# Purpose

This InterUSS document attempts to capture the design intent (currently in proposal phase) of the 2024 redesign of the InterUSS discovery and synchronization (DSS) implementation.

# Contributors

Michael Barroco (Orbitalize)

Callum May (Wing)

Benjamin Pelletier (Wing)

Punam Verma (NASA)

# Background

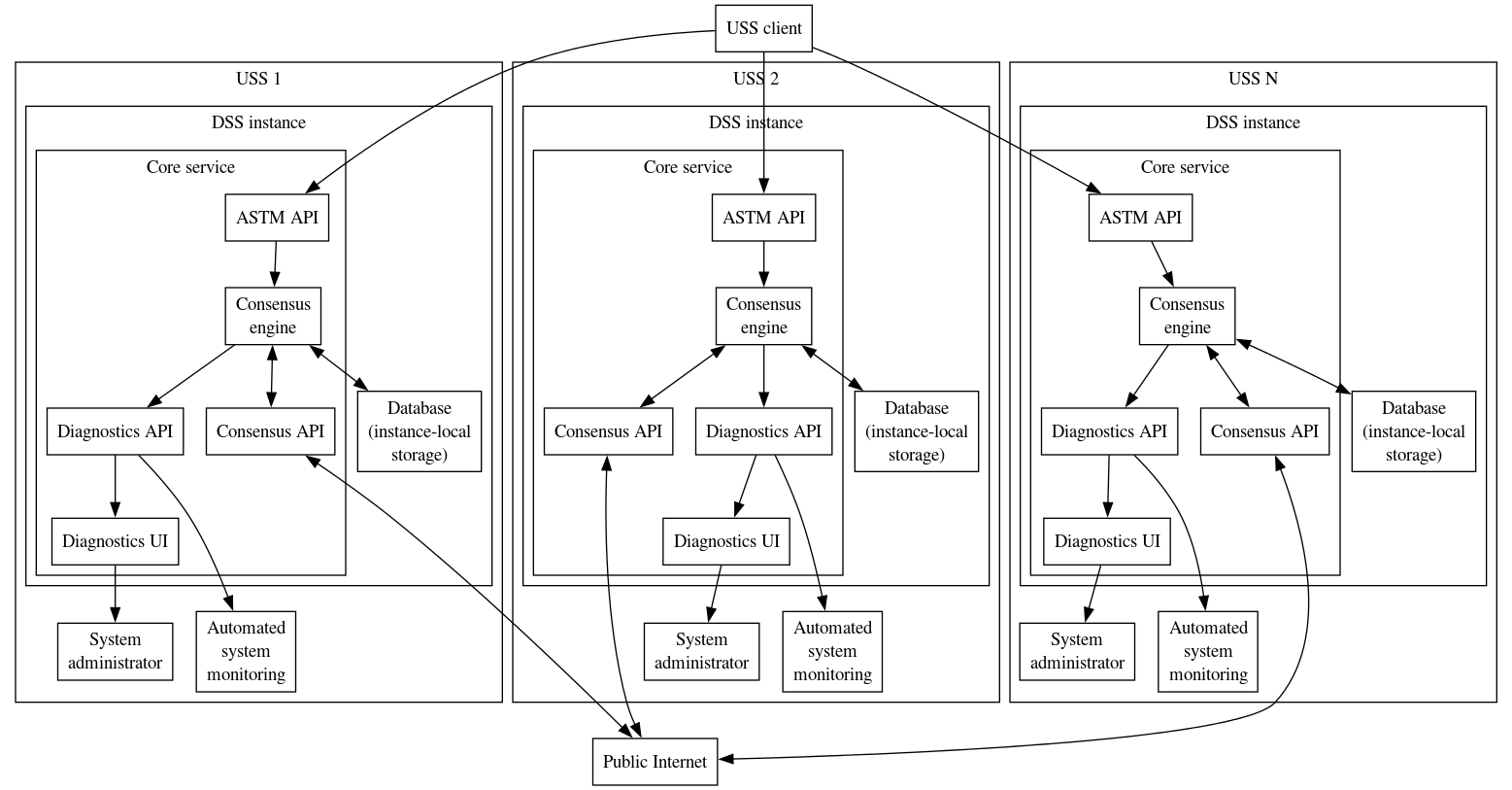
InterUSS’s initial DSS implementation developed in 2019 used CockroachDB as both a DSS instance’s local store of DSS airspace representation (DAR) data, and as the mechanism to synchronize that data across instances. Cockroach Labs announced a change in licensing terms for future releases of CockroachDB that are inconsistent with InterUSS’s open-source approach to aviation standards and implementations, prompting InterUSS to investigate alternatives to the use of CockroachDB.

This investigation included potential approaches such as:

* Tolerating new CRDB license terms
* Selecting a different (but similar) database product to achieve both synchronization and instance-local storage
* Selecting a different technology (e.g., NoSQL, distributed key-value store) to achieve both synchronization and instance-local storage
* Performing consensus via means other than Raft

The outcome of the investigation, however, was that the design described in this document most effectively achieved must-haves while maximizing the most important desirable characteristics.

