computed and included in the payload.
Node identity	Carried in the proposal (node ID and locality).
Go map iteration	Randomised by design. Sort into a total order before iterating.
Unstable sorts	sort.Slice is not stable. Use a total ordering.
Geometry	Floating point differs across architectures and compilers. Geometries shall be transformed to S2 cells before being proposed to Raft.
Floating-point arithmetic	Not allowed on the apply path.
Concurrency	One proposal at a time, in log order, per raft group.
Dependency upgrades	Any upgrade that can change an applied result is a behaviour change, gated by the version mechanism in Multi-Version Interoperability.
Direct modification to the memstore	No change to the memstore shall be performed without proposals. (e.g. expiry and housekeeping)

Alternatives considered

Computing the S2 cells covering on the apply path was considered as it kept the mapping from HTTP requests to proposals simple, but it put floating-point geometry on the apply path,

where results can differ between CPU architectures and compiler versions. Computing the covering before proposing removes geometry from the apply path, so nodes on any architecture apply identical inputs.

Enforcement

Apply-path code sits behind a package boundary and does not import time, math/rand, os or net which will be enforced by a CI check.

Runtime comparison

Nodes regularly exchange a hash of their memstore and compare them, to catch drift early.

A proposal initiates the call to calculate a hash at a specific log position, ensuring that every node hashes the same state. A node whose hash differs from the hash reported by a majority of voters stops serving requests. It is then recovered as for a lost data directory (see Node lifecycle).

How often the hash is computed remains an open question: after every proposal, at a fixed interval or at each snapshot, or on demand.

Future work

How often the hash is computed, and whether recovery of a diverged node can be automated, need to be evaluated in terms of performance and availability.

Advanced tooling for dumping and comparing all data within all instances of a cluster is also an option, but this requires extra consideration since a lot of data may be required and having a snapshot of data on a live cluster with running queries can be challenging.

Scope

This applies only to code running when a committed proposal is applied. Logging, metrics, admin interfaces and response formatting outside the applied result may use the clock and the local environment as practical.

Consensus

Consensus is provided by the etcd Raft library: leader election, log replication, and the guarantee that committed entries are never lost or reordered. It leaves persistence, networking and timing up to the user. The consensus.Consensus type wraps it and supplies the three.

The library is a state machine the application drives. It takes proposals, peer messages and clock ticks and hands back, in batches, the work that results: entries to persist, messages to send, committed entries to apply.

```

Go
// Only methods relevant to this document are shown
// Node represents a node in a raft cluster.
type Node interface {
    Propose(ctx context.Context, data []byte) error
    Ready() <-chan Ready
    Tick()
    Advance()
}

```

Propose submits a proposal. Tick advances the logical clock, in which election and heartbeat timeouts are counted; the consensus instance calls it at a fixed interval. Ready delivers the next batch. Advance acknowledges it.

One loop per Raft group processes each batch in order: persist new entries and Raft state to the write-ahead log, send outgoing messages, apply committed entries through the store, restore a snapshot if one is present, call Advance. Persist precedes send: a node never acknowledges an entry it has not written. Persistence is covered in the Raft Storage section, and sending in Raft Transport.

Tuning parameters will be exposed as CLI flags and are listed under Configuration.

Proposal lifecycle

The raft store wraps the payload in a Proposal, registers its ID with a tracker, and calls Propose. A non-leader node forwards it to the leader. The leader appends and replicates it; once a majority of nodes have persisted the proposal, it arrives committed in every node's next batch. Every node applies it. The proposing node matches the result to the waiting request by ID and returns it.

Raft Storage

The library reads its log through a raft.Storage interface and persists nothing. The consensus instance uses etcd's raft.MemoryStorage, which holds the log in memory, and writes every entry and Raft state change to a write-ahead log before it reaches memory storage. The WAL implementation is etcd's.

A proposal is committed once a majority has written it to disk. A node that crashes and restarts still holds every committed entry it acknowledged.

On startup, a node looks for an existing WAL. If it finds one, it loads the latest snapshot into the store and replays the entries after it. If none, it starts empty: bootstrapping a new pool or joining one, as described under Configuration.

An unreadable WAL is not recoverable locally, and the node must not start with an empty one: it would present itself to the pool as current while holding nothing. Recovery is the same as for a

diverged node: remove it, discard its data directory, add it back as a new node with a new ID. It then receives a snapshot from the pool.

Snapshots

Consensus decides when a snapshot is taken or restored and where it is stored. The store decides what it contains, through `GetSnapshot` and `RestoreFromSnapshot` on `RaftRepo` (see Storage API). Consensus treats the bytes as opaque.

Snapshots serve two purposes:

1. Locally, a snapshot prevents infinite growth. One is taken every `SnapshotIntervalEntries` applied entries. Entries it supersedes are compacted, so the WAL stays finite and a restart does not replay the pool's whole history. `SnapshotCatchupEntries` entries are kept, so a slightly lagging follower catches up from the log rather than by full transfer.
2. Across the pool, a snapshot brings back a node that has fallen too far behind. If a follower needs entries the leader has already compacted, the leader sends a snapshot instead, streamed by `rafthttp`. The receiving node surfaces it through `Ready`, and consensus signals the store to restore. This is also how every new node gets its initial state.

A restore replaces what the receiving node's data storage had. This is what makes remove-and-re-add a complete recovery for a node whose WAL is lost or whose state has diverged.

Snapshots are also where a version boundary shows up, since a snapshot taken by one release must be loadable by another. That is covered under Multi-Version Interoperability.

Raft Transport

Peer-to-peer communication uses `etcd's rafthttp` package, which carries Raft messages (votes, entries, snapshots) over HTTPs between the nodes of one Raft group, on the ports and under the mutual TLS described in Security.

Alternatives considered

We considered using our own gRPC transport. A basic implementation measured comparably so we went with the more practical choice of `rafthttp` as it is specifically optimized for the `etcd` Raft library.

Memstore

The memstore holds the entities in memory and all the methods reading or manipulating it do so in-memory only. Persistence to disk is exclusively handled by `raftstore`. The structures are plain Go maps, with an index only where a query needs one. Their layout is an implementation detail:

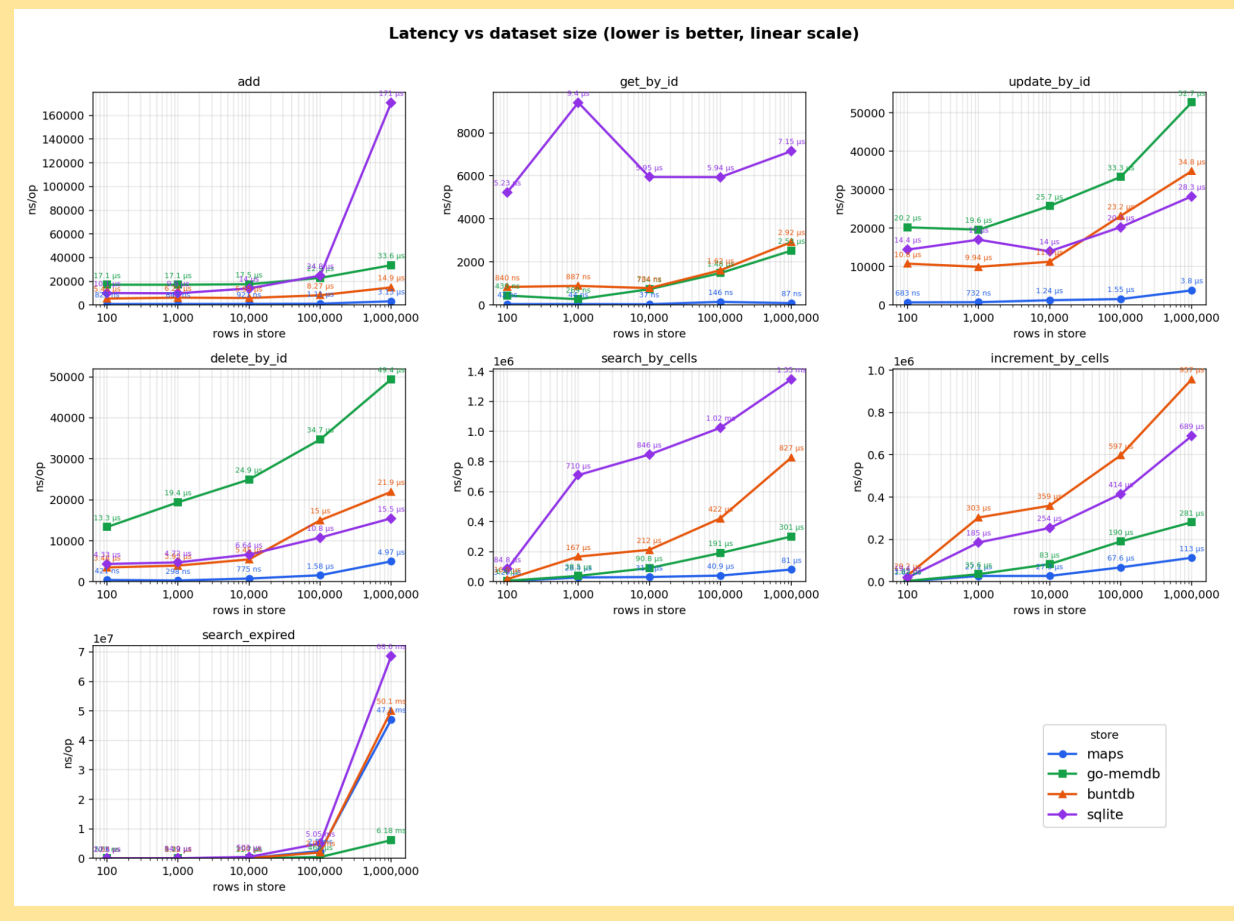
it is not replicated, not part of any interface, and can change between releases without touching the wire or snapshot format (see Data replication).

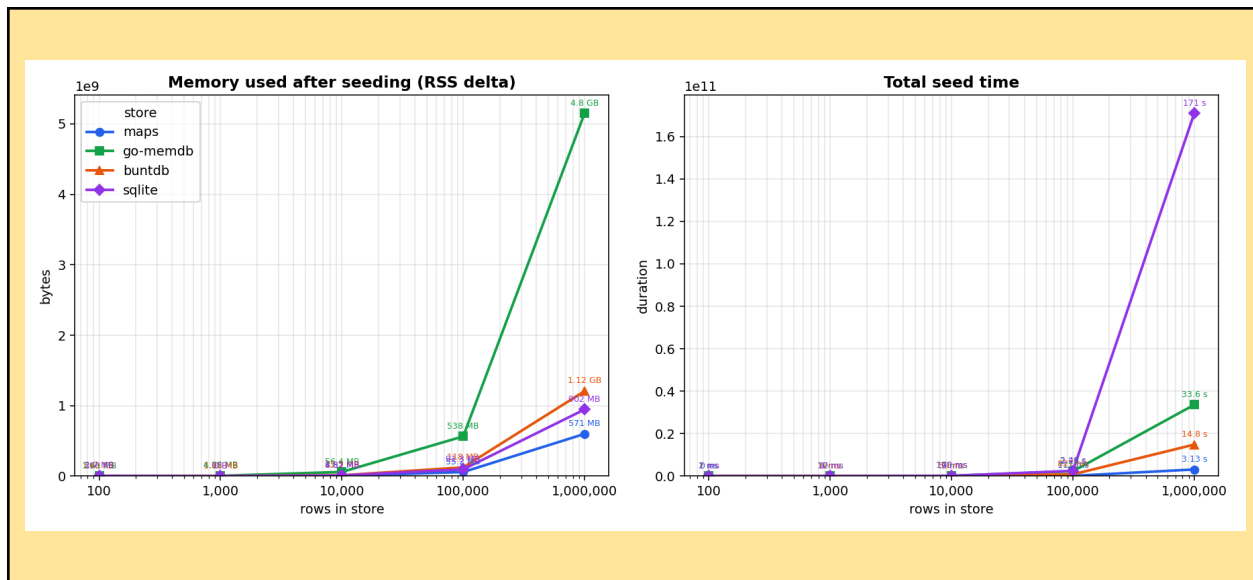
The memstore is called only from the apply loop. Write proposals are applied one at a time in log order. It holds no locks. The raftstore embeds a memstore.Store[R]. On a committed entry, its Apply method decodes the operation name and arguments (see Data replication), resolves the registered operation, and runs it against the memstore.

Expired entities are removed by a housekeeping eviction job as for the other stores, which will be submitted as a proposal, never by a local timer. The proposal names what to remove, and every node removes the same entities at the same log position.

Alternatives considered

In-memory maps, go-memdb, buntdb and SQLite were benchmarked (#1526). Plain maps were faster in most cases and used less memory. Given the simple structure of the data, plain maps with indexes added only where a specific query needs one carry no overhead paid elsewhere.





Pool Membership

A pool is the set of nodes, run by several participants, that replicate the same data. Participants come and go: a new USS joins, one leaves, one rotates a node. Each such change must be agreed by the pool, since it changes who holds the data and who counts towards a majority, and it must work without a central administrator, since no participant is privileged. This section describes what membership is, how a node comes to be trusted before it can be a member, how nodes join and leave, and who may propose a change.

The library keeps the list of nodes in a Raft group and their voter or learner status. It is handled as a log entry, proposed and committed like any other query.

A DSS node is a member of three groups, AUX, RID and SCD, and each group keeps its own list. Adding a node means adding it to all three; dssctl issues the three proposals and reports on each. The groups are independent, so a node may be a voter in one and a learner in another.

Trust

Nodes authenticate each other with mutual TLS. Each participant operates its own certificate authority (CA) and issues a certificate to each of its nodes. Every node keeps a trusted list: the CAs of the participants it accepts, held as local configuration and changed only by the participant who runs the node. A node accepts a Raft connection only from a peer whose certificate is signed by a CA in that list, and no Raft message is exchanged before the certificate is verified.

Communication over a mutually authenticated TLS channel is therefore a prerequisite for Raft membership. The membership list may change through proposals, but a proposal can only add a node that can already communicate with the other nodes.

Only CA certificates are exchanged. The exchange is symmetric: the joining participant installs every existing CA, and every existing participant installs the joining one. How participants exchange and verify CAs outside of the scope of this design; it is the same exchange participants already perform to form the database cluster today.

A participant joins in three steps: the CAs are exchanged, every node installs the new one, then the addition is proposed. A participant leaves in reverse: the removal is committed first, and its CA is dropped once it holds no node in the pool. Dropping the CA before the removal cuts the node off while it is still a member, which the pool reads as a failure rather than a departure.

A CA installed on some nodes and not others fails silently. The failure may be detected with the `raft_peer_tls_handshake_failures_total` metric (see Observability).

