DISTRIBUTED SHARED MEMORY

Head First DSM

From Shared State to Verifiable Consistency Boundaries

26 chapters · 6 learning arcs · 1 running case

English Edition · DSM 0.1.2

LeanowTech

Source baselines: DSM 505e755 · examples 0eca40d

Contents

Foreword: DSM's Value and the Learning Path

A recurring distributed-systems risk is treating a local guarantee as if it were a global one. A successful local write does not mean that a remote node can already read it, and eventual replica convergence does not revoke business responses that have already been returned.

DSM is a distributed shared memory foundation for Java applications. It shares coordination state—such as route hints, shard ownership, and mergeable counters—across service instances. A single Runtime provides state identity, typed collections, local operations, online propagation, conflict handling, and replica repair. Teams can reuse these defined and tested mechanisms instead of implementing propagation, merge, and repair logic in every service.

That value has clear limits. Eventual convergence does not provide transactional consistency, and replica repair does not replay a complete business-event history. Orders, inventory, and payments retain their authoritative systems. Application teams still define business invariants, partition behavior, external fencing or idempotency, and the security and capacity standards of the deployment environment.

The book follows a distributed order-fulfillment control plane:

- The `route-hints` Register stores service routing hints.
- The `shard-owners` Lease manages shard ownership.
- The `request-counter` CRDT merges request counts from multiple nodes.

The six arcs form one continuous path. Arc 1 identifies suitable shared state and creates a Runtime. Arc 2 selects Register, Lease, or CRDT from business invariants. Arc 3 explains online propagation, loss, repair, and partition boundaries. Arc 4 integrates DSM through a domain adapter and Spring Boot. Arc 5 adds observability, security, lifecycle, and capacity responsibilities. Arc 6 tests the conclusions through real ports, federation, technology rejection, and a capstone.

Each chapter introduces a problem before its mechanism. The experiment acceptance card records the command, observable result, evidence level, and uncovered scope. After completing the book, a developer should be able to adopt or reject DSM for a particular state and defend that decision with code, fault experiments, and runtime evidence.

Continue with Chapter 1 to decide which states belong in DSM.

LEARNING ARC

See Shared State

Chapters 1–4

Separate business facts, coordination state and event history, then start one Runtime and observe a second node.

Chapter 1 —DSM Responsibilities and Applicability

Chapter objective: After completing this chapter, a developer can decide whether a state belongs in DSM by examining data responsibility and recovery behavior.

Learning objectives

1. Describe the coordination problem that DSM solves.
2. Distinguish business facts, coordination state, event history, and analytical data.
3. Explain why peer repair does not make DSM a database replacement.
4. Make an initial storage decision for the four data categories in the fulfillment case.

Prerequisites

- Ability to read basic Java interfaces.
- Familiarity with database transactions and multi-instance services.
- This chapter does not require a running DSM instance or knowledge of replication protocols.

Case progress

The fulfillment system runs several API and worker instances. They share service routes, order-shard ownership, and request counts. Orders, inventory, and payments already have authoritative systems.

This chapter defines the responsibility boundary. Chapter 2 creates the first Runtime.

The coordination problem DSM solves

Consider a dangerous proposal: a team sees that DSM replicates data and plans to move order state into DSM and remove the database.

The decisive question is not whether DSM can serialize an `Order`. Order state is a business fact that needs transactional constraints, audit history, and a determinate write result. Replica convergence after network recovery cannot reconstruct a lost payment or revoke a success response already returned to a customer.

DSM fits a different category: relatively small coordination state needed while services run, recoverable from peers or rebuildable from an authoritative source.

Divide storage by data responsibility

The architecture assigns three responsibilities:

- Transactional databases retain orders, inventory, payments, and other business facts.
- DSM Runtime retains route hints, leases, mergeable counters, and other coordination state.
- Messaging or event systems retain events that require reliable delivery and historical replay.

One application may use all three. Choosing DSM does not exclude a database or event system.

DSM through its public contract

`DsmRuntime` is the public facade for several typed collections:

```
<E extends DsmEntity<E>> DsmRegister<E> register(CollectionSpec<E> spec);

<E extends LeaseEntity<E>> DsmLeaseRegister<E> leaseRegister(
        LeaseCollectionSpec<E> spec);

<E extends DsmEntity<E>, S extends MergeableState<S>>
DsmCrdtCollection<E, S> crdt(
```

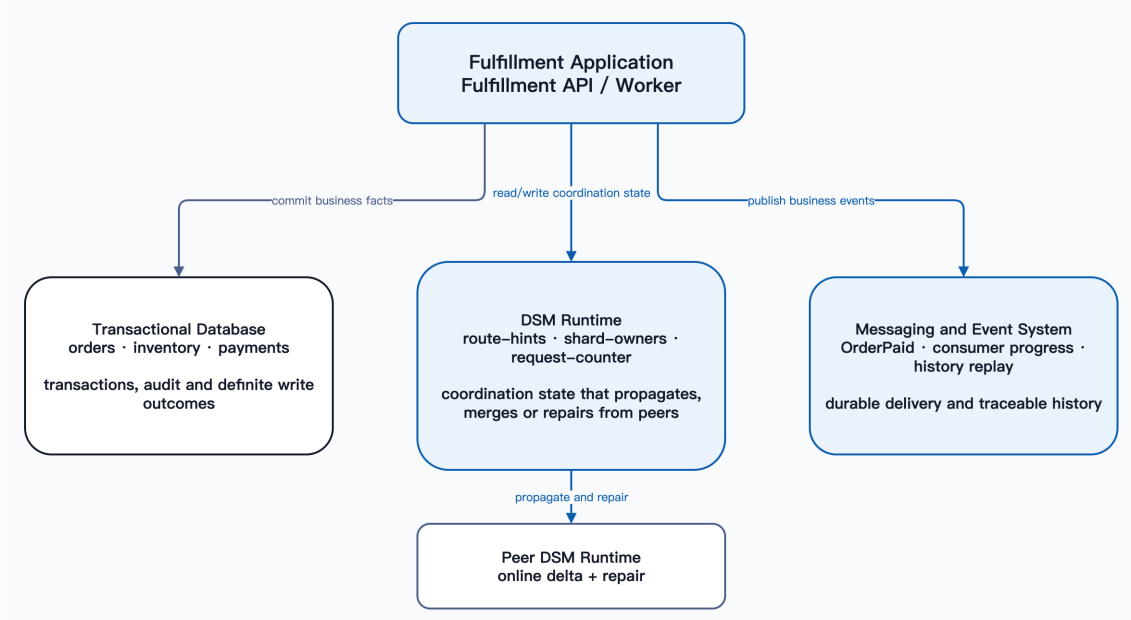

Figure 1: Data responsibility boundaries in order fulfillment

```
CrdtCollectionSpec<E, S> spec,
StateMerger<E, S> merger);
```

The entry points answer different coordination questions:

Question	Preferred collection	Book case
Which logical value should currently be visible for a key?	Register	`route-hints`
Who currently owns a time-bounded right to work?	Lease	`shard-owners`
How should local updates from several nodes merge?	CRDT	`request-counter`

Chapters 5–9 develop these semantics. At this stage, identify the business invariant before choosing a collection.

First verifiable increment: classify state

Open `classification-cases.json` and classify all eight candidates before reading the reference answer:

- `transactional-fact`: a business fact that needs transactions and audit.
- `coordination-state`: runtime state that can be repaired or rebuilt.
- `event-history`: events that need reliable delivery or replay.
- `analytical-data`: data intended for large-scale query and analysis.

Run the asset check:

```
npm test -- --test-name-pattern="chapter 01 classification"
```

The command validates the chapter asset and reference answer. It does not run DSM.

Responsibility details

Business facts depend on commit history

An order transition from `PENDING` to `PAID` usually accompanies a payment record, inventory change, and audit entry. The system identifies the successful transaction, the actor, and the compensation path through its transactional contract. DSM's collection API is not a cross-resource transaction protocol.

Orders, inventory deductions, and payment ledgers therefore stay in transactional systems.

Coordination state supports runtime decisions

A route hint tells a gateway which fulfillment instance to call. It may be temporarily stale and can be republished by a health source. If a node misses an online delta, repair can bring its current state forward.

That state can fit DSM, provided the stale window and failure behavior are explicit.

Event history needs delivery guarantees

`ChangeStream` observes collection changes. It is not a durable event log. A system that requires replay of every `OrderPaid` event uses a message platform, outbox, or event store.

Large-scale analysis needs a different data model

DSM collections center on keys and coordination semantics. They do not provide joins, secondary indexes, or analytical queries over large datasets. High-volume logs and metrics belong in corresponding data pipelines.

Counterexample and fault injection

Suppose inventory balance is stored in a Register:

1. Node A reads a balance of 1 and returns a successful deduction.
2. During a partition, node B also reads 1 and returns success.
3. After recovery, the Register deterministically retains one visible value.

The replicas converge, but two customers received success. Convergence did not revoke the second sale or create an auditable inventory transaction. DSM applicability therefore depends on whether business responses are reversible and whether transactional history is required—not only on whether an object is serializable.

Why this happens

A Register selects a deterministic visible result from candidate state. It does not coordinate several business resources or withdraw external effects.

Keep one boundary in view:

> Repair restores replica state; it does not repair business commitments already made outside DSM.

Revised state model

Replace “inventory balance”with “the preferred route to the inventory service”and ask:

1. Can a health source regenerate it after loss?
2. Can a caller retry or switch routes after reading a stale value?
3. Is complete modification history unnecessary?
4. Is a deterministic winner acceptable when publishers race?

Four positive answers make the route hint a better Register candidate than inventory balance.

Common misconceptions

Treating “memory”as a local cache

DSM state is managed by a Runtime and can propagate through sync and repair. It is neither a `Map` exposed to every JVM nor a simple local-cache wrapper.

Treating replicas as proof that data cannot be lost

Durability still depends on persistence configuration, topology, correlated failures, backup, and recovery. Several in-memory replicas do not replace database backups.

Selecting a collection before finding the business reason

“CRDT is the most advanced option”is not a selection criterion. Without concurrent local updates and a valid merge model, a CRDT only adds modeling and verification cost.

Understanding check

1. What makes a service-discovery address a possible Register candidate?
2. Why can an eventually convergent inventory value still oversell?
3. Should data requiring customer, time, and status queries use DSM as its primary store?
4. Why can a worker heartbeat fit DSM while the complete history of finished tasks does not?

Experiment

Classify these states as business facts, coordination state, event history, or analytical data:

- Current order status.
- Fulfillment-service route hint.
- Shard owner and lease expiry.
- Per-node request count.
- `OrderPaid` event history.
- Payment ledger entry.
- Request-latency histogram.
- Temporary feature flag.

Compare the result with the reference answer. A feature flag is not classified mechanically: its authoritative source, stale window, rollback needs, and business effect still matter.

Experiment acceptance card

Field	Content
Command	`npm test -- --test-name-pattern="chapter 01 classification"`
Input or fault	Eight candidate states and their responsibility descriptions
Observable result	The asset is structurally valid and the answer covers every candidate
Evidence level	E1: book assets and static responsibility boundaries only
Not proven	Runtime startup, replication, production capacity, or availability

Review

- DSM stores distributed coordination state; it is not a universal shared-data system.
- Business facts, coordination state, event history, and analytical data need different guarantees.
- Register, Lease, and CRDT address current value, ownership, and mergeable updates.
- Repair restores replica state; it does not revoke business responses.

Next

Chapter 2 starts a minimal `DsmRuntime`, creates the `route-hints` Register, and performs the first write and read.

Evidence links

- `DsmRuntime`
- `DsmRegister`
- `DsmCrdtCollection`
- DSM project constraints
- Phase 0 baseline validation

Chapter 2 —Start the First Runtime

Chapter objective: After completing this chapter, a developer can build a single-node `DsmRuntime`, register `route-hints`, and explain exactly what a successful local write means.

Learning objectives

1. Identify the responsibilities of Runtime, membership, and collection spec.
2. Apply the build, register, start, and close order.
3. Write and read the first `RouteHint`.
4. Distinguish local API success from remote visibility.

Prerequisites

- Completion of the Chapter 1 responsibility classification.
- JDK 25; the example repository already includes Maven Wrapper.

Case progress

The fulfillment gateway publishes `fulfillment-primary → 10.0.0.8:8080`. This chapter stores and reads the hint on one node. Network propagation comes later.