# System overview



## Request fulfillment

A DSS pool consisting of InterUSS DSS instances will synchronize by storing DAR information into a [Raft](https://raft.github.io/) group whose membership is the full set of DSS instances in the pool. A USS client will call the ASTM API Interface (same as today’s implementation). InterUSS’s DSS implementation will fulfill a request to the ASTM API by obtaining Raft consensus that this request has been accepted, and this will be accomplished by the consensus engine.

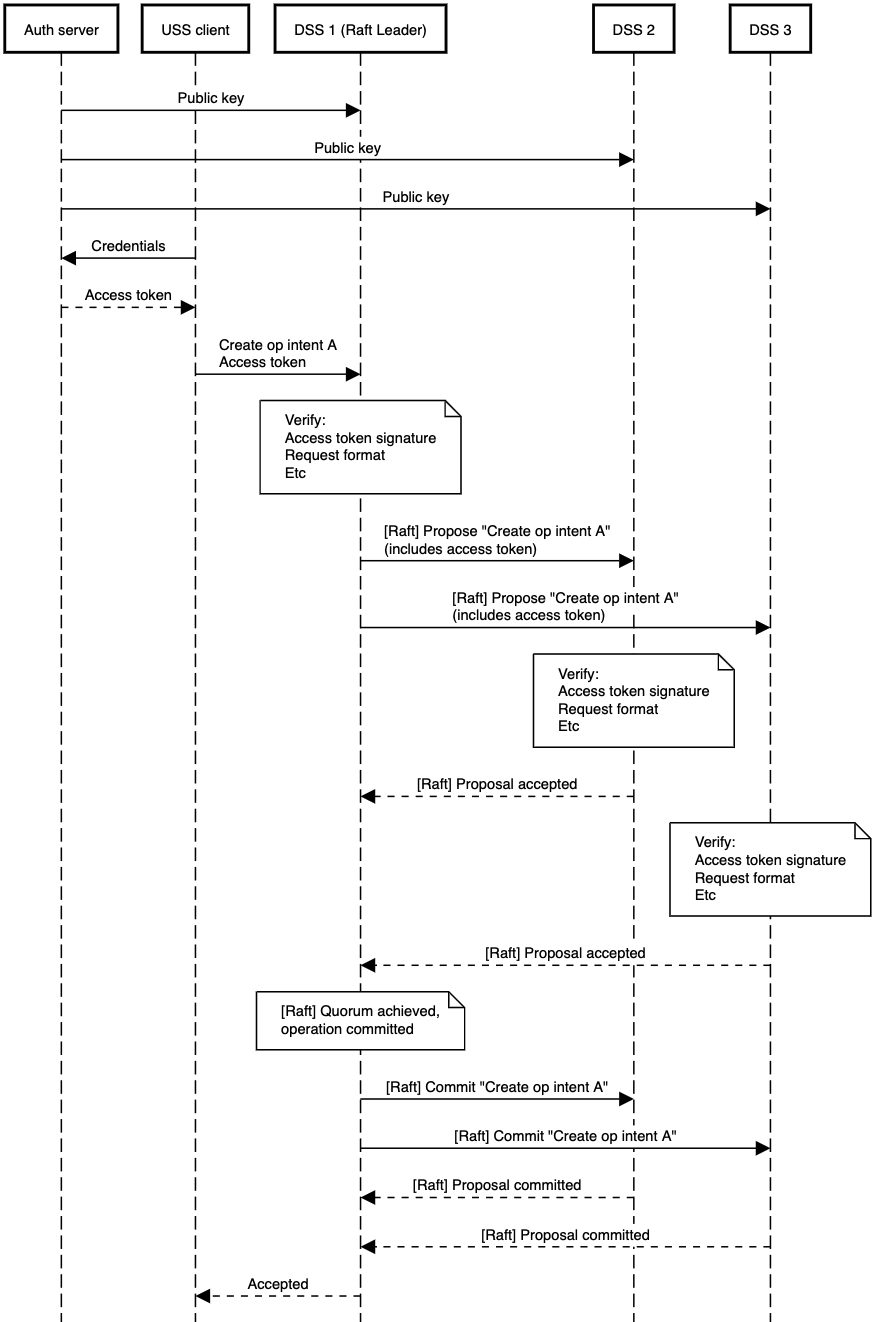
## Consensus engine

The consensus engine will use [the etcd Raft library](https://github.com/etcd-io/raft). The means provided to the Raft library to persist data locally will be via a connection to an instance-local [rqlite](https://rqlite.io/) database, though sophisticated users may choose to substitute a different database solution in their instance as long as it supports the minimum features identified by InterUSS. The means provided to the Raft library to communicate between Raft members will be calls to a consensus API exposed by the other Raft members. The consensus API will be a gRPC interface in order to achieve maximum performance (as compared to a RESTful https interface similar to the ASTM standards). The design of this consensus API will be part of the development effort.