Bootstrap and joining

Bootstrap happens once, when the pool is created. The founding nodes start with the same list of peers and the same cluster ID, and every founding node is a voter. The founding participants exchange their CAs before any node starts, since the nodes attempt to elect a leader as soon as they are up.

For a new node to join, an existing participant proposes the addition with `dssctl`. Its participant's CA is in every node's trusted list by then; see Trust between participants. Once the proposal is committed, the new node is started with an empty data directory. It presents itself to the pool, receives a snapshot from the leader, applies the entries since, and is caught up. See Snapshots.

Node IDs are never reused. A node that is removed and re-added comes back with a new ID. The log still contains entries about the old ID, and reusing it would let stale messages be mistaken for current ones. The IDs of removed nodes are tracked (when deletions are applied) and are rejected when an addition tries to reuse them.

Learners and voters

A node can be a learner. A learner receives the log and applies it, serves reads and forwards writes like any other node, but does not count towards quorum.

The users may add nodes as learners or as voters. They can also promote or demote nodes across these statuses. The section Risks and Mitigations outlines the recommendations on the matter.

When joining an existing cluster, newcomers join as learners until they catch up before getting promoted to the voter status. This avoids increasing the quorum with a node that is not up to date yet.

Users will have the ability to manually manage the learner / voter status of the nodes in the cluster.

Removing a node

Removal is a proposal like addition. Once it is committed, the other nodes drop the removed node from the transport's peer list and stop accepting its messages. The removed node, on applying the entry that names it, persists it and shuts down. It does not need to be reachable for the removal to succeed. The departing participant's CA stays in every trusted list until it holds no node in the pool; see Trust between participants.

Removing a voter changes the quorum immediately. A pool of 9 voters that removes one tolerates 3 failures; removing two more tolerates 2. dssctl refuses to propose the removal of a voter if the resulting pool would have fewer than three. Removing the current leader triggers an election.

Removal also drops the node's behaviour-version report, so an old node that is decommissioned rather than upgraded does not hold the pool version back; see Multi-Version Interoperability.

One change at a time

Changes are proposed one node at a time, and dssctl waits for each to commit before proposing the next. There is no scenario in a DSS pool where two nodes must be added or removed atomically.

Who may propose a change

Any node may propose a membership change, and the protocol has no approval step. A participant that can reach a node of its own can add or remove any member of the pool.

What the protocol does not protect against is a participant acting in bad faith. That is the trust assumption stated in the Summary, and it is the same assumption the current database cluster makes.

Alternatives considered

Static membership

A peer list in each node's configuration, changed by editing every node and restarting.
Rejected: every change becomes a coordinated restart across participants.

A designated administrator instance

One participant's node is the only one allowed to propose changes.

Rejected: it makes that participant a privileged peer, which the requirements exclude, and a single point of failure for membership.

Join directly as voter

Skip the learner step.

Rejected: the quorum widens before the new node can acknowledge anything, so commits slow or stall until it has caught up.

APIs

The DSS node uses four interfaces. Three on the request path, plus the admin interface used by `dssctl`. Each has its own callers, its own format and its own versioning rule. The following table summarizes the next subsections.

Layer	Caller	Carries	Versioned by
Public API	USSs	ASTM REST requests and responses	ASTM Standards
Consensus API	REST endpoint handlers	A proposal: request type, encoded arguments, and the values needed for deterministic execution	Behaviour version, see Multi-Version Interoperability
Storage API	The consensus instance, on snapshot and restore	A serialised copy of one service's applied state	Behaviour version, see Multi-Version Interoperability
Admin API	<code>dssctl</code>	Membership changes, option changes, status	Release version

Public API

The ASTM REST API is unchanged. USSs cannot tell which backend serves them. Its versioning is the standards', plus augmentation like proposed OVN.

Consensus API

There are two entry points to the consensus instance: one for proposing membership changes (adding or removing nodes from the cluster) and another one for proposing values to be replicated by Raft.

Pool Membership API

Membership changes are possible via the `ProposeConfChange` method of Consensus.

(The full method signature and design of pool membership handling will be defined later).

Replication API

For data replication, the `raftstore.Store[R]` calls consensus through:

```
Go
func (c *Consensus) HandleClientRequest(ctx context.Context, requestType
RequestType, value []byte, readOnly bool) (any, error)
```

This call blocks until the proposal has been committed and applied. It returns the result (e.g. a subscription / operational intent) and an error in the following cases:

- The proposal failed to reach Raft
- Context timeout
- The proposal was applied but rejected by business logic (e.g. invalid OVN's)

The communication format between DSS nodes for replication is fully internally handled by the etcd Raft library. We only provide it a payload in the form of an encoded proposal:

```
Go
type Proposal struct {
    ID          string    `json:"id"`
    Locality    string    `json:"locality"`
    NodeID      uint64    `json:"node_id"`
    Timestamp   time.Time `json:"timestamp"`
    RequestType RequestType `json:"request_type"`
    Value       []byte    `json:"value"`
    ReadOnly    bool      `json:"read_only"`
}
```

Once committed, each node hands the proposal back to the service's store which then applies it using `Apply` which runs the relevant business logic:

```
Go
func Apply(ctx context.Context, proposal consensus.Proposal) (any, error)
```

Storage API

In this design, there are two layers of storage.

- Raft's log storage persists the agreement itself: the ordered sequence of committed proposals. It is backed with a WAL and periodic snapshots on disk. This layer is internal to the `raftstore` which stores the log entries with proposals as opaque blobs.
- Application data storage persists the result of applying the proposals (the actual DSS data).

The two layers meet in the `RaftRepo` interface:

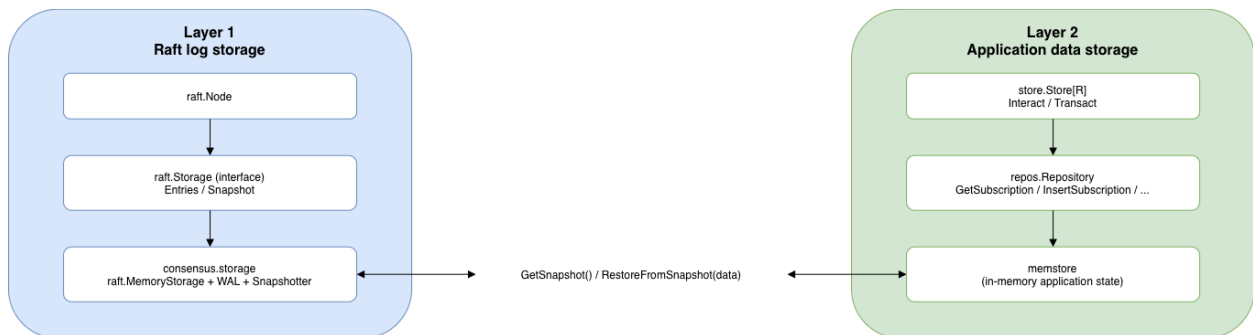
```

Go
// Only relevant methods are shown
type RaftRepo[R any] interface {
    // GetSnapshot returns a serialized view of current state, suitable
    // for restoring via RestoreFromSnapshot.
    GetSnapshot() ([]byte, error)

    // RestoreFromSnapshot replaces all state with the snapshot in data.
    // data is always the output of a prior GetSnapshot.
    RestoreFromSnapshot(data []byte) error
}

```

A snapshot is a serialized dump of layer 2 stored inside layer 1's persistence. This is how a node recovers its state after restarting. Consensus treats these bytes as opaque. It only decides when to trigger a snapshot, where to store it and when to signal to the service to restore its data.