Four lifecycle steps

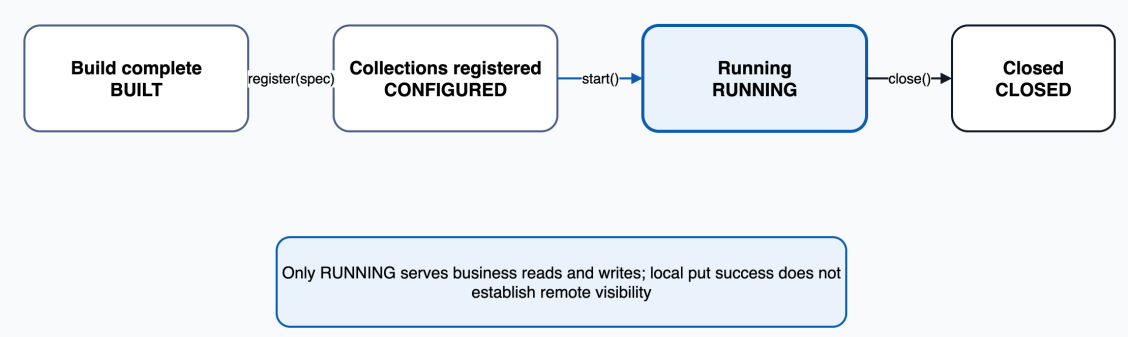

Figure 2: Lifecycle of the first Runtime

`DsmRuntimeBuilder` assembles the runtime. `ClusterMembership` supplies node identity and communication. `CollectionSpec` gives a collection a stable locator and schema. `DsmRegister` is the business-facing typed handle.

```
NodeInfo self = new NodeInfo("gateway-a", "127.0.0.1", 19090);
DsmRuntime runtime = DsmRuntimeBuilder.builder()
        .clusterId("fulfillment-local")
        .serviceId("gateway")
        .membership(new StandaloneClusterMembership(self))
        .build();

CollectionSpec<RouteHint> routes =
        CollectionSpecBuilder.<RouteHint>register("demo", "gateway", "route-hints")
                .schemaId("route-hints/v1")
                .recordCodec(RouteHint.class)
                .build();
DsmRegister<RouteHint> register = runtime.register(routes);
runtime.start();
```

Register collections before startup so the Runtime knows the schema when it enters `RUNNING`. Call `close()` after use. Tests and examples should use `try/finally` or an equivalent resource boundary.

First local commit

```
register.put(new RouteHint(
        "fulfillment-primary",
        EntityMetadata.empty(),
        "10.0.0.8:8080"));

String address = register.get("fulfillment-primary")
        .orElseThrow()
        .address();
```

The return from `put` means that this node accepted the local change. It does not report a replication acknowledgement count or promise that another node can already read the value. Single-node membership creates no remote replica.

Counterexample and fault injection

Treat the Runtime as externally ready before `runtime.start()`, or omit `close()`. The first mistake confuses completed configuration with an operational Runtime. The second leaks threads and resources. A successful local `get` also does not establish cluster-wide agreement.

Experiment

Run the upstream Runtime/REPL example test:

```
cd submodule/dsm-examples/basic-usage-examples
./mvnw -q -Dtest=DsmReplTest test
```

The command handler can store and list entries while the test creates and closes the Runtime explicitly. It uses controlled fake peers, so success proves the in-process path only, not real networking.

Experiment acceptance card

Field	Content
Command	`cd submodule/dsm-examples/basic-usage-examples && ./mvnw -q -Dtest=DsmReplTest test`
Input or fault	Single-node reads/writes and controlled test peers
Observable result	All three `DsmReplTest` cases pass and Runtime closes explicitly
Evidence level	E2: fixed-source in-process behavior
Not proven	Real networking, production discovery, remote write acknowledgement, or capacity

Review

- Runtime owns the lifecycle of collections and node cooperation.
- Collection specs are registered before `start()` and accessed through typed handles.
- Local `put` success proves a local commit, not remote visibility.
- Examples close Runtime so resource leaks do not hide behind a green test.

Next

Chapter 3 divides `RouteHint` into entry identity, entity fields, metadata, codec, and schema so both sides of replication interpret the same state.

Evidence links

- `DsmRuntimeBuilder`
- `RuntimeExampleFactory`
- `DsmReplTest`
- Phase 0 baseline validation

Chapter 3 —Stable Identity for Shared State

Chapter objective: After completing this chapter, a developer can define a `RouteHint` that nodes can identify, encode, and evolve consistently.

Learning objectives

1. Distinguish collection locator from entry key.
2. Explain how `DsmEntity`, `EntityMetadata`, and a codec cooperate.
3. Use `schemaId` to protect the wire-format compatibility boundary.
4. Detect identity drift and schema mismatch.

Prerequisites

- Ability to start the single-node Runtime from Chapter 2.
- Familiarity with Java records and serialization.

Case progress

Chapter 2 stored an address. This chapter gives it a stable identity: the collection is `demo/gateway/route-hints`, the entry key is the service name, and the entity value carries the address and health hint.

Five identity layers

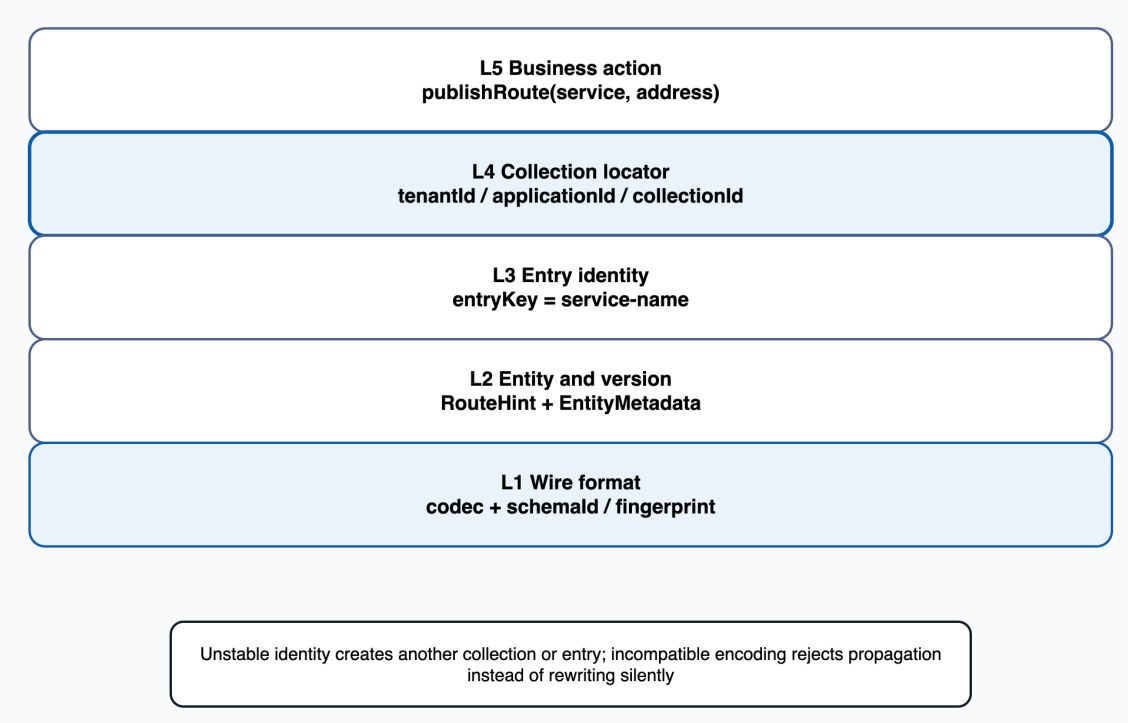

Figure 3: Stable identity of shared state

```
public record RouteHint(
        String entryKey,
        EntityMetadata metadata,
        String address)
        implements DsmEntity<RouteHint> {

    @Override
    public RouteHint withMetadata(EntityMetadata next) {
```

```
        return new RouteHint(entryKey, next, address);
    }
}
```

`entryKey` is the identity inside a collection and remains stable as content changes. `metadata` contains lineage, HLC, epoch, fencing token, and optional expiry. Business code leaves these protocol fields to the Runtime instead of forging them to win a conflict.

A collection locator consists of `tenantId/applicationId/collectionId`. Identically named keys under different locators are isolated entries. `clusterId/serviceId` defines the runtime communication domain and does not replace the locator. Chapter 17 tests both layers.

Codec and schema form a protocol boundary

Both nodes need to interpret the same bytes as the same entity. `recordCodec(RouteHint.class)` supplies record encoding. `schemaId("route-hints/v1")` records the wire-format intent in the collection contract. If one node treats `address` as a string and another changes it to an incompatible structure, rejection is safer than guessed conversion.

A safe evolution commonly deploys readers for old and new formats, switches writers, and then removes the old reader. Renaming a Java class does not complete a distributed schema migration.

Counterexample and fault injection

Create two nodes with the same locator and `schemaId` but incompatible codec meanings. Registration may succeed while remote decoding later fails. A rejected schema fingerprint protects visible state from incompatible data.

Another error uses a random UUID as the `entryKey` for every health publication. Each update then creates a new entry, so the Register never replaces the same logical service.

Experiment

Validate the identity card and run the fixed schema-mismatch integration test:

```
node --test tests/chapter-assets.test.mjs
cd submodule/dsm
mvn -q -pl dsm-integration-test -am \
  -Dtest=TwoNodeIntegrationTest#schemaFingerPrintMismatchIsRejectedGracefullyBetweenTwoNodes \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

The first check covers the book asset. The integration test proves explicit rejection between two incompatible nodes.

Experiment acceptance card

Field	Content
Command	Asset test plus the focused Maven command above
Input or fault	Schema fingerprint mismatch on the same route collection
Observable result	Asset passes; incompatible replication is rejected
Evidence level	E3: controlled two-node integration test
Not proven	Automatic schema migration, rolling cross-version upgrade, or production data repair

Review

- Locator identifies a collection; entry key identifies a logical entry.
- Business fields and Runtime metadata have different owners.
- Codec defines byte meaning; schema ID/fingerprint protects compatibility.
- Identity drift creates new state, while schema drift fails closed.

Next

Chapter 4 connects a second node and observes the same `RouteHint` moving from local commit to remote visibility.

Evidence links

- `DsmEntity`
- `EntityMetadata`
- `CollectionSpecBuilder`
- `TwoNodeIntegrationTest`

Chapter 4 —From Local Commit to Remote Visibility

Chapter objective: After completing this chapter, a developer can run a two-node replication experiment and use event origin to prove that remote visibility follows local commit.

Learning objectives

1. Connect two memberships and register the same spec on both nodes.
2. Track local put, delta, and remote apply.
3. Verify eventual visibility with a condition instead of fixed `sleep`.
4. State the evidence boundary of an in-process two-node test.

Prerequisites

- Completion of Chapters 2–3.
- Ability to distinguish locator, entry key, and schema.

Case progress

Gateway node A publishes `fulfillment-primary`. Node B eventually reads the same address from its Register. Order facts remain outside DSM.

From local commit to remote apply

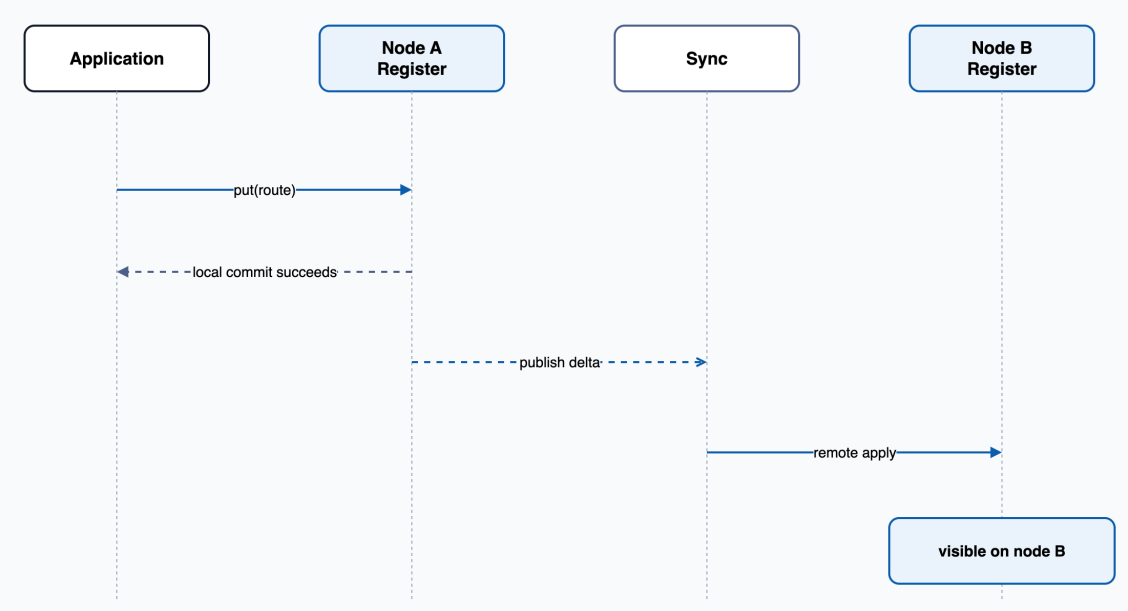

Figure 4: An update reaches a second node

Both sides need the same cluster/service communication domain, collection locator, compatible schema, and sync routing. Ordering matters: local `put` can return before the peer processes `REMOTE_DELTA`.