## Raft specifics

### Log message content

The content of a Raft log message will generally be the API operation being performed (e.g., create operational intent), as received from the USS client, including the client-provided access token with redacted signature. When the Raft leader attempts to replicate this log message to followers, each follower will evaluate the log message for validity using the same validation on the request as if they had received it themselves, including verifying provided access token (though an audience of the other DSS instance will be accepted, the time of message receipt will be indicated when evaluating the expiration date of the token, and the signature will not be verified) and management of the primary resource being changed. Followers will reject a replication attempt for invalid messages.



### Log message storage

To store/commit a Raft log message, the consensus engine will first convert it into effects on primitives (e.g., operational intent entities, subscriptions), and then write those primitives to the database using nearly the same database schema as the current implementation. See [Database section](#_rx3juedal6im) below.

### Snapshot content

Snapshots will be provided as the sum of effects on primitives (i.e., the current internal database state for an instance), and not a list of API operations.

### Group membership

A Raft group composed of the DSS instances in a pool will determine whether a new member is valid by consensus of the current members of the Raft group. A single DSS instance will maintain a list of DSS instances it accepts/would accept as part of the pool, along with cryptographic verification of each instance (i.e., access token public key). A DSS instance will replicate a log message to add a DSS instance to the pool when the DSS instance to be added is among the existing DSS instance’s list of accepted DSS instances.

Therefore, the process for a new DSS instance to join a pool is:

1. New entrant provides the consensus API base URL of their new instance and the public key associated with the access tokens they will be using when making calls to other instances’ consensus API endpoints
   1. Note: the access tokens (or similar authorization) for the consensus API are entirely separate from the access tokens authorizing the standard APIs.
2. Each existing DSS instance operator adds this information to their DSS instance allowlist and notifies new entrant
3. After new entrant has received confirmation from a majority of existing DSS instances, new entrant starts their system which connects to the Raft group (using the consensus API base URL of one of the existing DSS instances) and adds itself as a new instance

# 

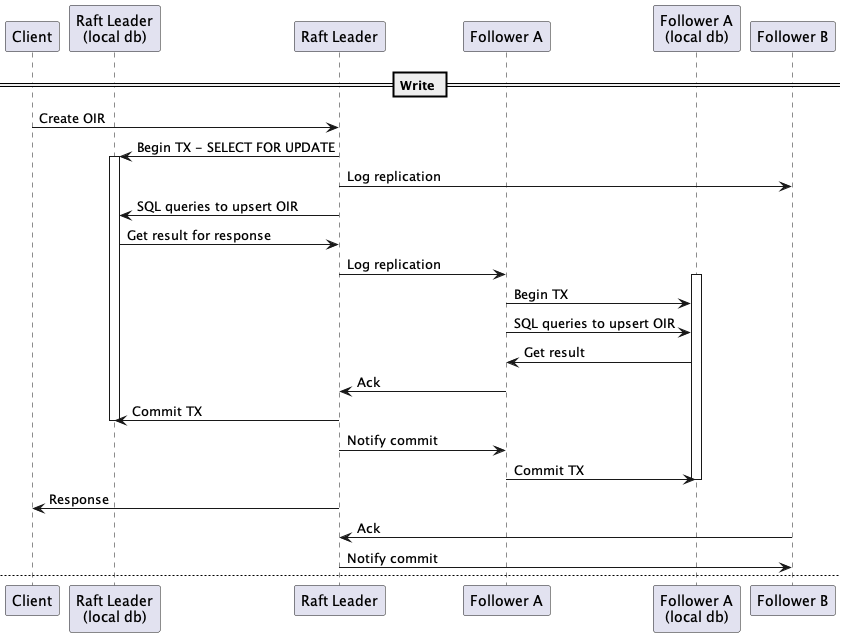
## Database (instance-local storage)

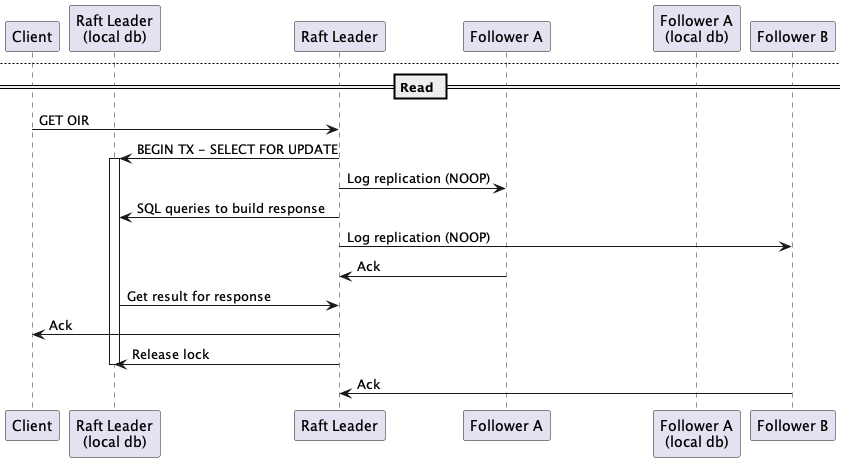
### Access patterns

The database local instance will be used to compute the responses returned to the client by the leader as well as building snapshots required by the raft protocol.