Admin API

The admin endpoint used by `dssctl` is served on its own listener, under the same mutual TLS. It is not exposed to other participants.

Multi-Version Interoperability

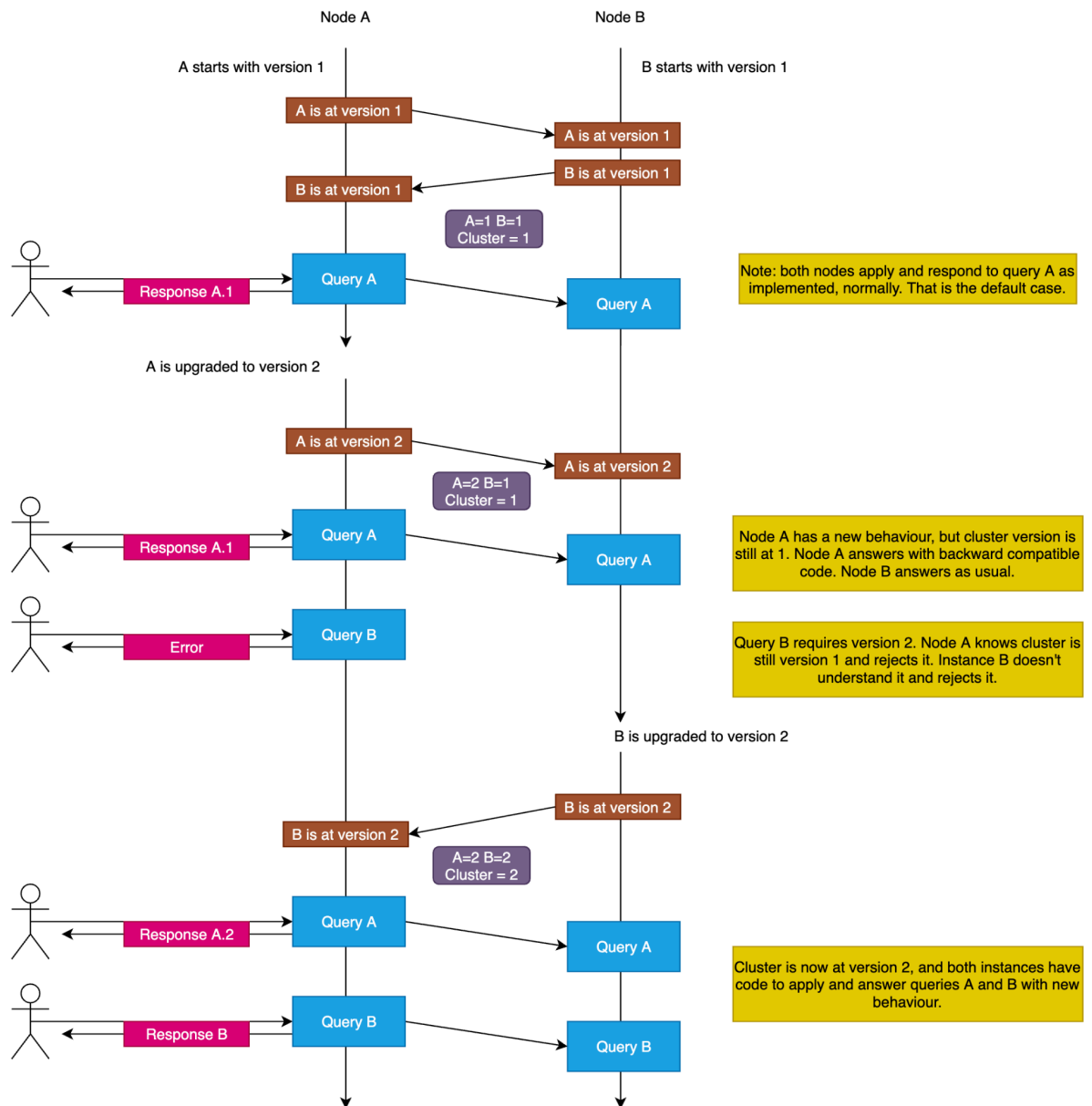
A pool is usually upgraded one DSS instance at a time. This section describes the rules that keep a mixed fleet correct, allowing a DSS pool to run two different versions (N and N+1) during a few weeks transition period.

Maintaining a mixed-version pool imply two major elements: the behaviour of the apply path, which under Determinism must be identical on every node for a given proposal as well as the format of data exchanged between nodes which shall remain processable by all nodes in the pool during transition.

Behaviour changes

To ensure the behaviour of the apply path is identical on every node for a given proposal, release N+1 shall always be capable of applying a proposal using the highest behaviour version supported by release N to prevent mixed pool drifts. The hash check under Integrity catches this but by then the node must be rebuilt, and every upgrade would trigger it. Under Determinism, any change that can alter an applied result counts: a bug fix in business logic, a new field that changes an index, a dependency upgrade. The rules below allow such changes to ship while the pool runs mixed releases.

- Each release carries a behaviour version, incremented whenever the apply path changes. The pool agrees on which version is in force, and every node applies every proposal with that version, whatever code it is running.
- Reporting: on startup, a node publishes a proposal announcing its node ID and its behaviour version. Like any proposal it is committed to the log of each Raft group (AUX, RID, SCD) and applied by every member, so every node holds the same table of who runs what.
- Pool version: the version in force is the lowest reported by any current member. It is recomputed whenever a report or a membership change is applied, and stored as part of the replicated state.
- Applying: on each 'apply', a node reads the pool version and executes the behaviour for that version. Code for a change ships with the previous behaviour kept alongside it, selected by version. A node running N+1 in a pool at version N behaves exactly as N. This is what keeps the hash check under Integrity green throughout an upgrade.
- Requests that need the new version: a request whose semantics only exist at N+1 is rejected by the receiving node with an explicit error before it is proposed, while the pool is at N. It is not proposed and silently dropped. A node at N that receives a proposal type it does not know rejects it.
- Activation: the new behaviour takes effect for a proposal applied after the pool version reaches N+1, which happens when the last member at N is upgraded or removed. Activation is a point in the log, so every node switches at the same proposal. There is no manual step.
- The memstore follows the same versioning. The snapshots created with version N are compatible with behaviour N and N+1. When the pool version reaches N+1, and if needed, all nodes migrate their snapshot to the N+1 version.
- Membership: removing a member removes its report, so a decommissioned node at an old version does not hold the pool back. A member that is down still counts. Its report is in the state and it will apply the log when it returns, so the pool cannot move past it without risking drift. The pool version therefore does not advance while an old member is offline; the operator's choice is to wait or to remove it.
- Minimum version: each release has an oldest behaviour version it can execute. A node whose code cannot execute the pool version refuses to join. A node whose code can execute it but not any newer one is simply the oldest member, and holds the pool at its version.



This requires the code to support all versions, which may become complex to manage over time. To limit complexity, backward-compatibility will be guaranteed only between two versions requiring users to upgrade one version at a time. This is however something very important to consider and manage correctly over time. Even fixing a bug in non-direct code (e.g. updating a dependency that changes its behaviour / result) needs to be versioned and be backward compatible: all service instances in a cluster, regardless of version, must have the same exact behavior or their local storage may drift.