```
registerA.put(new RouteHint("fulfillment-primary", metadata, "10.0.0.8:8080"));
await(() -> registerB.get("fulfillment-primary").isPresent(), 2_000);
assertEquals("10.0.0.8:8080",
        registerB.get("fulfillment-primary").orElseThrow().address());
```

The condition expresses the acceptance target. A fixed `Thread.sleep(500)` can be too short and flaky or long enough to hide abnormal replication latency.

Let event origin carry evidence

`ChangeOrigin.REMOTE_DELTA` distinguishes remote apply from a local operation. A test that asserts both value and origin is stronger than polling `get` alone. `ChangeStream` is still not a durable event log and cannot preserve complete history while a subscriber is offline.

Counterexample and fault injection

Register the collection only on node A, or give node B a different `schemaId`. Connected membership alone does not route and apply collection data. A missing route or incompatible schema should surface explicitly; a timeout is not evidence that eventual consistency is merely slow.

Experiment

Run the focused two-node test:

```
cd submodule/dsm
mvn -q -pl dsm-integration-test -am \
  -Dtest=TwoNodeIntegrationTest#runtimeFirstRegisterDeltasReplicateAcrossTwoNodes \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

The test covers A→B put, A→B remove, B→A put, and remote event origin.

Experiment acceptance card

Field	Content
Command	Focused `TwoNodeIntegrationTest` command above
Input or fault	Two fake-membership nodes, bidirectional writes, and removal
Observable result	The peer observes state within a 2-second condition window and origin is `REMOTE_DELTA`
Evidence level	E3: controlled two-node protocol integration
Not proven	Cross-process networking, loss repair, production discovery, or linearizability

Review

- Local commit and remote visibility occur at different times.
- Membership alone is insufficient; collection routing and schema also match before application.
- Eventual conditions should poll the target state instead of sleeping for a guessed duration.
- `REMOTE_DELTA` proves online propagation origin, not complete event history.

Next

Chapter 5 develops the full Register contract for put, get, all, remove, and local visibility.

Evidence links

- `TwoNodeIntegrationTest`
- `RuntimePlatformSyncService`
- `ChangeOrigin`

Arc 1 Recap —Make Shared State Visible (Chapters 1–4)

Arc 1 moves from business responsibility boundaries to two-node visibility. Concurrent conflict, repair, and production networking remain for later arcs.

Understanding established in this arc

- Chapter 1 keeps orders, payments, and event history in their authoritative systems and assigns only repairable coordination state to DSM.
- Chapter 2 establishes Runtime lifecycle and limits successful `put` to local commit.
- Chapter 3 defines stable identity through locator, entry key, entity, metadata, codec, and schema.
- Chapter 4 observes online delta and eventual visibility on node B.

Verifiable learning outcomes

- Decide whether state can be repaired from peers or rebuilt from an authority.
- Build and close a single-node Runtime with a registered collection.
- Choose a stable key and explicit schema for `RouteHint`.
- Prove controlled two-node propagation with a condition and `REMOTE_DELTA`.

How this arc realizes DSM's value

Arc 1 turns "shared state"into inspectable responsibility boundaries, stable identity, Runtime lifecycle, and remote visibility. Services reuse registration, encoding, and basic propagation while retaining responsibility for state suitability and the point at which remote results become safe to depend on.

Four distinct moments

Moment	Proven	Not proven
`runtime.start()` returns	Local Runtime entered its operational lifecycle	Peer discovery or collection synchronization
`register.put()` returns	Local node accepted the change	Any remote node applied it
Node B `get()` hits	B currently sees the value	Every node agrees or no future conflict
Two-node test passes	Fixed source satisfies assertions under fake membership	Real networking, production capacity, or fault repair

Common errors

Error	Correction
Treat DSM as a transactional database	Reclassify from irreversible commitments and audit responsibility
Use a new entry key for every publication	Use stable business identity so Register updates one logical entry
Verify replication with `sleep`	Wait for the target condition with an explicit timeout
Treat a green test as production readiness	Record topology, transport, and uncovered boundaries

Transfer exercise

Design a temporary feature flag twice. In the first design, an authoritative configuration service exists and DSM stores a rebuildable hint. In the second, no authority exists and the flag directly controls a monetary action. Explain why the first may fit DSM and the second needs stronger fact and audit guarantees.

Next

Continue to Arc 2, where Register, Lease, and CRDT address current value, ownership, and mergeable updates.

LEARNING ARC

Choose Consistency Semantics

Chapters 5–9

Begin with business invariants and examine Register, Lease and CRDT conflict or merge semantics.

Chapter 5 —Register: Current Values and Stale Reads

Chapter objective: After this chapter, a developer can use a Register to represent the current logical value for a key and avoid treating a local query as a strongly consistent cluster read.

Learning objectives

1. Use the local `put/get/all/remove/size` contract.
2. Distinguish updating one key from creating another key.
3. Explain why removal also participates in replication and conflict ordering.
4. Define a stale-read policy for a route hint.

Prerequisites

- The two-node delta from Chapter 4 has been observed.
- Locator identity and entry keys are understood.

Case progress

The fulfillment service rolls from `10.0.0.1` to `10.0.0.2`. The gateway updates the existing `fulfillment-primary` entry and removes the hint after the old service retires.

One visible value per key

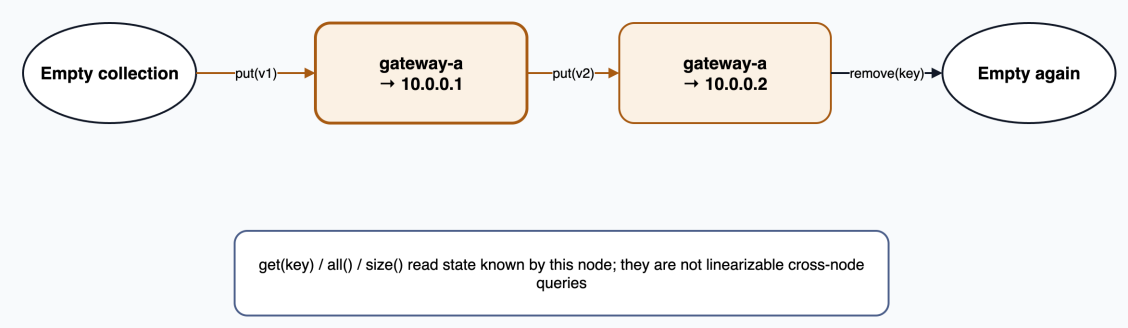

Figure 5: Visible operations on a Register

`put(v2)` submits a candidate with lineage metadata. It does not provide database row-lock overwrite semantics. The local node sees v2 immediately; peers catch up through online deltas or later repair.

```
register.put(route("fulfillment-primary", "10.0.0.1:8080"));
register.put(route("fulfillment-primary", "10.0.0.2:8080"));
Optional<RouteHint> current = register.get("fulfillment-primary");
Optional<RouteHint> removed = register.remove("fulfillment-primary");
```

`all()`, `size()`, and `get()` inspect the state currently known by one node. They support local routing and diagnosis. They do not create a cluster-wide snapshot or issue a read quorum.

The business stale-read policy

“Read from the Register”is incomplete as an operating rule. The gateway also defines whether a missing entry triggers an authoritative lookup, whether a failed connection tries a secondary address, and when a hint becomes too old to trust. DSM should not decide alone when a routing error can cause an irreversible effect.

In this case, a failed route hint triggers one lookup in the authoritative service directory and one retry. The fulfillment service’s transactional API remains responsible for idempotent order submission.

Counterexample and fault injection

A poor design writes every health probe under a random key and calls `all().get(0)` to select an address. Entries grow without bound, and their order carries no business meaning. A stable service key should identify the current hint; auditable history belongs in a history system.

Another weak assertion expects every node to be empty immediately after a removal. Removal propagates like a write and can meet concurrent candidates.

Experiment

Run the two-node put/remove/reverse-put test and record the local commit, remote put, and remote removal:

```
cd submodule/dsm
mvn -q -pl dsm-integration-test -am \
  -Dtest=TwoNodeIntegrationTest#runtimeFirstRegisterDeltasReplicateAcrossTwoNodes \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

Experiment acceptance card

Field	Content
Command	The focused Maven command above
Input or fault	Put and remove on the same key; a reverse-direction write
Observable result	The peer's visible value follows each operation; the remote removal has a source event
Evidence level	E3: controlled two-node integration test
This experiment does not prove	Strongly consistent cluster reads, history queries, or transactional conditional updates

Review

- A Register represents one current logical value per key, not complete history.
- Queries expose the local node's current view.
- Removal is a state change that requires ordering, propagation, and repair.
- The business defines fallback behavior for missing, stale, and failed hints.

Next

Chapter 6 lets nodes A and B update the same key during a partition and explains how both nodes later select the same winner.

Evidence links

- `DsmRegister`
- `RegisterLineageOrder`
- `TwoNodeIntegrationTest`

Chapter 6 —Concurrent Register Writes and Deterministic Convergence

Chapter objective: After this chapter, a developer can explain deterministic convergence of concurrent candidates and verify whether a custom resolver is commutative.

Learning objectives

1. Distinguish wall-clock time, HLC, and lineage total order.
2. Explain the deterministic basis of the default last-write-wins policy.
3. Write a commutativity check for a resolver.
4. Identify external effects that equal final values cannot repair.

Prerequisites

- The Register's local view and two-node propagation are understood.
- Each side may accept local writes during a network partition.

Case progress

Node A points `fulfillment-primary` east while node B points it west. After recovery, both sides need the same visible hint or traffic remains split indefinitely.

Determinism matters more than speed

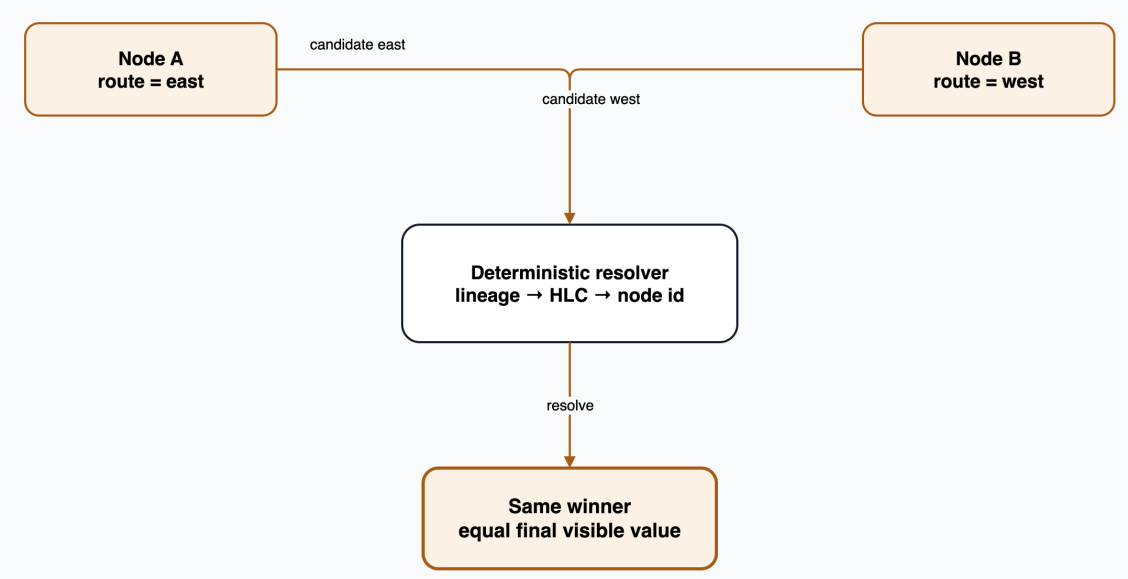

Figure 6: Deterministic resolution of concurrent candidates

The default `ConflictResolver.lastWriteWins()` compares metadata through `RegisterLineageOrder`. The order includes lineage/HLC information and uses a stable node identifier as a tie-breaker. Correctness depends on every node applying the same total order to the same pair of candidates; one machine's interpretation of local time cannot supply that guarantee.

At minimum, the following property holds:

```
resolve(A, B).winner == resolve(B, A).winner
```

A first-argument-wins resolver can leave A on east and B on west. Duplicate delivery and reversed message order do not heal the split.

The safe entry point for a custom resolver

`ConflictResolver.verified(delegate)` invokes the resolver with `(local, remote)` and `(remote, local)`, then compares the winning entity and outcome family. A violation raises `CONFLICT_RESOLVER_NOT_COMMUTATIVE`, which is useful in tests, staging, or a controlled rollout.

The check covers only the observed input pair and executes the resolver twice. It is not a formal proof. A resolver remains pure, deterministic, and free of external side effects.

Counterexample and fault injection

Define `resolve(local, remote) -> localWins(local)`. Each node treats its own candidate as local and the state remains split. A resolver that reads the current time or a random number can occasionally agree after argument reversal while still failing deterministic replay.

Even when two concurrent inventory deductions eventually choose one winner, both customers might already have received success. Conflict resolution converges replica state; it does not undo external commitments.

Experiment

Run the two-node custom-resolver test:

```
cd submodule/dsm
mvn -q -pl dsm-integration-test -am \
 -Dtest=TwoNodeIntegrationTest#customConflictResolversConvergeMergedRegisterValuesAcrossTwoNodes \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

Then inspect this chapter's `resolver-cases.json` and classify each candidate policy for purity, commutativity, and determinism.