It will be critical to ensure that the messages are processed in the correct sequence on every node and that the local database only reflects committed information.

The following diagram is an attempt at outlining the interactions between the local database instance and the raft node, ensuring that data can be projected and processed while keeping the ability to rollback in case a message is not committed.





### Schema Migration

Each DSS instance will need a database instance. Schema migration will have to be performed on every database instance before using a new core-service version requiring a schema change. Migrations will need to be designed in order to support multiple versions. The standard procedure to update the schema may look like this:

1. DSS Instances run core-service at version 1 which uses schema version A.
2. Every instance upgrades their database to schema version B which must be backward compatible.
3. DSS Instances progressively roll out core-service version 2.

# Geographical sharding

## Development phases

Phase 1 of development will consist of finishing, to production level, a DSS implementation that uses a single Raft group to store DAR information for the entire world (no geographical sharding). Phase 1 development will anticipate Phase 2 where a performance optimization will be added to subdivide the world into smaller shards, and the DAR information for each shard will be stored in a separate Raft group dedicated to that shard (though each separate Raft group will still have each DSS instance in the pool as members). ASTM API operations affecting only one shard can be performed directly on that single shard, which enables horizontal scaling by decoupling geographical regions from each other in nominal usage.

Phase 1 will not implement any shard coordination, but it will attempt to avoid making the addition of shard coordination in Phase 2 any more difficult than necessary by assuming that the concept of multi-shard coordination exists, and reflecting that assumption in data structures, API design, etc. The data contained in the DAR is inherently geospatial (operational intents for example contain their geospatial data in their volumes) and thus multi-sharding would only require agreeing upon the geospatial boundaries, and would not require adding additional data or context to any requests.

## Phase 2 shard coordination

Any ASTM API operation whose effects lie entirely within a single shard will behave and perform identically to Phase 1. Because of this, ASTM API operations in separate shards can be fully parallelized without any appreciable impact on each other, and this enables horizontal scalability. When an ASTM API operation impacts multiple shards, an expensive two-phase commit (2FC) will be used to coordinate changes to the multiple shards.

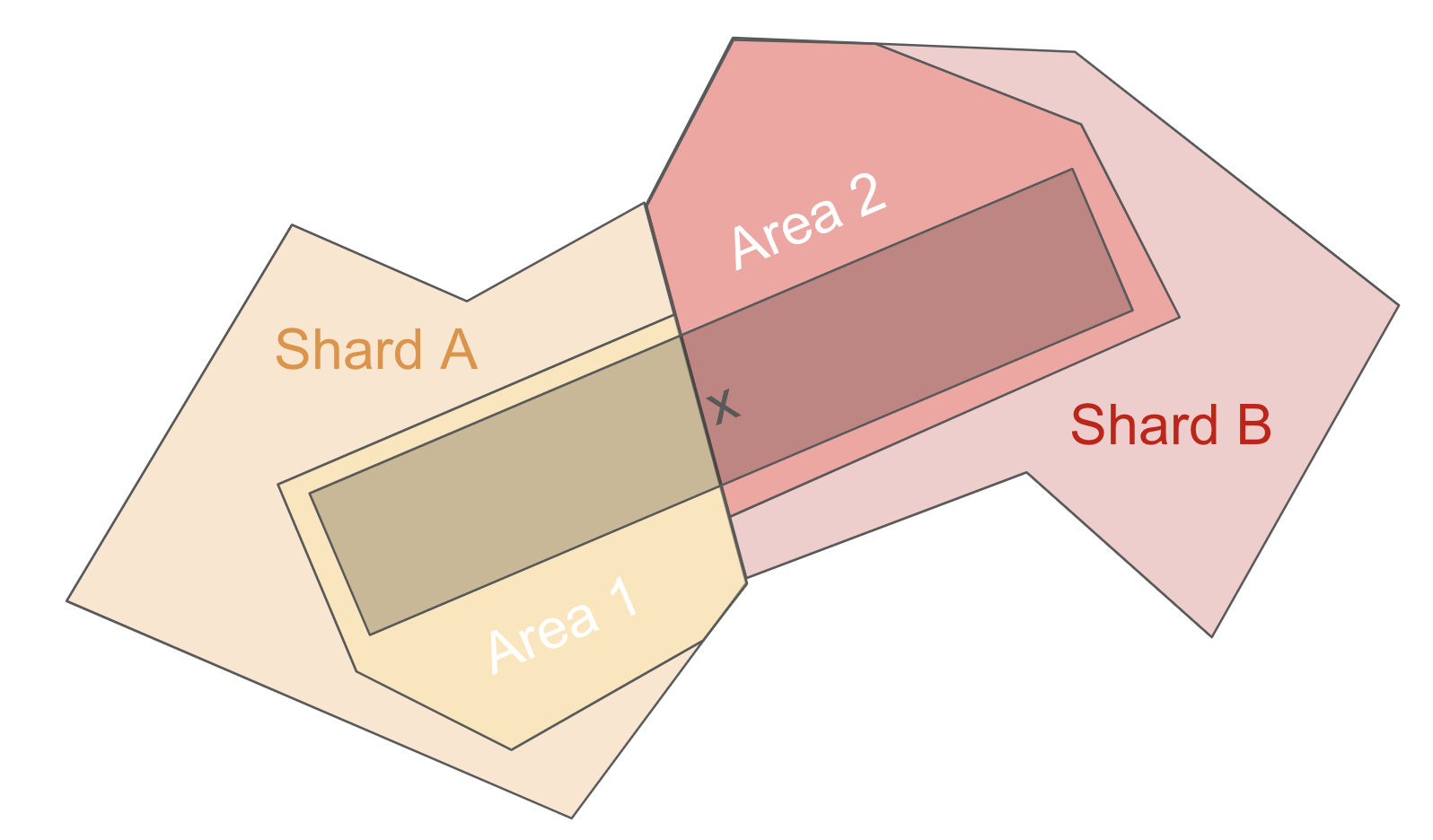
Every DSS instance will know the geographical shard boundaries and corresponding Raft group IDs out of band, or synchronized via a mechanism to be designed. An additional “transaction coordination” Raft group will be established between the DSS instances, similar to the Raft group for each shard. The transaction coordination raft group will be deployed separately from the geographical DAR shards, to increase resilience to outages affecting the shards. The leader of the transaction coordination Raft group will be considered the transaction coordinator, and every DSS instance will direct ASTM API operations that affect multiple geographical shards to the transaction coordinator for fulfillment. To fulfill a multi-shard request, the transaction coordinator will:

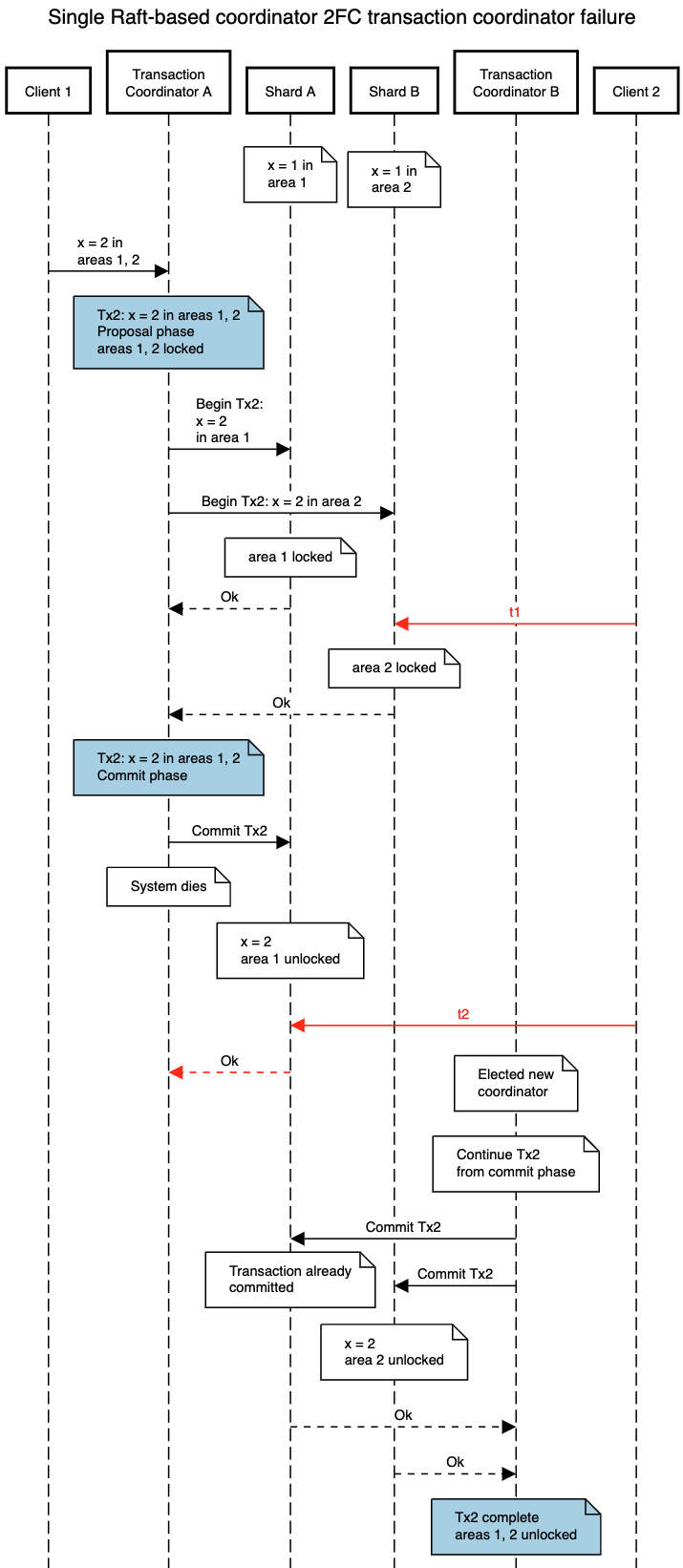
1. Record transaction X in the proposal phase in the transaction coordination Raft group, and set all affected areas as locked in that group
   1. Note that individual geographical shards are not aware of the area locks established in this step
2. Propose transaction X to all affected shards. Each affected shard will:
   1. Verify the validity of the proposal
   2. Lock the affected areas in the geographical shard Raft group
   3. Record the pending change in the geographical shard Raft group
   4. Respond ok to the transaction coordinator
3. Upon positive acknowledgement from all affected shards, record that transaction X is now in the commit phase in the transaction coordination Raft group
4. Instruct each affected shard to commit transaction X. Each affected shard will (atomically, within that shard):
   1. Apply the pending change in the geographical shard Raft group
   2. Mark the pending change as complete in the geographical shard Raft group
      1. (or, delete pending change in a way that allows the affected shard to still determine that the transaction is already complete if a duplicate request was sent to commit the transaction – for instance, by using a monotonically-increasing transaction index and maintaining only the largest completed transaction index)
   3. Unlock the affected areas in the geographical shard Raft group
   4. Respond ok to the transaction coordinator
5. Upon positive acknowledgement from all affected shards, remove transaction X and locks on affected areas from the transaction coordination Raft group