Alternatives considered

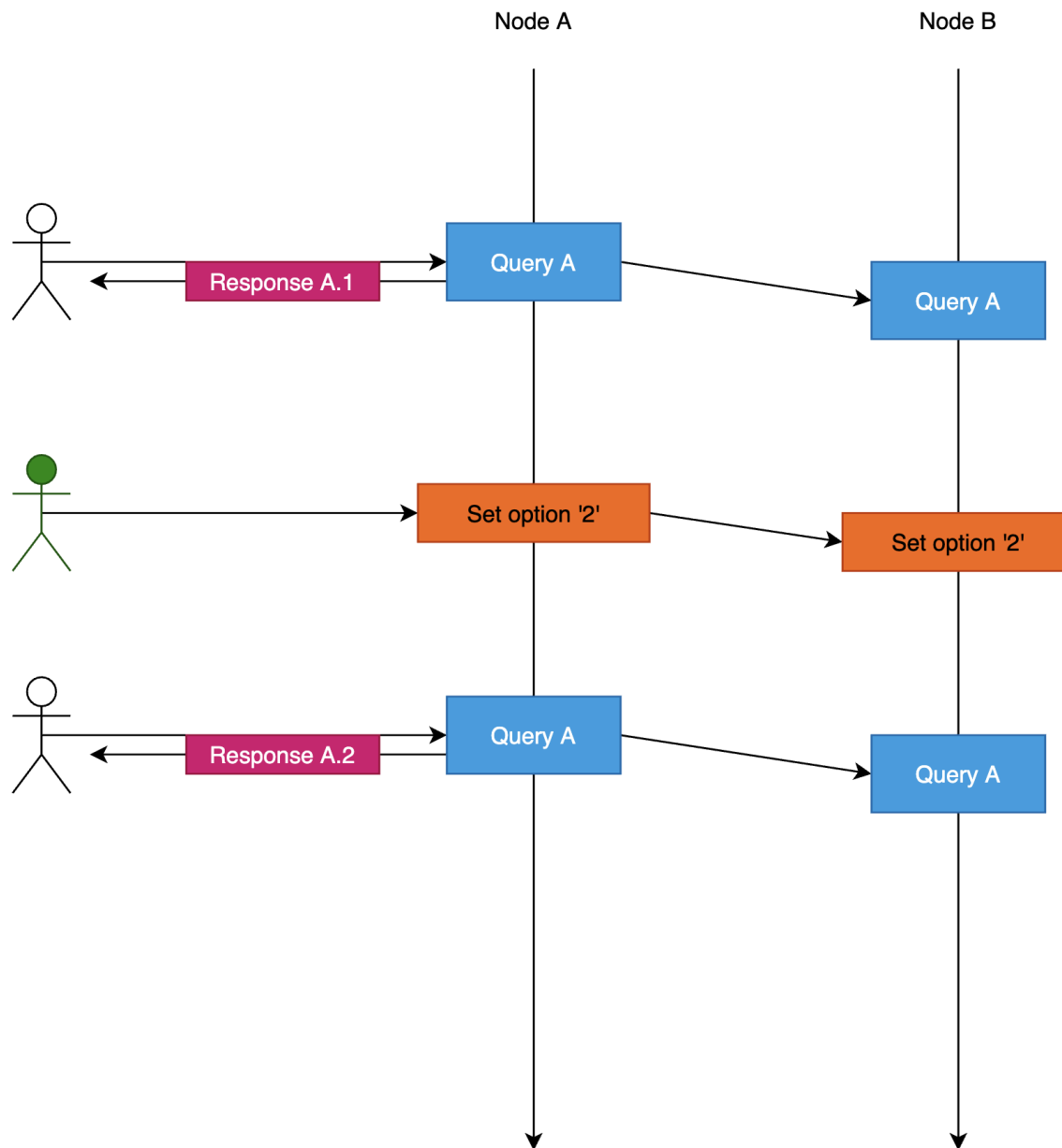
Should this be not maintainable due to complexity or too complex change, switching to incompatible releases with required downtime is always a possible solution.

Options introducing new behaviour

Some behaviour is switched by configuration rather than by release. The time-based notification index is an example. Such an option, set on one node and not another, is a determinism violation like any other.

Options are therefore propagated. Each has a default, stored with a proposal. An operator changes it through administrative command, which submits the change as a proposal. Every node applies the change at the same point in the log and reads the current value on every subsequent apply. Because a new option is a behaviour change, its proposal is rejected unless the pool version already includes it.

Every participant can change any option. Agreement on what a pool runs is a governance question, as for membership.



Compatibility window and downgrade

A release N can read snapshots and execute behaviour from the release N-1. A pool more than one release behind will need to upgrade through some intermediate releases first. Two keeps the amount of legacy code in the apply path bound and matches the rhythm at which pools are expected to upgrade. This number is a project rule, not a design constraint, and can be revised.

Downgrade is unsupported. Snapshots do not convert downwards, and a node at version N cannot execute a pool at N+1. Reverting a release is done by removing the node, discarding its

data directory, and re-adding it at the older release before the pool version advances, which is only possible while at least one other member still reports the older version.

Alternatives considered

Incompatible releases: every behaviour change requires the whole pool to stop, upgrade, and restart together. The code would be simpler, but it makes the mixed fleet, which is unavoidable across organisations, an error state rather than a supported one, and it requires downtime for every new release. It remains the fallback for a change that cannot be made backward compatible, and such a release is announced as such.

Admin User Interface

A `dssctl` CLI will be used to handle admin tasks such as cleanup, options change described in previous section or pool membership changes.

We expose a protected `admin` endpoint that the CLI will interact with.

The CLI will cover the following operations:

- Add / remove node with ability to skip pre-join checks.
- List nodes
- Promote / demote node. (Voting or learner status)
- Set pool options.
- Eviction jobs.
- On-demand hash state trigger.

The full design of the CLI and the endpoint will be specified later.

Quality Attributes

Raft's protocol guarantees (leader election safety and log matching) are taken as given and the etcd Raft implementation is assumed correct. Everything in this section follows from this assumption plus how the raftstore builds on it.

ACID properties

The proposed design gets them from the shared ordered Raft log. Within one service, every node applies proposals one at a time, in log order, and the proposal is the unit of atomicity, isolation and durability. The properties below hold per service.

Atomicity

A proposal takes effect entirely or not at all. An operation submitted through Transact is one proposal, so a read-modify-write such as an OVN check followed by a mutation cannot be split by another node's write. Calls made through Interact are each their own atomic proposal (see Entry points).

There is no rollback. A proposal that fails validation (a stale OVN, a reference outside its time bounds) is committed to the log and applied as a rejection and the caller receives the error. A node that fails part way through an apply stops rather than continues from a half-applied state, and is rebuilt from a snapshot (see Determinism).

Consistency

Every proposal moves the state from one valid state to another. The invariants required by F3411 and F3548 (OVN matching, time bounds, ownership) are checked inside the apply, against the state as it stands at that log position, before anything is mutated. Because every node checks the same proposal against the same state, a proposal that is valid on one node is valid on all of them.

Isolation

Serial. Within one service, proposals and reads (via `ReadIndex`) are applied strictly one at a time, in log order. No two proposals ever execute concurrently against the state, hence the serial guarantee.

Durability

A proposal is committed once a majority of the Raft group's voters have written it to their write-ahead log and synced it to disk. Only then is it applied and its result returned.

The guarantee is bound by the quorum: a committed proposal survives the loss of any minority of voters. Simultaneous loss of a majority's disks can lose committed entries.

Linearizability

The output of Raft replication is an ordered log of entries meaning that each committed entry has a definite position relative to every other and all nodes share this view of the Raft log and apply it sequentially. This leads to writes being linearizable. Reads are served using `ReadIndex` which confirms the current leadership and waits for local state to catch up before serving the read, making it linearizable as well.