Experiment acceptance card

Field	Content
Command	The Maven command above; `node --test tests/chapter-assets.test.mjs`
Input or fault	Concurrent candidates for one key; reversed input order
Observable result	Both nodes reach the same merged result; the asset marks unsafe strategies
Evidence level	E3: controlled two-node conflict test
This experiment does not prove	Formal correctness over every input, cross-resource transactions, or reversal of external effects

Review

- Concurrent writes require one total order or a commutative merge rule on every node.
- HLC supplies logical ordering, not a globally synchronized clock.
- A custom resolver remains pure, deterministic, and commutative.
- Convergence repairs replicas, not business effects that already escaped.

Next

Chapter 7 turns to time-bounded ownership: shard processing asks who can act within a lease window rather than which candidate is newer.

Evidence links

- `ConflictResolver`
- `RegisterLineageOrder`
- `TwoNodeIntegrationTest`

Chapter 7 —Lease: Time-Bounded Ownership

Chapter objective: After this chapter, a developer can express time-bounded shard ownership with acquire, renew, transfer, release, and fencing tokens.

Learning objectives

1. Distinguish a current value from a current right to act.
2. Explain the separate duties of TTL, epoch, and fencing token.
3. Describe renewal, transfer, release, and expiry.
4. Make an external resource reject an old holder.

Prerequisites

- Register conflict resolution is understood.
- Process pauses, network partitions, and time progression can leave an old owner running.

Case progress

A fulfillment worker claims shard `orders-17`. Only a valid holder can commit scheduling results for the shard. Ownership expires, and after a transfer the external write boundary rejects the old worker.

Lifecycle and guardrail

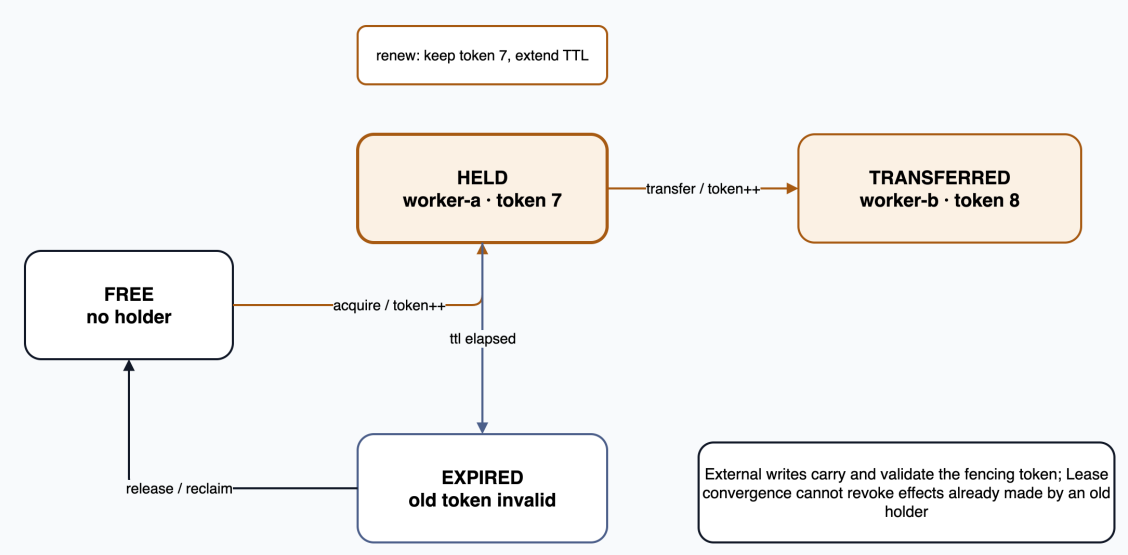

Figure 7: Lease lifecycle for shard ownership

```
lease.acquire("orders-17", acquireOptions).join();
lease.renew("orders-17", renewOptions).join();
lease.transfer("orders-17", "worker-b", transferOptions).join();
lease.release("orders-17", releaseOptions).join();
```

TTL answers when ownership expires. A fencing token lets an external system identify an old holder. A successful transfer creates a higher token; a database, object store, or downstream executor stores the last accepted token and rejects lower values.

When an external resource ignores the token, an expired worker can still write. Additional DSM replication cannot repair that boundary.

Why a stable member view matters

Lease operations coordinate high-risk actions. Blind reassignment during an unstable member view can leave two workers believing that they own a shard. The API and runtime signals define the current implementation's rejection and mode boundaries; a business service cannot rely on a single local boolean.

Counterexample and fault injection

Worker A pauses after receiving token 7. The lease transfers to B with token 8. A resumes and writes again. An executor that does not compare tokens accepts the old owner. A useful experiment checks both lease state and `FencingExecutor` rejection of token 7.

Experiment

```
cd submodule/dsm
mvn -q -pl dsm-integration-test -am \
  -Dtest=RuntimeIntegrationTest#fencingExecutorRejectsStaleHolderAfterLeaseTransfer \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

Besides checking the holder name, the test invokes the transferred resource with the old token to verify rejection at the external boundary.

Experiment acceptance card

Field	Content
Command	The focused `RuntimeIntegrationTest` command above
Input or fault	A holds the lease, transfers it to B, then acts with its old token
Observable result	B receives a higher token; the fencing executor rejects the old token
Evidence level	E2: integration behavior in one Runtime
This experiment does not prove	A real membership split, cross-process pause behavior, or correct token checks in every external store

Review

- A Lease represents a time-bounded right to act, not an ordinary latest value.
- TTL closes the ownership window; a fencing token protects the external side-effect boundary.
- Transfer advances the token monotonically because the old holder may still be alive.
- Complete fencing depends on token validation by the external resource.

Next

Chapter 8 lets every node update a counter locally and designs a mergeable state instead of selecting one owner.

Evidence links

- `DsmLeaseRegister`
- `LeaseState`
- `RuntimeIntegrationTest`

Chapter 8 —CRDT: Multi-Node Local Updates and Merge

Chapter objective: After this chapter, a developer can model multi-node local increments and decrements with a PN Counter and explain the algebraic properties required by its merge function.

Learning objectives

1. Explain state, update, version vector, and delta.
2. Distinguish CRDT merging from Register winner selection.
3. Explain why commutativity, associativity, and idempotence matter.
4. Identify business quantities that a counter alone cannot represent.

Prerequisites

- The concurrent candidates in Chapter 6 are understood.
- Request counts may differ temporarily, while duplicate merges cannot inflate them.

Case progress

The east node records three requests locally and the west node records two. After recovery, both sides reach five without taking a central lock for every request.

Merge state instead of choosing one winner

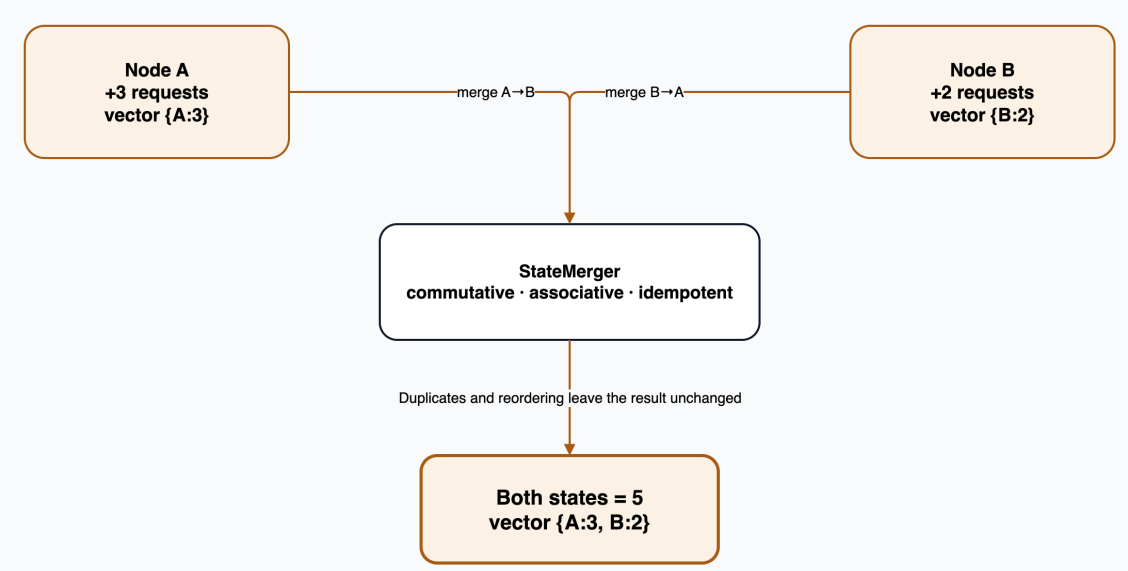

Figure 8: Local counters and their merge

The core `DsmCrdtCollection` operations are `apply(entity)` and `state()`. `versionVector()` describes observed node progress, and `deltaSince(vector)` produces state for a lagging peer.

```
counter.apply(new PnCounterUpdate("east-1", metadata, "east", 3, 0));
counter.apply(new PnCounterUpdate("west-1", metadata, "west", 2, 0));
long total = counter.state().value();
```

The pinned source defines the exact constructor parameters. The important structure is an independent contribution per node. Merge retains monotonic information for each component and then calculates the total. Receiving the same state twice does not add three twice.

Three required properties

- Commutativity: `merge(A, B) = merge(B, A)`, so message order does not change the result.
- Associativity: `merge(merge(A, B), C) = merge(A, merge(B, C))`, so repair batching does not change the result.
- Idempotence: `merge(A, A) = A`, so retries and duplicate delivery do not amplify state.

These properties describe merge. They do not make every business operation commutative. Balance deductions, inventory bounds, and at-most-once external effects need additional contracts.

Counterexample and fault injection

Suppose `merge(left, right) = left.total + right.total` stores the result as new state. Repeated delivery of the same snapshot doubles the total repeatedly. A correct PN Counter retains per-node positive and negative components, or an equivalent monotonic structure, instead of re-adding an already aggregated total.

Experiment

```
cd submodule/dsm
mvn -q -pl dsm-integration-test -am \
  -Dtest=RuntimeIntegrationTest#oneRuntimeMergesCrdtCounterUpdatesThroughTheRuntimeFacade \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

Then use `crdt-merge-cases.json` to calculate the results for reversed order, regrouping, and duplicate input.

Experiment acceptance card

Field	Content
Command	The Maven command above; `node --test tests/chapter-assets.test.mjs`
Input or fault	Two node components; order changes and duplicate input
Observable result	The Runtime facade returns the merged count; the expected results cover all three properties
Evidence level	E2: in-Runtime merge test plus book algebra cases
This experiment does not prove	Two-node network propagation, business deduplication, exactly-once billing, or production capacity

Review

- A CRDT merges contributions instead of retaining only one candidate.
- A version vector records observed progress; a delta can target a lagging peer.
- Merge is commutative, associative, and idempotent.
- Technical mergeability does not establish a business invariant.

Next

Chapter 9 places Register, Lease, and CRDT in one business-invariant decision tree for the first design review.

Evidence links

- `DsmCrdtCollection`
- `PnCounterStateMerger`
- `RuntimeIntegrationTest`

Chapter 9 —Choose Collections from Business Invariants

Chapter objective: After this chapter, a developer can begin with a business invariant and give a falsifiable, testable reason for choosing Register, Lease, CRDT, or a different system of record.

Learning objectives

1. Decide whether state fits DSM before selecting a collection.
2. Map an invariant to collection semantics and a failure boundary.
3. Define a minimal validation experiment for each decision.
4. Recognize state that belongs in a database or messaging system.

Prerequisites

- The three collection experiments in Chapters 5–8 are complete.
- Local success, remote visibility, and eventual convergence are distinct concepts.

Case progress

The team reviews the route hint, shard owner, and request counter separately. Each decision records adoption conditions, failure behavior, and rejection reasons. DSM is not assumed to fit every state.

Decision order

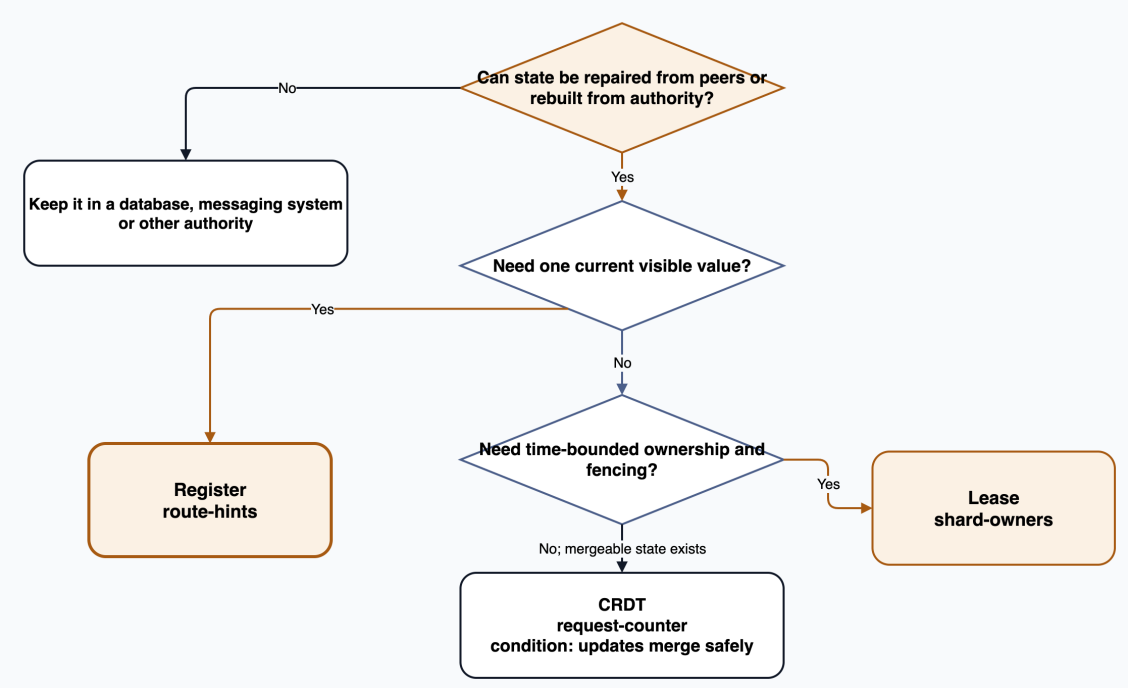

Figure 9: Choosing a collection from business invariants

The first question asks whether lost state can be repaired from a peer or rebuilt from an authoritative source. When it cannot, or when local success immediately creates an irreversible commitment, the state stays in a transactional or event-based system of record.