The above procedure is expected/understood to guarantee consistency at all times. If the transaction coordinator goes down, a new transaction coordinator will be elected leader of the transaction coordination Raft group, and it can continue any open transaction by repeating the requests for the phase of that transaction (resend proposals or resend commit instructions) and then proceeding with the transaction.

Only one multi-shard transaction affecting any particular area may be conducted at a given time; additional concurrent requests will be queued behind the active transaction. This is accomplished via area locks at the transaction coordination level.

An example sequence of events is shown below in the presence of an inopportune failure of the transaction coordinator.





# Diagnostic information

A recognized drawback to this approach is that InterUSS will need to develop any and all status and diagnostic infrastructure related to the consensus activities whereas this infrastructure is available for free when leveraging an existing product (like CRDB, YugabyteDB, rqlite, etc) to synchronize information across DSS instances.

## Automated access to status and diagnostic information

When information is relevant to understanding system status and/or diagnosing system behavior, it will be available via an automated diagnostics API which will be a RESTful https API in the style of ASTM APIs, using access token authorization in the same way as ASTM APIs.

## Human access to status and diagnostic information

When a human system administrator is likely to need to view the diagnostic information, a diagnostics UI will provide access to information in the automated diagnostics API via a browser. The human user will be responsible for providing an access token to the UI, and the UI will use that access token to call and display the results from endpoints in the automated diagnostics API.

# Deployment & Maintenance

InterUSS acknowledges the resource constraints faced by participants and is committed to delivering a solution that is easy to deploy and maintain. At the same time, it offers well-defined API contracts to allow participants the flexibility to implement custom designs if desired.

To streamline deployment, InterUSS will offer a pre-packaged container, similar to the existing core-service, which will encapsulate all consensus logic and include SQL bindings for the persistence layer. Additionally, for greater convenience, InterUSS will provide an integrated second/separate container with a database, simplifying the setup of the persistence layer for participants. This approach ensures that users have the option to either use the provided turnkey solution or implement their own configurations with minimal overhead.

Maintenance will be similar to the current architecture - updates will deploy via new container versions.

## Migration

To minimize required development effort, DSS instance hosts are expected to deploy this new implementation by tearing down their CockroachDB-based implementation and bringing up an instance of the new implementation in their place. This upgrade process is expected to require a pause in operations for the duration of this update. InterUSS does not intend to provide a live migration plan from the CockroachDB implementation to the implementation described in this document.

# Phase 1 execution plan

Milestones will be achieved on the redesign-2024 branch of InterUSS dss repository

| Milestone | Success criteria |
| --- | --- |
| core-service functionality is replaced by/augmented with “hello world” etcd-raft example | * Multiple core-service instances can be brought up and formed into a Raft group * Data can be written to one core-service instance and read from a different core-service instance (though perhaps via simple/temporary APIs or other means), and that data is synchronized via etcd-raft |
| Performance baseline | * A Raft group is deployed across multiple regions with a realistic number of participants (ie east and west coast) * An evaluation of the actual performance can be identified as a baseline and compared with the expected load |
| Diagnostic access is established | * Basic information about the Raft group (e.g., members, leader identity) is available in a diagnostics API (API not complete) * Information above is visible in a browser |
| core-service uses gRPC API to communicate between Raft members | * Current version of this consensus API is documented in /interfaces * Calls to this API are used for all inter-member Raft communication |
| All Raft functionality is supported | * New core-service instance can be added at runtime * Existing core-service instance can be removed at runtime * A minority of fully-initialized core-service instances can fail but allow the remaining core-service instances to continue to operate * A restored core-service instance automatically resynchronizes and can eventually continue all activity as if it had never gone down |
| core-service persists data using a SQL database | * From an established core-service Raft group, one core-service can be restarted (without restarting its underlying database instance) and continue operation without snapshot catchup |
| core-service exchanges ASTM-API-call information, but stores entity-level information in the database | * Triggering a particular action causes ASTM-API-call information to be exchanged between Raft group members, but entity-level information is stored in the database for that action * A new node can be added to the Raft group (demonstrating that snapshots work) |
| One existing read and one existing write ASTM API endpoint in core-service is serviced via etcd-raft | * All previously-demonstrated Raft functionality can be verified by using the ASTM read and write endpoints |
| All existing ASTM API endpoints in core-service are serviced via etcd-raft | * All previously-demonstrated Raft functionality can be verified by using any appropriate ASTM read and write endpoints |
| Redesigned system is deployable | * Someone not involved in the redesign development effort can follow instructions in the repository to bring up a DSS instance to form a new pool * …or join an existing pool (of redesigned instances; no backwards compatibility with CRDB expected) |
| Redesigned system is ready for production | * All legacy content not applicable to the redesigned system has been removed from the repository * Diagnostics support is sufficient for an operator to maintain a deployment at a production level * A cross-cloud DSS pool has been deployed and successfully tested with existing DSS automated testing (uss\_qualifier) |