Fault tolerance

A Raft group tolerates the loss of a minority of its members. Of $2n+1$ nodes, n can be down or partitioned. Recovery after losing a leader is bounded by $\text{election_tick} \times \text{tick_interval}$ plus one election round; during it, no proposal commits. The thesis prototype, with default timeouts of 2 seconds for election timeouts and a heartbeat timeout of 100 ms, absorbed a leader crash with a 20 % throughput dip and about 25 s of recovery.

Horizontal scalability

Adding nodes does not add throughput. Each added node widens the majority a proposal must reach and adds its own round trip to the commit path. Gate 1 measured a 9-node pool at under 300 ms against a 5 s bound; whether the 24-node requirement stays within the bound is not yet known and will be measured as part of Gate 2 work. If it does not, two options are available, both discussed under Future Considerations: decoupling the voting set from membership, and partitioning the single Raft group into multiple groups.

Cross-Cutting concerns

Security

Today, a DSS node exposes two things to the internet: the ASTM API and the database. This design removes the database but substitutes it with ports for the raft protocol, so how peers authenticate to it don't change.

The ASTM API is authenticated using JWT tokens. The CockroachDB port is authenticated by a certificate; every node reads and writes over it, and database nodes replicate among themselves over it. A node exposes the ASTM API, as before, and three Raft listeners over HTTP, one per service, carrying peer traffic only under mutual TLS. The ASTM API, its authentication and its authorization are untouched. The rest of this section describes the Raft layer only.

Authentication and Authorization

Each participant operates a self-signed CA. Every node holds the list of public certificates its participant has configured it to trust. A Raft listener accepts a connection only if the peer presents a certificate signed by a trusted CA, and the peer verifies the listener the same way. The trusted list is configuration on every node, changed by every participant when a participant joins or leaves; see Pool Membership.

There is no authorization layer at the Raft level. Any authenticated node may propose any entry, including membership changes and option changes. The pool is a group of participants that trust each other's operation; this is the same assumption the current shared database makes, and Raft does not tolerate a member acting in bad faith. What the design does provide is accountability: every membership and option change is a log entry carrying the proposing node's ID, so who changed what is recorded in the log until the next snapshot compacts it.

Certificate rotation is a local operation: a node restarts with its new certificate and reconnects. CA rotation is a trusted-list change on every node, made before the old CA expires, with both CAs trusted during the overlap.

The admin endpoint used by `dssctl` is served on its own listener and authenticated by the same mutual TLS. It accepts connections from certificates issued by the node's own participant only.

Future work

After initialization, trusted CA list may be managed as a configuration propagated on a Raft group and automatically reloaded by nodes.

Confidentiality

In transit, node-to-node traffic runs over mutual TLS; nothing crosses the network in clear.

At rest, every node holds a full copy of the service's data, on disk and in memory. Its protection is the participant's responsibility, as the database host is today. Encryption at rest and erasure policies are applied there, per participant; the design imposes none.

Between participants there is no confidentiality by design: the pool holds nothing that one participant may not see. This is the position of the current cluster as well, made explicit.

Integrity

In transit, integrity is delegated to TLS, as confidentiality is. A message altered on the wire is rejected before it reaches Raft.

At rest, local storage is trusted without additional protection. A node's write-ahead log and snapshots are as safe as the disk the participant gives them. Silent corruption is detected on load using CRC in etcd's WAL and snapshot files.

Integrity of the replicated state is the one place a node can be wrong without any message or disk being tampered with: a bug or a non-deterministic code path can make one node's view diverge from the others while the log stays identical (see Sources of divergence).

TODO
Mention hash mechanism

Availability

The pool stays available as long as a majority of voters are available (see Fault tolerance). Every node answers every request, so a USS whose node is down could call another DSS instance.

Node availability is the participant's responsibility, just like database hosting is today. Those responsibilities imply:

- Independent hosting. Tolerance of n failures assumes they are independent. Voters sharing a provider, a region or a network path fail together (see Placement).
- A tested restore procedure. A new node is able to bootstrap itself after a catastrophic failure by fetching a snapshot from the pool.

When a majority is lost, the pool stops accepting writes until voters return. It does not evict unreachable members, since a dead node and a partitioned one look the same (see Future Considerations).

Observability & Telemetry

Metrics

Raft-related metrics are added to the existing core-service metrics¹ through the OpenTelemetry interface (first set in #1545), exported the same way and shown in a provided Grafana dashboard. They are read from values the library exposes:

- raft_peer_latency_seconds
- raft_peer_latency_average_seconds
- raft_peer_latency_stddev_seconds
- raft_peer_latency_min_seconds
- raft_peer_latency_max_seconds
- raft_peer_send_success_total
- raft_peer_send_fail_total
- raft_term
- raft_leader
- raft_commit_index
- raft_applied_index

These metrics would be extracted directly from values exposed by the underlying library and will be used as usual with other metrics, with a provided grafana dashboard.

In addition, the following metrics will be collected by raftstore. All are per Raft group (AUX, RID, SCD):

- raft_behaviour_version
- raft_pool_behaviour_version

The two metrics below are collected per node rather than per Raft group. A node holds one trusted CA list for all three listeners, and a rejected handshake fails before any group context exists.

- raft_peer_tls_handshake_failures_total, by reason and direction
- raft_trusted_ca_not_after_seconds, by issuer
- raft_proposal_duration_seconds
- raft_proposals_pending

¹ Memory usage, QPS are already available and are out of scope of this document.

Web interface

The goal of the web interface is to provide an alternative, quick view of current data, like the Cockroach web interface.

It will include:

- View of the cluster, with participants and latencies
- View of metric values (historical data is left to opentelemetry consumer such as prometheus/grafana)

The interface will only allow a read-only view: administrative tasks are left to CLI tools, limiting risks should an error in authentication / protection of the interface occur and reducing the hardening needs.

Future work

- Provide hash analysis capability to inspect possible divergence.
- Advanced debugging capabilities: surface the internal state of the nodes through the web interface (e.g. proposals, log, leadership information, etc.). We stick to analyzing logs manually for v1.

Testing

Applicative side

The DSS already includes extensive testing through the monitoring repository, which ensures compliance with the standards and expected behavior.

As of [#1538](#), automated testing has been added to CI (and passes), ensuring automated testing of the functionality of the new implementation to the same level as the existing implementation.

Pooling

A new test suite shall be developed to validate the correct behavior of the Raft cluster in common failure and operational scenarios, including:

- Loss of connectivity to a node
- Partial loss of connectivity between nodes (e.g., A and B cannot communicate, but A-C and B-C work)
- Dropped messages
- Addition of a new node
- Removal of a node
- Node restart during apply

- Leader crash under load (recovery time against the p95 bound)
- Upgrade behavior
- Corruption or loss of local snapshot data
- Proper validation of TLS certificates

While some of these cases primarily test etcd's Raft library, they should still be included to ensure that standard cluster scenarios execute successfully.

Operations, Rollout & Configuration

Configuration

Parameters

The backend (sqlstore vs raftstore) is selected using the `--store_type=raft` CLI flag.