State	Key invariant	Choice	Partition window	Validation focus
route hint	one current visible hint per key	Register	the two sides may differ briefly	deterministic convergence and stale fallback
shard owner	time-bounded ownership; old owner cannot write	Lease	the old holder may still run	fencing rejects the old token after transfer
request counter	local updates merge safely	CRDT	a local count is incomplete	duplicate, reordered, and regrouped merges agree
order/payment	transactional fact and audit history	keep outside DSM as primary storage	two successes can be irreversible	transaction, idempotency, reconciliation, and audit

A choice is a testable hypothesis

"Lease supports ownership"does not complete a decision. A stronger version is:

> For `orders-17`, only the worker holding the newest fencing token can commit scheduling results. After transfer, the executor rejects the old token. A transfer test and an old-token counterexample validate the rule.

This statement identifies the actor, state, failure window, external responsibility, and evidence.

Counterexample and fault injection

- A Register can identify a newer shard owner but supplies no TTL, renewal, transfer, or fencing contract.
- A Lease for request totals forces all updates through one owner and discards local-write availability.
- A CRDT for order state does not establish legal state transitions, cross-resource transactions, or audit history.

Experiment

Open `collection-review.json` and fill in `decision`, `invariant`, `failureWindow`, and `evidenceCommand` for every scenario. Then run:

```
node --test tests/chapter-assets.test.mjs
```

The reference answer expects a business reason for each adoption and rejection, not only a collection name.

Experiment acceptance card

Field	Content
Command	`node --test tests/chapter-assets.test.mjs`
Input or fault	Seven business states with invariants and failure windows
Observable result	All three DSM collection types and the rejected cases are covered; required fields are present
Evidence level	E1: design-review asset check backed by E2/E3 experiments from preceding chapters
This experiment does not prove	Automatic fit for a new domain, production capacity, or organizational acceptance

Review

- DSM suitability is decided before a collection type.
- Register covers current values, Lease covers time-bounded ownership, and CRDT covers mergeable local updates.
- Every decision records a failure window and a validation command.
- Orders, inventory, and payments remain with transactional systems of record.

Next

Chapter 10 begins Arc 3 by identifying who belongs to the cluster and who is only suspected of being offline before discussing delta and repair.

Evidence links

- `DsmRegister`
- `DsmLeaseRegister`
- `DsmCrdtCollection`
- Chapter 1 responsibility boundary

Arc 2 Recap —Choose the Right Consistency Semantics (Chapters 5–9)

Arc 2 establishes a collection-selection process based on business invariants, failure windows, and validation methods.

What this arc established

- A Register retains one current logical value per key, viewed locally.
- Concurrent Register candidates converge through a deterministic total order or a commutative resolver.
- A Lease uses TTL and a fencing token for time-bounded ownership and depends on the external system rejecting old tokens.
- A CRDT permits local updates while its merge remains commutative, associative, and idempotent.
- State that cannot be repaired or rebuilt stays outside DSM despite convenient APIs.

Capability matrix

Business invariant	Collection	During a partition	After recovery	Validation focus
one current hint	Register	nodes may see different values	deterministic winner	reversed order agrees; stale fallback
time-bounded exclusive right to act	Lease	old holder may still run	new token identifies the valid owner	external resource rejects old token
multi-node local updates merge safely	CRDT	local state is incomplete	states converge after merge	commutativity, associativity, idempotence
transactional fact or reliable history	keep outside DSM as primary storage	local success may be irreversible	convergence cannot compensate effects	transaction, idempotency, audit, replay

How this arc realizes DSM's value

Register, Lease, and CRDT provide reusable coordination semantics for three recurring problem shapes. The value appears only when a collection matches the business invariant. The service team still records partition consequences, fencing responsibilities, and the authoritative source.

Five review questions

1. Which system owns the authoritative state?
2. Can lost state be recovered from a peer or rebuilt from that authority?
3. May both nodes succeed during a partition?
4. How do fencing, idempotency, or transactions protect external effects?
5. Which executable command proves the selected semantics rather than compilation alone?

Common mistakes

Mistake	Correction
model a Lease with a Register	include TTL, transfer, and external fencing in the contract
wrap any object in a CRDT	establish algebraic properties for the business merge first
read time or randomness in a resolver	keep it pure and deterministic; reverse inputs to check commutativity
test only the happy path	inject an old token, duplicate delta, changed order, and partition candidates

Migration exercise

Choose storage semantics for remaining coupon quantity, batch-worker ownership, and page-view counts. For each DSM choice, record the failure window. For each rejection, name the needed transaction or history guarantee. Selecting DSM for all three earns no special credit.

Next

Arc 3 decomposes “eventual consistency”into member views, online propagation, missed-update detection, and repair.

LEARNING ARC

Understand Replication and Repair

Chapters 10–14

Follow membership, delta, repair and change streams to establish observable consistency boundaries.

Chapter 10 —Member Views and Stability

Chapter objective: After this chapter, a developer can distinguish membership events, currently visible members, and a stable member view, and explain why a high-risk Lease cannot act on one observation.

Learning objectives

1. Explain discovery, suspicion, departure, and recovery signals.
2. Distinguish `activeMembers()`, `clusterSize()`, and stable cluster size.
3. Explain how `serviceId` prevents unrelated logical clusters from discovering each other.
4. Protect QUORUM Lease decisions with a membership-stability window.

Prerequisites

- The two-node replication experiment in Chapter 4 is complete.
- Lease fencing and QUORUM mode from Chapter 7 are understood.
- A temporary lack of response is not treated as permanent departure.

Case progress

After `node-b` joins the fulfillment cluster, `node-a` first sees a membership event and only later observes the same member count over consecutive rounds. During this unstable window, a worker tries to claim `orders-17`. The team needs to choose the membership view used as the quorum denominator.

One observation is not a global fact

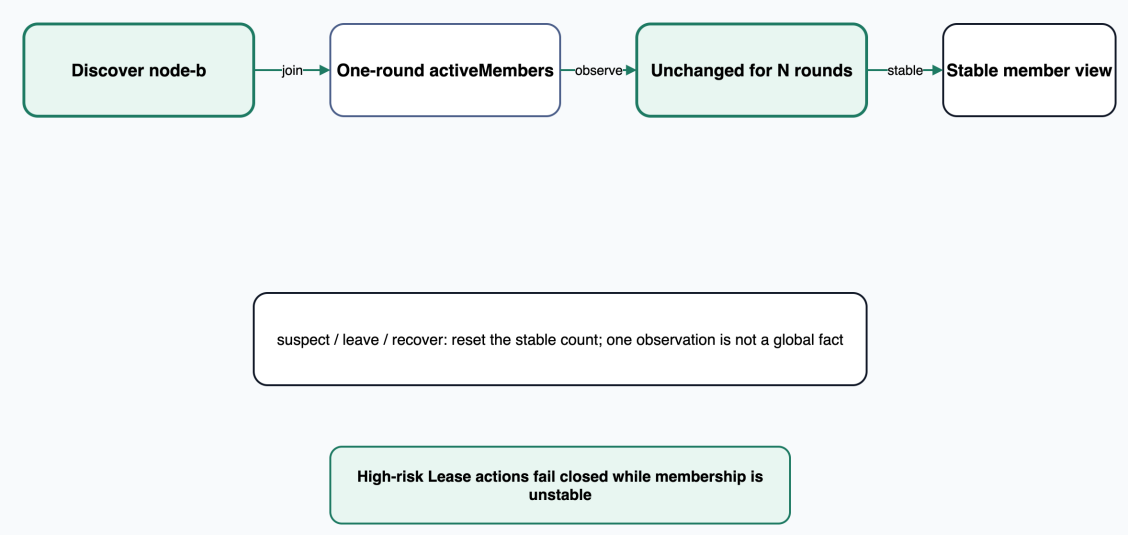

Figure 10: From member discovery to a stable member view

`ClusterMembership` exposes both membership observations and DSM control/data channels: `self()`, `activeMembers()`, `clusterSize()`, `membershipRoundInterval()`, and event/data listeners. These calls report one node's current observation, not a linearizable cluster roster.

Signal	What it shows	What it does not show
`NodeJoined`	this node received join information	every node now has the same view
`NodeSuspected`	failure detection suspects a node	the node has permanently failed
`NodeLeft`	this node confirmed or received a departure	the other side of a partition agrees
`activeMembers()`	currently reachable members	the original cluster size has changed

Signal	What it shows	What it does not show
`clusterSize()`	the implementation's cluster-size view	that size has stayed stable for several rounds

The test double deliberately separates `clusterSize()` from `activeMembers()`. During a partition, the original size remains while visible membership shrinks. A QUORUM check therefore cannot redefine one isolated node as the complete cluster and allow a write.

The purpose of a stability window

`MembershipStabilityTracker` observes member count across rounds. `stableClusterSize()` remains `-1` until the count reaches `quorumStabilityRounds`; any size change resets the counter.

With three required stable rounds:

Round	Observed size	Stable count	`stableClusterSize()`
1	2	1	-1
2	2	2	-1
3	2	3	2
4	3	0	-1
5–7	3	1–3	eventually 3

A QUORUM Lease returns `membership-unstable` while stable size is unknown and `quorum-unavailable` when a stable view lacks a majority. This fail-closed behavior pauses new leases instead of calculating a majority from a drifting denominator.

`serviceId` is the first isolation boundary

Development, test, and production clusters may share one network. Equal transport addresses do not make them one logical service. Membership implementations isolate discovery and messages with `serviceId`; nodes with different values stay out of each other's membership lists.

`serviceId` is also unsuitable as a business tenant identifier. It defines a cluster broadcast domain. Chapter 17 adds tenant/application isolation at the collection-locator layer.

Counterexample and fault injection

Partition one node from the other two in a three-node cluster. Its `activeMembers()` can contain only itself. If visible count `1` becomes total size, the majority threshold also becomes `1` and the isolated node incorrectly considers itself writable. Retaining the cluster size or waiting for a stable view causes the QUORUM Lease to reject the claim.