# Known risks

When considering tradeoffs between different approaches, the following risks were identified and deemed sufficiently acceptable to proceed with development.

## Large scope of work

Designing and implementing our own consensus mechanism, even building on a well-established and tested library like etcd-raft, is expected to involve a relatively large amount of work and therefore a moderate amount of risk in the final completion date of that large scope of work. We intend to mitigate this risk by estimating the amount of work required for each milestone and ensuring the sum does not exceed the timeframe in which we need to deliver the final result. We also intend to minimize the presence of “unknown unknowns” by limiting the volume of work between milestones so that any unknowns in how to complete that smaller volume of work can be more easily identified.

## Accuracy and integrity of distributed system

Distributed systems are hard to design and even harder to implement fully correctly when considering all the edge and corner cases. Because we are not relying on a third-party product to have done this work for us, including extensive verification and real-world use, we intend to mitigate this risk by using a well-established library (etcd-raft) to perform all the core distributed system logic for us in Phase 1. This use of “vanilla Raft” via a well-established third-party library limits the opportunity for mistakes to a much smaller domain than the entire distributed system as a whole.

This risk will be re-examined before commencing Phase 2 development since we will be implementing a somewhat-novel distributed system in that case.

## Performance

Especially with the Phase 1 system, this approach is expected to be less performant than using the hyper-optimized CockroachDB consensus system previously used. With the Phase 1 system, all writes and all consistent reads for an entire DSS region/pool must go through the single Raft leader for that DSS region/pool, incurring latency between the responding DSS instance and the Raft leader, and between the Raft leader and Raft followers. Overall system latency is expected to scale with the median DSS instance latency (the slowest system needed to achieve a majority when replicating a new log entry), including both network traversals (e.g., if the median follower is geographically distant from the leader) and request processing (e.g., if the median follower’s database VM uses a slow hard drive and/or instance-internal distributed consensus rather than a solid state drive in a single VM).

We believe this performance profile will be sufficient even to support country-wide DSS pools, but we have mitigated the risk that evaluation may be incorrect by identifying the Phase 2 work that would allow us to arbitrarily parallelize most operations in the event this level of performance was unexpectedly insufficient.

# Appendix

## Raft Quorum with Validation

participant "Auth server" as Auth

participant "USS client" as Client

participant "DSS 1 (Raft Leader)" as DSS1

participant "DSS 2" as DSS2

participant "DSS 3" as DSS3

Auth -> DSS1: Public key

Auth -> DSS2: Public key

Auth -> DSS3: Public key

Client -> Auth: Credentials

Auth --> Client: Access token

Client -> DSS1: Create op intent A\nAccess token

note over DSS1: Verify:\nAccess token signature\nRequest format\nEtc

DSS1 -> DSS2: [Raft] Propose "Create op intent A"\n(includes access token)

DSS1 -> DSS3: [Raft] Propose "Create op intent A"\n(includes access token)

note over DSS2: Verify:\nAccess token signature\nRequest format\nEtc

DSS2 --> DSS1: [Raft] Proposal accepted

note over DSS3: Verify:\nAccess token signature\nRequest format\nEtc

DSS3 --> DSS1: [Raft] Proposal accepted

note over DSS1: [Raft] Quorum achieved,\noperation committed

DSS1 -> DSS2: [Raft] Commit "Create op intent A"

DSS1 -> DSS3: [Raft] Commit "Create op intent A"

DSS2 --> DSS1: [Raft] Proposal committed

DSS3 --> DSS1: [Raft] Proposal committed

DSS1 --> Client: Accepted

## System overview GraphViz source

digraph {

node [shape=box]

USSClient [label="USS client"]

subgraph cluster\_USS1 {

label="USS 1"

subgraph cluster\_Instance1 {

label="DSS instance"

subgraph cluster\_CoreService1 {

label="Core service"

ASTMAPI1 [label="ASTM API"]

ConsensusEngine1 [label="Consensus\nengine"]

ConsensusAPI1 [label="Consensus API"]