To configure the node identity and raft tuning, the following flags are available and shared by the three services:

Raftstore Flag	Function
<code>raft_node_id</code>	ID of the node
<code>raft_cluster_id</code>	ID of the cluster
<code>raft_datadir</code>	Directory for raft data (WAL and snapshots)
<code>raft_snapshot_catchup_entries</code>	Log entries retained after compaction for slow followers to catch up
<code>raft_snapshot_interval_entries</code>	Applied entries before triggering snapshot
<code>raft_tick_interval</code>	Base time unit for Raft's logical clock
<code>raft_election_tick</code>	Ticks before starting an election
<code>raft_heartbeat_tick</code>	Ticks between leader heartbeats
<code>raft_max_size_per_msg</code>	Max bytes in a Raft message
<code>raft_max_inflight_msgs</code>	Max in-flight Raft messages

Since each service has its own Raft group, each of them defines its own Raft group initial cluster:

Service	Flag	Example
AUX	<code>aux_raft_peers</code>	1=https://node1:9031,2=https://node2:9031,3=https://node3:9031
RID	<code>rid_raft_peers</code>	1=https://node1:9011,2=https://node2:9011,3=https://node3:9011
SCD	<code>scd_raft_peers</code>	1=https://node1:9021,2=https://node2:9021,3=https://node3:9021

General configuration

Applicative configuration (example: the time based notification index flag) will be configured at the cluster level, without requiring restart nor synchronization (beyond administrative approval) in a cluster. See the “Multi-Version Interoperability” section for technical details.

Eviction of expired entities

The eviction of expired entities does not change compared to SQL-based storage. A configurable job runs and interacts with the store to delete the expired entities, which in the case of the Raft store, happens through proposals.

Node lifecycle

A node leaves the cluster and returns in one of four ways depending on what state survives and how long recovery takes.

Restart with an intact data directory: The node replays its snapshot and the entries after it, then rejoins and catches up on whatever it missed.

Restart after a long absence: If the node was away long enough that the leader has compacted the entries it needs, the leader sends a snapshot instead.

Restart with a lost or unreadable data directory: The directory should be discarded and the node added as a new node with a new ID. The WAL and snapshots contain information about votes cast, so a node without that information should no longer attempt to vote since it could introduce inconsistency.

Divergence: A node whose state hash differs from the pool stops serving. Recovery is the same as for a lost data directory. See Runtime comparison.

Restarting more than one node

A pool of $2n+1$ nodes tolerates n unavailable at once. A restart makes a node unavailable for its recovery time, so no more than n may be restarting simultaneously, and the pool should be at full membership before the next one begins. For a 24-nodes pool that is 11 at a time in theory. One at a time is the default when operators coordinate across organisations, since every restart also shrinks the margin against unplanned failures.

Learners do not count towards quorum. A pool with learners has fewer voters than nodes, and the limit follows the voter count.

Rollout

Validation

Regular conformance validation using the USS qualifier and extended validation similar to v0.23.0 will be conducted including performance characterization on the release cloud deployment.

We do not plan to attempt interoperability between a deployment with the existing implementation and a deployment with raft because this would require significant development effort and could prove challenging to implement properly:

- Both implementations may slightly differ (within specifications).
- Results are partially random (e.g. generated ID).
- Results under load are nondeterministic (due to order of operations, that are definitely not consistent between both clusters)
- A single operation accepted or not may lead to cascading different results, making it very hard to debug
- Result may not be even sharable as production data has been historically kept private for good reasons, making the validation results impossible to use to improve the software

Upgrade to raft

As of now, it's not planned to provide a way to upgrade from CoachroadDB or Yugabyte to Raft. This version of the design will require a zero-traffic downtime and recreation of existing data in the new instance.

Alternatives considered

Bulk load from database store

A tool could be implemented to read the existing database and insert data into a raft instance at pool initialization. This could be implemented with relatively low complexity, but will still require a zero-traffic downtime during the migration, to ensure consistency of the database.

Automatic migration with limited downtime

An option to initialize a raft store from a database cluster could be provided. Users would upgrade the DSS to the supported version and send an administrative command to start the migration. The migration would initialize an empty raft pool, stop traffic, load database data into raft, restart servicing traffic with raftstore.

Temporary mirrored execution

A system would write on the two clusters at the same time to allow switch when sufficient time has passed, ensuring flights are in both clusters.

This however would be challenging for the same reasons, as both implementations may return

different data (eg. different random id) and/or slightly diverse in their implementation, staying within specifications. Under load, handling failed operation would be problematic as well, should one cluster accept one but not the other, either by being overloaded or receiving operations in a different order across nodes in the cluster.

Upgrades

With the new implementation, upgrades will be doable in a rolling-upgrade manner, asynchronously. Versions are planned to be compatible with the previous one, without any need for zero-traffic upgrade or down time during those. The only restriction will be to potentially have to use intermediate versions should the version gap between the source and target version be too wide.

See “Multi-Version Interoperability” for technical details.

User documentation

Production readiness includes documented procedures for deployment, membership changes, upgrades, monitoring and TLS certificate management.

Initial performance evaluation

A proof-of-concept implementation has shown the following results. (#1594)

Scenario	Target	Measured (PoC)
SCD single-cell throughput	> 2 QPS	~95 QPS
SCD multi-cell throughput (116 sites)	> 45 QPS	~80 QPS
P95 latency	< 5s	< 300 ms
Error rate	< 1%	0 %

The Raft implementation outperformed the distributed SQL implementation in every scenario and met the targets with significant buffers. Under emulated latency, proposal payload size became the limiting factor; #1587 addresses it (see Cost).

TODO

Evaluate limits of the design using benchmarker and determine available buffer.

Risks and Mitigations

The proof-of-concept implementation passed the full DSS test suite (uss_qualifier and prober included) and outperformed the distributed SQL implementation in every performance scenario, meeting throughput, latency and error-rate targets. However, several operational risks, technical challenges, and edge cases must be managed.

Performance

Deployments of 24 nodes have not been tested yet. Under Raft, every added voting node widens the majority quorum required for commits and increases the required CPU and network bandwidth, potentially impacting write latency and throughput in large pools. If performance degrades to unacceptable levels, the voting set can be decoupled from the overall pool membership (e.g., maintaining a fixed subset of voters, such as one per participant). Exact thresholds and recommended procedures will be documented with the implementation.

Snapshots divergence: non-determinism

Nodes regularly calculate and exchange state hashes; if drift is detected, the affected node stops serving traffic and must be rebuilt from a snapshot.

Snapshots divergence: versioning

Downgrading code versions is explicitly unsupported because snapshots and state representations do not convert downward, and older code cannot process pool versions established by newer releases. If an incompletely-deployed release must be rolled back, each upgraded node must be removed, its data directory wiped, and re-added at the older release version before the rest of the pool advances its behavior version.

Upgrading code versions is limited by policy to the N+1 release in general to limit the complexity of dealing with many different versions of the behavior.

Data loss

If a node suffers local disk failure, corrupted snapshots, or an unreadable Write-Ahead Log (WAL), local recovery is impossible. This will surface as an error and the node will not start.

Starting with an empty WAL under an existing node ID could lead to severe state inconsistencies or split-brain behavior because the WAL contains information regarding votes cast and is therefore required to cast votes consistently. The required mitigation procedure is to treat the node as lost, discard its data directory completely, remove its old node ID via administrative proposal (dssctl), and add a new node instance with a fresh, unused node ID to catch up via snapshot streaming.

Node unreachability

Raft cannot distinguish between a permanently dead node and a temporarily network-partitioned node. Automated eviction is dangerous as it can cause unexpected quorum shifts or split-brain scenarios if partitions heal. Consequently, unreachable members are never

automatically evicted by the pool. Operators must manually decide whether to wait for recovery or explicitly remove the member.