Experiment

Verify the membership failure model and the Lease gate:

```
cd submodule/dsm
mvn -q -pl dsm-test-support,dsm-runtime -am \
 -Dtest=FakeClusterMembershipChaosTest,MembershipStabilityTrackerTest,InMemoryDsmLeaseRegisterTest \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

Classify every item in `membership-observations.json` as an event, visible membership, total size, or stable view. No single event supports a claim that the cluster is globally consistent.

Experiment acceptance card

Field	Content
Command	The focused Maven tests above; `node --test tests/chapter-assets.test.mjs`

Field	Content
Input or fault	loss, delay, partition, suspicion, crash/recovery, and member-count change
Observable result	visible membership can shrink; stable size requires consecutive rounds; unstable QUORUM Lease rejects
Evidence level	E2: single-process membership and Lease behavior tests
This experiment does not prove	Cross-host network quality, multicast reachability, or suitable production failure thresholds

Review

- A member view is a local observation, not an instantaneous global fact.
- Visible membership and original cluster size serve different purposes.
- High-risk quorum decisions wait for consecutive stable rounds.
- `serviceId` isolates logical clusters; it does not replace tenant isolation.

Next

With a stable membership view, Chapter 11 follows one `route-hints.put` from local commit to remote visibility.

Evidence links

- `ClusterMembership`
- `MembershipStabilityTracker`
- `FakeClusterMembershipChaosTest`
- `InMemoryDsmLeaseRegisterTest`

Chapter 11 —The Remote Path of One Write

Chapter objective: After this chapter, a developer can observe local commit, delta encoding, transport, remote validation, and application separately, and explain why online propagation does not replace repair.

Learning objectives

1. Trace a Register update through five stages from local mutation to remote application.
2. Explain collection locator, schema, and delta-envelope responsibilities.
3. Separate control-plane messages from data-plane transfer.
4. Define a repairable boundary for missed propagation.

Prerequisites

- Register operations from Chapter 5 are complete.
- Member views and data content are distinct.
- A successful local `put` includes no remote acknowledgement.

Case progress

Route Publisher updates the healthy `payment` route to `10.0.0.8` on `node-a`. The local API sees it immediately while `node-b` briefly retains the old address. The team inspects each propagation stage instead of calling the whole condition "slow synchronization."

Five stages and five observation points

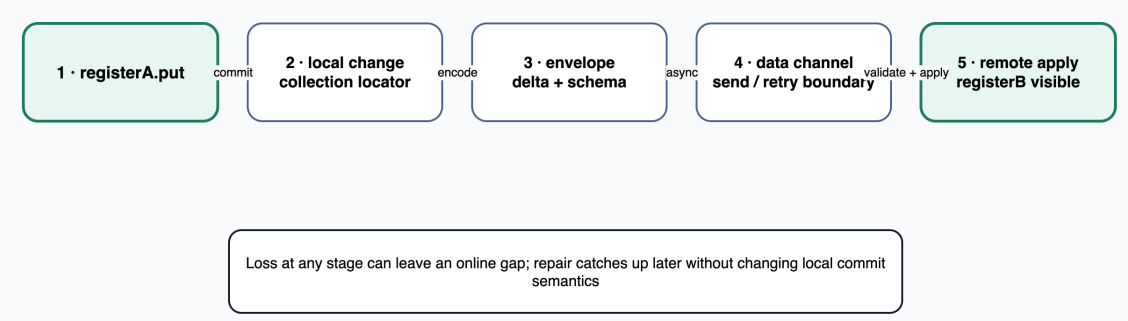

Figure 11: A delta from local commit to remote application

1. **Local commit:** the collection processes resolver/metadata rules and becomes readable locally.
2. **Collection location:** the change carries tenant, application, and collection identity.
3. **Envelope encoding:** delta and schema/capability information enter the wire envelope.
4. **Data channel:** sending, framing, retries, or loss occur at the transport boundary.
5. **Remote application:** the peer validates target and capability, then applies `REMOTE_DELTA` to the collection.

Stage 1 can return local success while Stage 5 happens later or never happens on the online path.

Identity precedes payload correctness

Two collections that both store `RouteHint` cannot automatically accept each other's deltas. `RuntimePlatformSyncService` addresses a collection route/locator and routes Register, Lease, or CRDT messages after capability negotiation.

A propagation contract answers:

- Which logical service sent the message?
- What is the target collection locator?
- Does the payload belong to Register, Lease, or CRDT?
- Are schema and codec compatible?

- Is this an online delta, replay item, or snapshot chunk?

Ambiguity in any answer can turn “message delivered”into an incorrect claim that state was applied safely.

The boundary of online delta

Online delta optimizes low-latency propagation under normal conditions. It is not a durable business-event log.

Condition	Possible online outcome	Follow-up responsibility
receiver temporarily offline	no real-time application	compare digest and repair after recovery
control message lost	difference remains undiscovered briefly	anti-entropy sweep detects it later
incompatible envelope	application rejected	record the reason and correct version/capability
duplicate arrival	idempotence or metadata arbitration needed	collection merge/resolver handles it
out-of-order arrival	old value cannot replace new	lineage, HLC, or collection semantics handle it

A ChangeStream consumer therefore cannot infer complete history. The stream observes collection changes and does not provide message-queue audit semantics.

Counterexample and fault injection

Drop the data path from `node-a` to `node-b` and perform a local `put`. A reads the new route, B retains the old route, and local success remains valid. After network recovery, repair closes the gap. An assertion against A proves only local semantics. An assertion against B without event origin cannot distinguish online delta from later repair.

Experiment

```
cd submodule/dsm
mvn -q -pl dsm-integration-test -am \
  -Dtest=TwoNodeIntegrationTest,ChaosIntegrationTest \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

Inspect `delta-trace.json` and confirm that every stage records input, output, an observable signal, and a continuation after failure.

Experiment acceptance card

Field	Content
Command	The focused Maven tests above; `node --test tests/chapter-assets.test.mjs`
Input or fault	local update, remote node, missed online message, and recovery
Observable result	local success occurs first; the normal path reaches the peer; repair catches a missed update
Evidence level	E3: in-process multi-node and chaos integration tests
This experiment does not prove	Cross-host throughput, real-network retry limits, or exactly-once business events

Review

- Local commit and remote visibility are separate completion points.
- Locator, collection type, and schema constrain the remote target together.
- Online delta favors low latency and does not guarantee complete history.
- Missed messages are an expected input; repair detects and closes gaps continuously.

Next

Chapter 12 lets a late `node-b` compare digests and select replay or snapshot with an explainable decision.

Evidence links

- `RuntimePlatformSyncService`
- `RegisterCollectionRoute`
- `TwoNodeIntegrationTest`
- `ChaosIntegrationTest`

Chapter 12 —Digest-Based Difference Detection and Replica Repair

Chapter objective: After this chapter, a developer can detect differences with a digest, explain replay/snapshot selection, and reject a repair that would move the requester backward.

Learning objectives

1. Connect digest, anti-entropy, and repair.
2. Select replay or snapshot from retention and estimated cost.
3. Detect regression when the requester is newer or more complete at the same sequence.
4. Separate current-state repair from business-event replay.

Prerequisites

- The delta path from Chapter 11 is understood.
- Online messages can be lost while collections still need eventual convergence.
- Repair is a background process outside the local write transaction.

Case progress

`node-b` misses ten route-hint updates while offline. On return it reports its collection digest. The source decides whether retained deltas can close the gap or a current-state snapshot is more appropriate; B does not request every business event.

From difference to repair plan

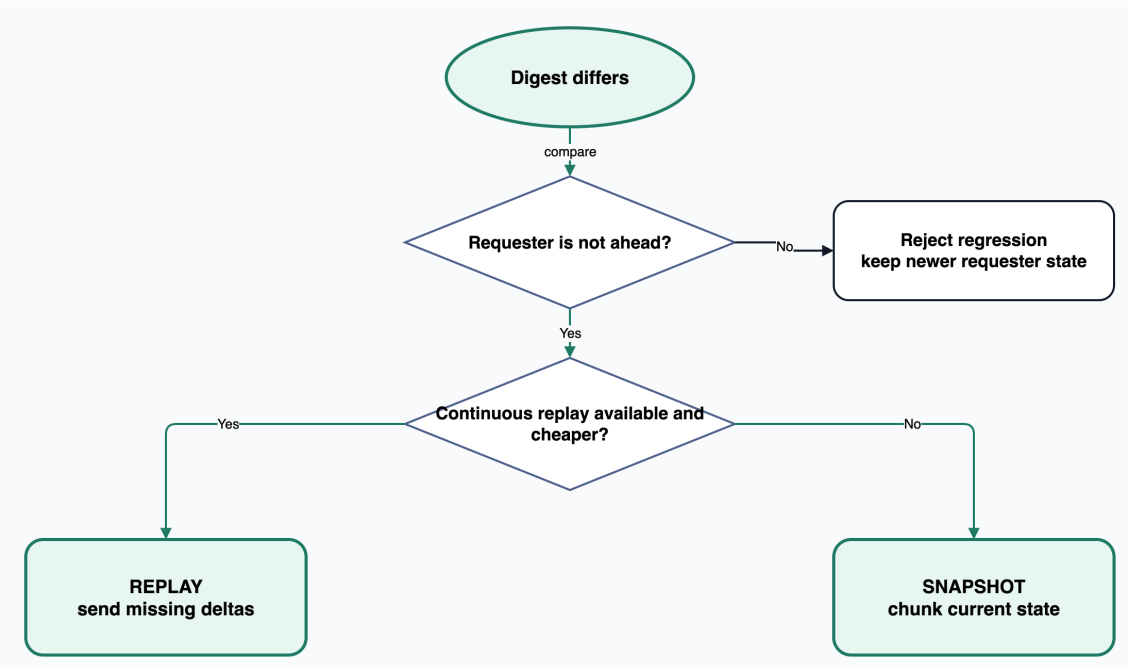

Figure 12: Repair chooses replay, snapshot, or rejects regression

A digest summarizes state so peers can compare observed sequence, entry count, or bucket hash. It carries no complete values and is not automatically authoritative; it supplies input to negotiation.

`AdaptiveRepairPlanner` follows four boundaries:

1. No difference produces `NONE`.
2. A requester that is ahead, or more complete at the same sequence, does not receive an older snapshot.

3. When the requester remains inside the source's replay window, estimated replay and snapshot costs are compared.
4. When replay is unavailable, snapshot restores current state.

Replay and snapshot differ in more than size

Dimension	Replay	Snapshot
Content	retained deltas after a sequence	chunked current collection state
Fit	continuous gap inside retention	gap too old/large or snapshot cheaper
Benefit	incremental and ordered evolution	independent of complete delta history
Risk	retention exhausted; many small messages	large transfer; safe assembly and switch needed
It is not	business audit replay	database backup or cross-resource recovery

Adaptive choice uses latency, throughput, and history in a peer profile. Without a profile, the planner falls back to a fixed strategy, which is more predictable than reacting to one slow request.

Repair cannot move state backward

If requester sequence is 101 while the source has 100, a hash difference does not authorize the source to overwrite the requester. At an equal sequence, a requester with more entries also rejects a smaller source snapshot. `AdaptiveRepairPlannerTest` fixes both cases at `RepairMode.NONE`.

Anti-entropy is a continuing control loop

```
exchange digest → detect difference → plan repair → transfer → apply → compare again
```

Control messages can also be lost. `ChaosIntegrationTest` injects 50% control-plane loss and shows that later sweeps can still detect and repair the difference. Completion of one request only proves one operation; equal final digests establish the repair outcome.

Counterexample and fault injection

- Put requester sequence ahead of the source and observe regression rejection.
- Move the requester outside replay retention and observe snapshot selection.
- Supply a profile that estimates snapshot as faster and observe snapshot even while replay is possible.
- Drop the first control round and observe compensation by a later sweep.

Experiment

```
cd submodule/dsm
mvn -q -pl dsm-sync,dsm-integration-test -am \
  -Dtest=AdaptiveRepairPlannerTest,ChaosIntegrationTest \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

For each requester/source pair in `repair-scenarios.json`, select `NONE`, `REPLAY`, or `SNAPSHOT` and state the reason for avoiding regression.

Experiment acceptance card

Field	Content
Command	The focused Maven tests above; `node --test tests/chapter-assets.test.mjs`
Input or fault	inside/outside replay retention, cost samples, requester ahead, and control loss
Observable result	explainable planner choice; requester never regresses; a later sweep repairs the gap

Field	Content
Evidence level	E2 planner behavior plus E3 chaos repair
This experiment does not prove	Optimal thresholds at production scale, cross-region bandwidth cost, or complete business history

Review

- Digest detects a difference; a repair plan decides how to close it.
- Available replay is not always cheaper, and a snapshot is not automatically authoritative.
- An ahead or more complete requester is protected from regression.
- Anti-entropy repeats; current-state convergence does not replay business history.

Next

Chapter 13 turns to observers: a slow subscriber can overflow the local change buffer even when collection replicas converge.

Evidence links

- `AdaptiveRepairPlanner`
- `AdaptiveRepairPlannerTest`
- `CollectionDigest`
- `ChaosIntegrationTest`

Chapter 13 —ChangeStream Backpressure and Overflow

Chapter objective: After this chapter, a developer can choose ChangeStream capacity, origins, and overflow behavior explicitly, and make drops, blocking, and callback timeouts observable.

Learning objectives

1. Separate collection convergence from ChangeStream delivery.
2. Explain bounded buffering as a stability boundary.
3. Compare `DROP_OLDEST`, `DROP_NEWEST`, `BLOCK`, and `ERROR`.
4. Select origins and failure behavior for a consumer's purpose.

Prerequisites

- The Register change-stream experiment in Chapter 5 is complete.
- Delta and repair from Chapters 11–12 are understood.
- ChangeStream is not treated as a durable business-event log.

Case progress

Metrics Adapter sends `route-hints` changes to a slowing diagnostics dashboard. The collection continues updating while the consumer falls behind. The team decides whether to drop older or newer observations, block briefly, or report an explicit error.

Choosing behavior when the buffer is full

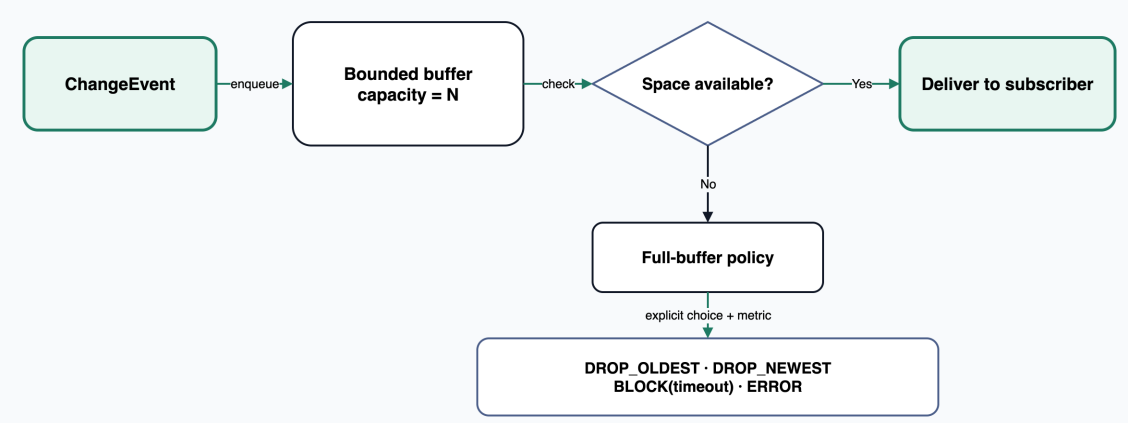

Figure 13: Bounded ChangeStream buffer and overflow policies

`ChangeStreamOptions` defaults to local and remote events, capacity 1024, `DROP_OLDEST`, a one-second block limit, and a five-second subscriber callback limit. The defaults bound resource use; they do not replace a business choice.

Policy	Full-buffer behavior	Suitable use	Main cost
`DROP_OLDEST`	discard oldest, admit newest	UI refresh and current-state hints	incomplete intermediate changes
`DROP_NEWEST`	keep queued events, reject newest	short FIFO work preserving its prefix	increasingly stale current view
`BLOCK`	wait for capacity until timeout	controlled consumer tolerating short jitter	producer waits; timeout can still drop
`ERROR`	raise a structured overflow error	validation/control path forbidding silent loss	caller handles failure

These policies change the observation stream, not the committed collection value. After a stream drops an event, `register.get(key)` can still return the latest value. A current-state consumer rereads the collection instead of assuming event completeness.

Origin filtering is part of the contract

Event origin distinguishes a local operation from a remote delta. Local-only observation can confirm actions on one node; local plus remote helps diagnose convergence. Disabling both has no meaning and the constructor rejects it.

For the running case, the diagnostics dashboard labels both origins. A local publishing aid may watch local events only, while remaining separate from the transaction audit source. Repair validation combines events with a final collection read.

A slow callback cannot hold the publisher indefinitely

A push subscriber callback also has a timeout and reports excess duration through `onError`. A stable design keeps the callback to a fast handoff, places slow IO in a separate bounded executor, and records low-cardinality overflow and callback-timeout metrics.

Counterexample and fault injection

Set capacity to one and publish two changes without consuming. `DROP_OLDEST` retains the second, `DROP_NEWEST` retains the first, `BLOCK` waits and then records overflow on timeout, and `ERROR` raises a structured error. The consumer's need for current view, ordered prefix, or explicit failure determines the policy.

Experiment