DiagnosticsAPI1 [label="Diagnostics API"]

DiagnosticsUI1 [label="Diagnostics UI"]

ASTMAPI1 -> ConsensusEngine1

ConsensusEngine1 -> ConsensusAPI1 [dir=both]

ConsensusEngine1 -> DiagnosticsAPI1 -> DiagnosticsUI1

}

DB1 [label="Database\n(instance-local\nstorage)"]

ConsensusEngine1 -> DB1 [dir=both]

}

{

rank="same"

SysMon1 [label="Automated\nsystem\nmonitoring"]

SysAdmin1 [label="System\nadministrator"]

}

DiagnosticsAPI1 -> SysMon1

DiagnosticsUI1 -> SysAdmin1

}

subgraph cluster\_USS2 {

label="USS 2"

subgraph cluster\_Instance2 {

label="DSS instance"

subgraph cluster\_CoreService2 {

label="Core service"

ASTMAPI2 [label="ASTM API"]

ConsensusEngine2 [label="Consensus\nengine"]

ConsensusAPI2 [label="Consensus API"]

DiagnosticsAPI2 [label="Diagnostics API"]

DiagnosticsUI2 [label="Diagnostics UI"]

ASTMAPI2 -> ConsensusEngine2

ConsensusEngine2 -> ConsensusAPI2 [dir=both]

ConsensusEngine2 -> DiagnosticsAPI2 -> DiagnosticsUI2

}

DB2 [label="Database\n(instance-local\nstorage)"]

ConsensusEngine2 -> DB2 [dir=both]

}

{

rank="same"

SysMon2 [label="Automated\nsystem\nmonitoring"]

SysAdmin2 [label="System\nadministrator"]

}

DiagnosticsAPI2 -> SysMon2

DiagnosticsUI2 -> SysAdmin2

}

subgraph cluster\_USSN {

label="USS N"

subgraph cluster\_InstanceN {

label="DSS instance"

subgraph cluster\_CoreServiceN {

label="Core service"

ASTMAPIN [label="ASTM API"]

ConsensusEngineN [label="Consensus\nengine"]

ConsensusAPIN [label="Consensus API"]

DiagnosticsAPIN [label="Diagnostics API"]

DiagnosticsUIN [label="Diagnostics UI"]

ASTMAPIN -> ConsensusEngineN

ConsensusEngineN -> ConsensusAPIN [dir=both]

ConsensusEngineN -> DiagnosticsAPIN -> DiagnosticsUIN

}

DBN [label="Database\n(instance-local\nstorage)"]

ConsensusEngineN -> DBN [dir=both]

}

{

rank="same"

SysMonN [label="Automated\nsystem\nmonitoring"]

SysAdminN [label="System\nadministrator"]

}

DiagnosticsAPIN -> SysMonN

DiagnosticsUIN -> SysAdminN

}

USSClient -> ASTMAPI1

USSClient -> ASTMAPI2

USSClient -> ASTMAPIN

Internet [label="Public Internet"]

ConsensusAPI1 -> Internet [dir=both]

ConsensusAPI2 -> Internet [dir=both]

ConsensusAPIN -> Internet [dir=both]

SysAdmin1 -> Internet [style=invis]

SysAdmin2 -> Internet [style=invis]

SysAdminN -> Internet [style=invis]

}

## Geographical sharding sequence diagram

To be rendered by sequencediagram.org

title Single Raft-based coordinator 2PC transaction coordinator failure

participant Client 1

participant "Transaction\nCoordinator A" as txcA

participant Shard A

participant Shard B

participant "Transaction\nCoordinator B" as txcB

participant Client 2

note over Shard A: x = 1 in\narea 1

space -6

note over Shard B: x = 1 in\narea 2

Client 1 -> txcA: x = 2 in\nareas 1, 2

note over txcA#lightblue: Tx2: x = 2 in areas 1, 2\nProposal phase\nareas 1, 2 locked

txcA -> Shard A: Begin Tx2:\nx = 2\nin area 1

txcA -> Shard B: \nBegin Tx2: x = 2 in area 2

note over Shard A: area 1 locked

Client 2 -#red> Shard B: <color:#red>t1</color>

note over Shard B: area 2 locked

space -9

Shard A --> txcA: Ok

space 2

Shard B --> txcA: Ok

note over txcA#lightblue: Tx2: x = 2 in areas 1, 2\nCommit phase

txcA -> Shard A: Commit Tx2

note over txcA: System dies

space -2

note over Shard A: x = 2\narea 1 unlocked

Client 2 -#red> Shard A: <color:#red>t2</color>

Shard A --#red> txcA: Ok

space -4

note over txcB: Elected new\ncoordinator

note over txcB: Continue Tx2\nfrom commit phase

txcB -> Shard A: Commit Tx2

txcB -> Shard B: Commit Tx2

space -5

note over Shard A: Transaction already\ncommitted

space -2

note over Shard B: x = 2\narea 2 unlocked

Shard A --> txcB: Ok

Shard B --> txcB: Ok

note over txcB#lightblue: Tx2 complete\nareas 1, 2 unlocked