Threat model

The Raft consensus model assumes all participants act in good faith and trust each other's operations. The protocol does not include an authorization layer to restrict validly authenticated nodes from proposing state, option, or membership changes. Accountability and mutual TLS via participant-operated CAs serve as the primary safeguards: every action is recorded in the Raft log with the proposing node's identity, until that part of the log is compacted at the next snapshot.

Blast radius

The 'blast radius' of this new implementation doesn't change: authentication on public ports is not changed (TLS with certificate and the same processes for certificate validation; ports that can be firewalled), however this is not a downgrade from the existing implementation and this provide already a state-of-the-art base protection, which could easily be improved by others means (e.g. use of dedicated, private links in a DSS pool). It has been therefore accepted for now but could always be revisited if needed.

Future Considerations

Payload

Continuation of #1587 (payload optimization) as further bottlenecks are identified. Initial implementation showed payload size, not consensus, as the first limit under emulated latency.

Sharding

If demand exceeds what a single Raft group can serve, the data can be partitioned into independent groups, removing the single-leader per service bottleneck. The thesis prototype partitioned the airspace into static S2-cell-bounded shards, one Raft group each, with cross-shard transactions coordinated by two-phase commit through a dedicated group. Three isolated shards gave three times single-group throughput on shard-local workloads; cross-shard workloads paid a coordination cost that a production design would first need to reduce. No requirement calls for this today.

Determinism under mixed-CPU architecture

Conduct a comprehensive risk analysis and establish safeguards to allow nodes running on different CPU architectures, compilers, or instruction sets to maintain deterministic execution.

Snapshots divergence

Evaluate the performance and availability trade-offs for state hash check frequency (e.g., per proposal, fixed interval, snapshot, or on-demand), define automated recovery actions upon drift detection, and build advanced diagnostic tooling for comparing live cluster snapshots.

Advanced debugging capabilities

Visually expose through the web interface the internal state of the nodes. Data like raft proposals, log, leadership status, etc. can be useful to access easily in order to debug issues. For now the only recourse is manually going through the logs, which can be done but is not very practical. Feedback from users once the system is being tested will also be valuable to determine what should be surfaced.

Migrations

If required by end-users, develop standalone utilities to convert snapshots to older format versions for backward compatibility, as well as zero-downtime online migration tools from CockroachDB/YugabyteDB.

Raft configuration fine-tuning

Performances, memory and disk usage optimizations obtained by fine-tuning the raft configuration. This includes e.g. the number of proposals to keep in memory, the compaction interval, etc.

Other open points

Points that remain open:

- Eviction of unreachable members (see Availability)
- Snapshot compare/dump tooling (see Determinism)
- Dynamic reload of CA configuration to avoid restart when changing trusted list of participants

Conclusion

The proposed Raft V1 technical design successfully provides a robust foundation for future traffic growth. Proof-of-concept implementation demonstrated that all key functional and non-functional requirements are met except the 24-nodes, which still needs evidence:

- Functional requirements: The design passes the complete DSS test suite, including `uss_qualifier` and proper conformance suites, while preserving the existing ASTM REST APIs and authentication models unchanged.
- Performance targets: Evaluated on a 9-node pool under emulated country-scale latency, the implementation significantly outperformed the distributed SQL system, achieving ~95 QPS on single-cell scenarios (against a >2 QPS target), ~80 QPS on multi-cell scenarios (against a >45 QPS target), p95 latency under 300 ms (against a 5 s bound), and a 0% error rate.
- Architectural constraints: the design achieves cloud portability, eliminates external database management overhead, and survives the loss of a minority of the pool participants.
- Governance: Operational equality is preserved across pool participants.

Having successfully passed functional practicability and performance fitness, the design provides a roadmap for production readiness, detailing multi-version interoperability, deterministic execution, and cluster operational lifecycle management.

References

Standards

- ASTM F3548-21, Standard Specification for UAS Traffic Management (UTM) UAS Service Supplier (USS) Interoperability, ASTM International. <https://store.astm.org/f3548-21.html>
- ASTM F3411-22a, Standard Specification for Remote ID and Tracking, ASTM International. <https://store.astm.org/f3411-22a.html>

Consensus

- D. Ongaro and J. Ousterhout, In Search of an Understandable Consensus Algorithm, USENIX Annual Technical Conference, 2014. <https://raft.github.io/raft.pdf>
- D. Ongaro, Consensus: Bridging Theory and Practice, PhD dissertation, Stanford University, 2014. Referenced for cluster membership changes and learners. <https://github.com/ongardie/dissertation/tree/master>

Databases evaluated

- CockroachDB. <https://github.com/cockroachdb/cockroach>
- Cockroach Labs, Get ready for CockroachDB's new enterprise license, 15 August 2024. <https://www.cockroachlabs.com/blog/enterprise-license-announcement/>
- Cockroach Labs, CockroachDB licensing FAQs. <https://www.cockroachlabs.com/enterprise-license-update/>
- YugabyteDB. <https://github.com/yugabyte/yugabyte-db>

InterUSS

- InterUSS DSS implementation. <https://github.com/interuss/dss>
- InterUSS monitoring tools, including `uss_qualifier` and `prober`. <https://github.com/interuss/monitoring>
- TSC investigation of CockroachDB mitigation strategies. https://github.com/interuss/tsc/blob/main/investigations/crdb_migration/README.md
- CockroachDB mitigation approaches (must-haves and desirables). https://github.com/interuss/tsc/blob/main/investigations/crdb_migration/%5BInterUSS%5D%20CockroachDB%20mitigation%20approaches.pdf
- InterUSS DSS 2024 redesign document. https://github.com/interuss/tsc/blob/main/investigations/crdb_migration/InterUSS%20DS%202024%20redesign.docx
- Raft-based application-level consensus, August 2026 update, discussion #1633. <https://github.com/interuss/dss/discussions/1633>

- Architecture and progress tracking, issue #1463. <https://github.com/interuss/dss/issues/1463>
- Performance measurements, issue #1594. <https://github.com/interuss/dss/issues/1594>
- In-memory storage evaluation, issue #1526. <https://github.com/interuss/dss/issues/1526>
- Proposal payload optimisations, issue #1587. <https://github.com/interuss/dss/issues/1587>
- Raft transport, pull request #1464. <https://github.com/interuss/dss/pull/1464>
- Raft metrics, pull request #1545. <https://github.com/interuss/dss/pull/1545>
- raftstore package README. <https://github.com/interuss/dss/blob/master/pkg/raftstore/README.md>
- Performance targets for the US UTM implementation, InterUSS issue #1541. <https://github.com/interuss/dss/issues/1541#issuecomment-4846753840>
- Proof-of-concept implementation, InterUSS pull request #1538. <https://github.com/interuss/dss/pull/1538>
- “Distributed transactions ensuring deconfliction of long range drone flights” EPFL Master Thesis by Mariem Baccari. https://github.com/interuss/tsc/blob/main/investigations/crdb_migration/mscthesis_consent_evaluation.pdf
- Technical Design Document structure and review, InterUSS issue #1639. <https://github.com/interuss/dss/issues/1639>
- Apache License 2.0, under which InterUSS software is released. <https://www.apache.org/licenses/LICENSE-2.0>

Libraries

- etcd Raft library. <https://github.com/etcd-io/raft>
- S2 Geometry Library, used for airspace volume indexing. <https://s2geometry.io/>
- go implementation of the S2 Geometry Library. <https://github.com/golang/geo>