```
cd submodule/dsm
mvn -q -pl dsm-runtime -am \
  -Dtest=BufferedChangeStreamTest,InMemoryDsmRegisterTest \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

Complete `overflow-cases.json` with the retained event, producer outcome, and required metric for all four policies.

Experiment acceptance card

Field	Content
Command	The focused Maven tests above; `node --test tests/chapter-assets.test.mjs`
Input or fault	capacity one, two events, slow consumer, and timed-out callback
Observable result	the four policies retain/reject differently; overflow and callback timeout are visible
Evidence level	E2: collection and buffer behavior tests
This experiment does not prove	Durable ChangeStream history, cross-process exactly-once, or sufficient production capacity

Review

- Correct collection state does not imply that every subscriber saw every event.
- Bounded capacity has an explicit overflow policy.
- Blocking is time-bounded behavior, not unlimited reliability.
- After observation loss, the consumer rereads state or uses a durable event system according to its purpose.

Next

Chapter 14 combines the failure windows: during a partition both nodes can return local success, and repair cannot revoke business commitments already made.

Evidence links

- `ChangeStreamOptions`
- `OverflowPolicy`
- `BufferedChangeStreamTest`
- `InMemoryDsmRegisterTest`

Chapter 14 —Convergence Boundaries and Responses Already Returned

Chapter objective: After this chapter, a developer can map local successes during a partition to post-recovery convergence and identify business commitments that eventual state cannot govern.

Learning objectives

1. Separate response semantics, current visible state, and eventual converged state.
2. Explain autonomous versus quorum coordination during a partition.
3. Identify external effects that repair cannot revoke.
4. Draw consistency boundaries for route hints, leases, counters, and order facts.

Prerequisites

- Concurrent conflict and Lease behavior from Chapters 6–7 are understood.
- Online delta, repair, and ChangeStream loss are understood.
- "One final value"is separate from "only one success along the way."

Case progress

A network partition separates `node-a` and `node-b`. Both update the route hint and return OK to their callers. After recovery, the resolver selects one winner. Replicas agree again, while both callers have already acted on their individual successes.

Convergence cannot revoke the past

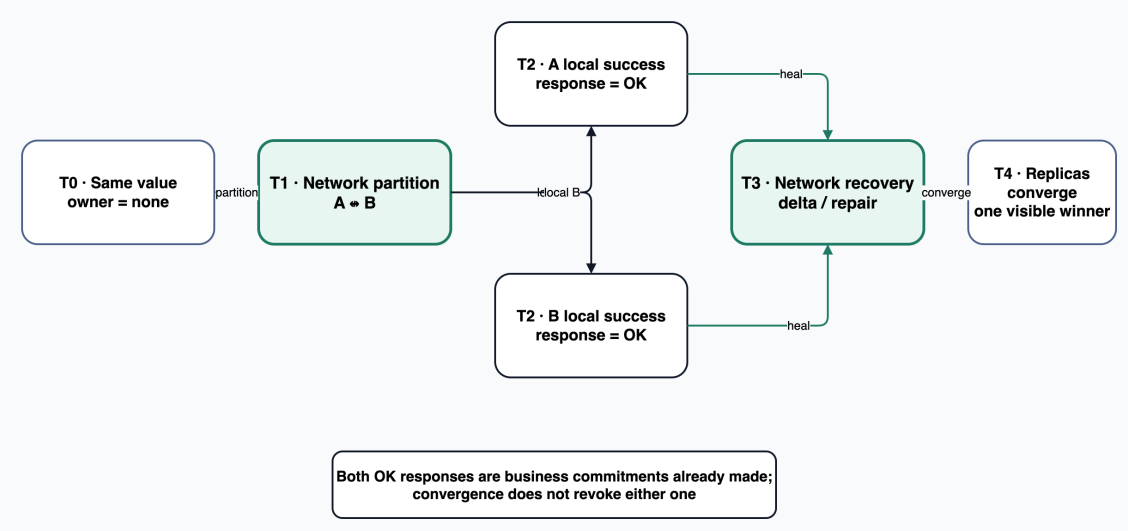

Figure 14: Two local successes during a partition and convergence after recovery

Both T2 responses happened. Delta/repair at T3 can converge the current T4 value, but it cannot make a client forget an OK or automatically compensate email, payment, shipment, or an external write.

Three separate contracts clarify the responsibilities:

Contract	Question	DSM alone can guarantee
local operation	when does `put/acquire/increment` return?	local collection outcome

Contract	Question	DSM alone can guarantee
convergence	how do replicas reach compatible state after recovery?	semantics specific to Register/Lease/CRDT
business commitment	can a returned success be retried, revoked, or reconciled?	requires transaction, idempotency, fencing, or compensation

Collection behavior during a partition

Register: two OK responses and one eventual winner

Both nodes can accept candidates and later select one current value by metadata and resolver. This fits a route hint because the losing candidate can be superseded and an authoritative health source can publish again. It does not make one of two accepted order updates disappear from history.

Lease: explicit availability/exclusivity trade-off

AUTONOMOUS mode can produce holders on both sides and relies on an external fencing token to reject the old owner. QUORUM mode rejects acquire while membership is unstable or a majority is absent, exchanging availability for a stronger issuance constraint. Both modes depend on downstream token validation; an in-memory holder decision cannot restrain a disconnected old process.

CRDT: safe merge does not imply legal business state

Request counts can increase locally and merge after recovery. That property fits metrics. It does not prove that inventory decrements respect a lower bound or preserve an audit trail.

Rejection list

The following state does not become a suitable DSM authority merely because it eventually converges:

- order transitions and irreversible approvals;
- inventory deductions and quota consumption;
- payment, refund, and ledger entries;
- business events requiring complete history;
- cross-resource transactions requiring one definite caller outcome.

DSM can cache or coordinate rebuildable state around these systems. Transactional facts, idempotency protocols, and audit systems retain authority.

Counterexample and fault injection

`ChaosIntegrationTest` contrasts two Lease choices: autonomous mode can create dual holders during a partition, while quorum mode rejects acquire without a majority and resumes only after recovery and restabilization. The comparison asks the business owner whether paused work or temporary duplicate execution is less harmful and whether fencing/idempotency protects the side effect.

Experiment

```
cd submodule/dsm
mvn -q -pl dsm-integration-test -am \
  -Dtest=ChaosIntegrationTest \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

Review `partition-contract.json`. Every state records `localResponse`, `partitionBehavior`, `healedState`, and `externalGuard`. “Eventual consistency handles it”is not a complete field value.

Experiment acceptance card

Field	Content
Command	The Maven chaos test above; `node --test tests/chapter-assets.test.mjs`
Input or fault	network partition, operations on both sides, recovery, and anti-entropy
Observable result	Register/repair converges; autonomous Lease can have dual holders; quorum Lease fails closed
Evidence level	E3: multi-node chaos behavior test
This experiment does not prove	Compensated business effects, cross-resource transactions, or suitable production network parameters

Review

- One eventual winner does not revoke two OK responses returned during a partition.
- Collection semantics govern current coordination state, not the whole business-commitment protocol.
- Lease mode makes the availability/issuance trade-off explicit.
- Orders, inventory, payments, and event history remain in their authoritative systems.

Next

Arc 4 carries these boundaries into application engineering. Chapter 15 introduces domain ports so business code publishes routes, claims shards, and records requests without scattering DSM APIs.

Evidence links

- `ChaosIntegrationTest`
- `ConflictResolver`
- `LeaseMode`
- Running-case design

Arc 3 Recap —Observable Replication, Partition, and Repair (Chapters 10–14)

Arc 3 expands "consistent after network recovery"into an observable process: where loss occurs, who detects it, how it is repaired, and why returned business responses still need external protection.

The path from membership to convergence

```
membership observation stabilizes
  → local mutation creates delta
  → online propagation attempts remote application
  → digest/anti-entropy detects a gap
  → replay or snapshot repairs it
  → another comparison confirms convergence
```

ChangeStream offers one observation surface, while its own bounded buffer can lose events. Final judgement returns to collection state, digest, and the business system of record.

Five boundaries

Boundary	Common misconception	Working contract
membership	one join/leave event is a global fact	wait for consecutive stable views before high-risk quorum decisions
propagation	local success means remote write complete	record local commit and remote application separately
repair	more replay/snapshot is always safer	avoid regression; choose from retention and cost
subscription	ChangeStream is a durable event log	bounded buffer, explicit overflow, reread state when needed
response	one eventual winner revokes the loser's OK	transaction, idempotency, fencing, or compensation protects effects

How this arc realizes DSM's value

Runtime reuses membership handling, online delta, difference detection, and replica repair across services. Business teams avoid rebuilding those pipelines for each service. DSM restores current coordination state; callers still interpret local success, protect external effects, and choose acceptable ChangeStream loss/failure behavior.

Fault-to-evidence map

Fault	Minimal evidence	Still unproven
membership churn	stable-round reset; Lease returns `membership-unstable`	real-network threshold suitability
missed online delta	local success first; repair catches up after recovery	exactly-once history delivery
lagging requester	planner chooses replay/snapshot without regression	optimal large-scale cost
slow subscriber	four overflow outcomes and metrics	lossless business-event flow
network partition	autonomous/quorum Lease contrast	protection of external effects

Six review questions

1. Does the decision use visible membership, original cluster size, or stable size?

2. When local success returns, which remote stage has completed?
3. Who detects a missed online delta?
4. How does snapshot take over safely when replay is unavailable or costlier?
5. When a subscription buffer fills, does it drop, wait, or fail, and which metric records it?
6. Did local success during a partition trigger an irreversible effect?

Migration exercise

Replace the route hint with a feature-rollout flag. Record its authority, locator, impact of two local OK responses during a partition, repair mode, ChangeStream overflow policy, and categories of flags excluded from DSM.

Next

Arc 4 wraps collections and failure boundaries in business ports, integrates Spring Boot, and protects engineering boundaries with two-layer identity isolation and layered test evidence.

LEARNING ARC

Integrate Application Engineering

Chapters 15–18

Contain DSM in domain ports, Spring assembly, isolation rules and layered tests.

Chapter 15 —Isolate DSM Details with a Domain Adapter

Chapter objective: After this chapter, a developer can wrap the three typed DSM handles in a domain port and keep locator, lifecycle, and failure translation inside the adapter boundary.

Learning objectives

1. Identify the maintenance cost of DSM APIs leaking into business code.
2. Design a `FulfillmentCoordination` port in business language.
3. Centralize Register, Lease, CRDT handles and locators in a composition root.
4. Validate domain decisions and real Runtime behavior separately.

Prerequisites

- Collection selection from Chapter 9 is complete.
- Failure and business-commitment boundaries from Chapter 14 are understood.
- The Maven example in `examples/order-fulfillment-control-plane` can run.

Case progress

Direct collection calls in the first three arcs exposed semantics. In an application, the fulfillment use cases say “publish a route,”“claim a shard,”and “record a request.”They do not refer to `shared/gateway/route-hints` or `LeaseAcquireOptions`.

The domain port owns language; the adapter owns mechanism

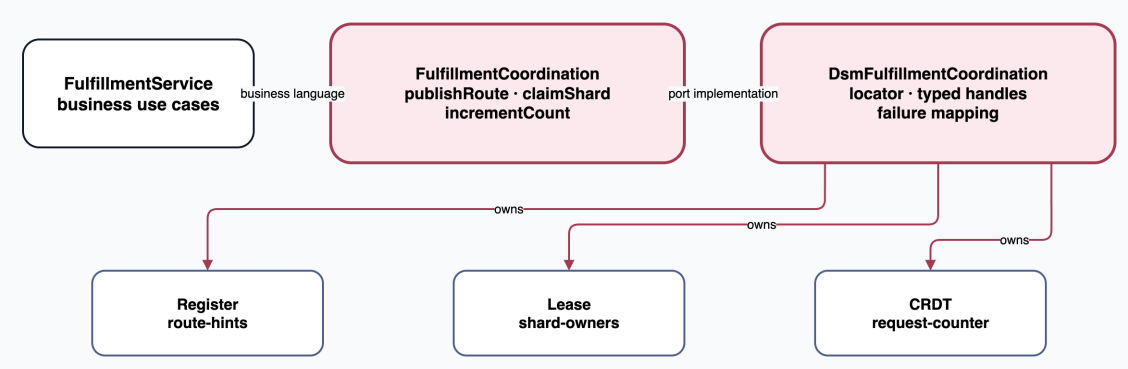

Figure 15: Domain port and DSM adapter boundary

The book example defines a small port:

```
public interface FulfillmentCoordination {
    void publishRoute(String serviceKey, String address);
    Optional<String> routeFor(String serviceKey);
    ShardClaim claimShard(String shardId);
    long incrementRequestCount();
}
```

The port returns no `DsmRegister`, `LeaseSnapshot`, `PnCounterState`, or `CollectionLocator`. A fake can test the business service directly. The DSM adapter translates the Lease result into `ShardClaim(granted, uncertain, fencingToken, reason)`.

One composition root

`DsmFulfillmentModule` owns:

- creation, start, and close of `DsmRuntime`;
- `clusterId` and `serviceId`;
- three stable locators and schema IDs;
- entity, codec, and spec definitions for Register, Lease, and CRDT;
- mapping of three typed handles into the domain port.

Locator and schema remain important contracts. The adapter owns them so they do not spread through controllers, services, and jobs.

Failure translation preserves uncertainty

A Lease acquire result contains `granted`, `uncertain`, a snapshot, and a reason. Reducing it to a boolean prevents the caller from distinguishing explicit rejection from an uncertain result. The domain result retains information required by a business decision while preventing arbitrary handle access.

The fencing token also crosses the port into a protected downstream write. Hiding it would break the Chapter 7 boundary. The example returns the token but does not implement a real downstream guard, so the acceptance card does not claim external fencing.

Two test classes prove two layers

1. `RecordingCoordination` shows that `FulfillmentService` depends only on the domain port: E1.
2. A real `DsmRuntime` and three collection types validate the adapter in one process: E2.

Only the second makes business decisions and DSM assembly inseparable; only the first can stay green while a real spec, codec, or Lease fails. Both are needed.

Counterexample and fault injection

- Copy locator strings into `FulfillmentService`, change a collection ID, and count business files touched.
- Reduce `LeaseAcquireResult` to a boolean and try to express `uncertain=true`.
- Omit `runtime.start()` and locate the lifecycle failure.
- Let business code call `requestCounter.state()` and observe CRDT types in domain tests.

Experiment

```
cd examples/order-fulfillment-control-plane
mvn clean test
```

Inspect `port-contract.json`. Each business action identifies domain input/output, DSM handle, and external responsibility outside the adapter.

Experiment acceptance card

Field	Content
Command	`cd examples/order-fulfillment-control-plane && mvn clean test`
Input or fault	fake port, real single-node Runtime, and three typed handles
Observable result	domain service imports no DSM types; adapter performs route, Lease, and counter operations
Evidence level	E1 domain unit test plus E2 single-process Runtime behavior
This experiment does not prove	Multi-node propagation, repair, real network behavior, downstream fencing, or production assembly

Review

- Domain ports own business language; adapters own DSM mechanism.
- Locator, schema, codec, and lifecycle belong in the composition root.

- Failure translation retains uncertainty and the fencing token.
- Fakes and real Runtime tests answer different questions.

Next

Chapter 16 replaces manual assembly with Spring Boot properties and conditional beans, then validates context startup and lifecycle.

Evidence links

- DSM examples: Runtime assembly
- DSM examples: Spring Boot assembly
- DSM examples: complete Basic Usage example
- Current integration guide

Chapter 16 —Assemble Runtime with Spring Boot

Chapter objective: After this chapter, a developer can create Runtime and named collection beans from `dsm.*` properties, understand auto-configuration gates, and validate context and lifecycle behavior.

Learning objectives

1. Explain the assembly order of `DsmProperties`, `DsmAutoConfiguration`, and collection beans.
2. Configure `clusterId`, `serviceId`, membership, and three collection types.
3. Inject a typed handle or domain adapter by bean name.
4. Fail fast on invalid configuration during startup.

Prerequisites

- The domain port and composition root from Chapter 15 are complete.
- Spring Boot configuration properties and bean lifecycle are familiar.
- Successful context startup is distinct from validated multi-node networking.

Case progress

The fulfillment application replaces manual builders with `application.yml`. Auto-configuration creates membership and Runtime, registers named collection beans, then injects them into the DSM domain adapter.

From configuration to domain port

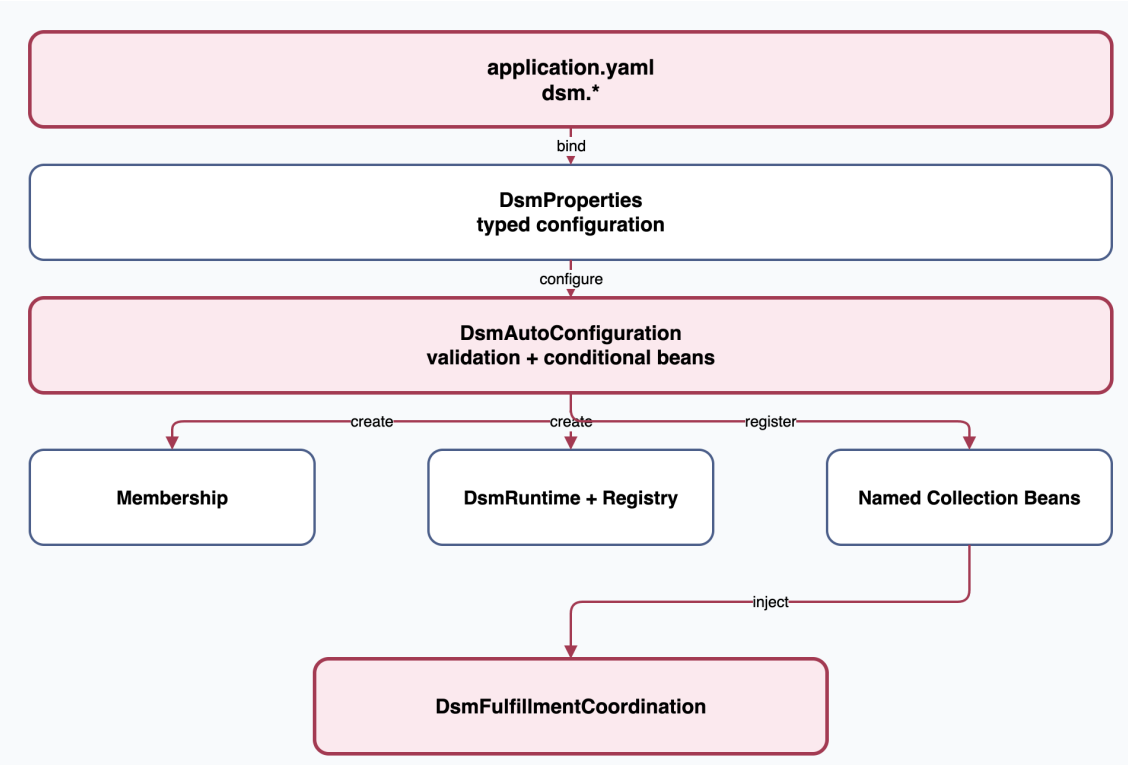

Figure 16: Spring Boot assembly from properties to domain adapter

The order is structural: collection registration needs Runtime; the domain adapter needs registered handles; shutdown lifecycle stops Runtime.

Minimal property contract

The Chapter 16 asset contains complete configuration for three collection types. Key fields include:

```
dsm:
  cluster-id: fulfillment
  service-id: fulfillment-control-plane
  cluster:
    mode: standalone
  collections:
    - bean-name: routeHintsCollection
      tenant-id: shared
      application-id: gateway
      collection-id: route-hints
      schema-id: route-hints/v1
      type: REGISTER
      consistency-tier: REGISTER
      entity-type: com.example.RouteHint
```

Referenced classes and codec/merger/entity factories need to exist. The full teaching asset uses placeholder `com.example.*` types for configuration review; it is not presented as a startable application. Startup evidence comes from pinned `DsmAutoConfigurationTest` and `SpringBootDsmApplicationTest` sources.

Two forms of bean name

Auto-configuration creates a canonical locator-based name:

`dsmCollection:shared/gateway/route-hints`

Configured `bean-name` supplies a business-friendly alias such as `routeHintsCollection`. Both resolve to the same registration. The adapter uses an explicit qualifier instead of asking the container to infer a target from erased generic types.

Record codec convenience and boundary

When `entity-type` is a record, auto-configuration can derive a `RecordCodec`. A non-record without `codec-bean` fails startup. CRDT also requires state codec, initial state, and merger; Lease requires an entity factory.

Fail-fast behavior locates an assembly error during startup instead of waiting for the first remote message.

Startup gates

`DsmAutoConfigurationTest` fixes these outcomes:

Configuration	Outcome
missing `cluster-id`	startup fails
duplicate locator	startup fails
non-record without codec	startup fails
CRDT without merger	startup fails
invalid Lease parameter	startup fails
custom `ClusterMembership` bean	default implementation backs off
context closes	Runtime lifecycle stops

Counterexample and fault injection

Copy the configuration asset, remove the CRDT merger bean, or assign one locator to two collections. The context should reject startup at the configuration boundary instead of delaying failure until a business action.

Experiment

```
cd submodule/dsm
mvn -q -pl dsm-spring-boot-autoconfigure -am \
```

```
  -Dtest=DsmAutoConfigurationTest,DsmSpringBootSmokeTest \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

Run the real example application context:

```
cd submodule/dsm-examples/basic-usage-examples
./mvnw -q -Dtest=SpringBootDsmApplicationTest test
```

Experiment acceptance card

Field	Content
Command	The focused auto-configuration and Basic Usage tests above
Input or fault	three collection configurations, missing codec/merger, duplicate locator, and context close
Observable result	Runtime starts; canonical and alias beans resolve; bad configuration fails fast; close stops Runtime
Evidence level	E2: real Spring ApplicationContext and single-process Runtime
This experiment does not prove	Multi-instance discovery, real-network replication, production secrets, or deployment readiness

Review

- Spring Boot changes assembly, not collection semantics or failure boundaries.
- Runtime precedes collections; collections precede the adapter.
- Canonical locator names and business aliases remain explicit, stable, and testable.
- Invalid configuration fails at startup rather than degrading silently at runtime.

Next

Chapter 17 separates two identity layers: `clusterId/serviceId` controls the communication domain and locator controls the collection domain.

Evidence links

- `DsmProperties`
- `DsmAutoConfiguration`
- `DsmAutoConfigurationTest`
- `SpringBootDsmApplicationTest`

Chapter 17 —Isolate Communication and Collection Domains

Chapter objective: After this chapter, a developer can configure communication and collection domains separately and use equal entry keys to prove that the two identity layers remain distinct.

Learning objectives

1. Distinguish `clusterId`, `serviceId`, and collection locator.
2. Explain validation of cluster, service, and collection identity on a message.
3. Prove that equal entry keys under different locators identify different state.
4. Avoid treating a Spring bean name as wire identity.

Prerequisites

- Locator from Chapter 3 and `serviceId` from Chapter 10 are understood.
- Spring bean registration from Chapter 16 is complete.
- Container identifiers and distributed-protocol identifiers are distinct.

Case progress

The gateway and worker both use entry key `fulfillment-primary` in one Runtime, under `route-hints` and `shard-owners`. A test environment can also use the same locator while a different cluster/service identity keeps its messages separate from production.

Two boundaries, not interchangeable names

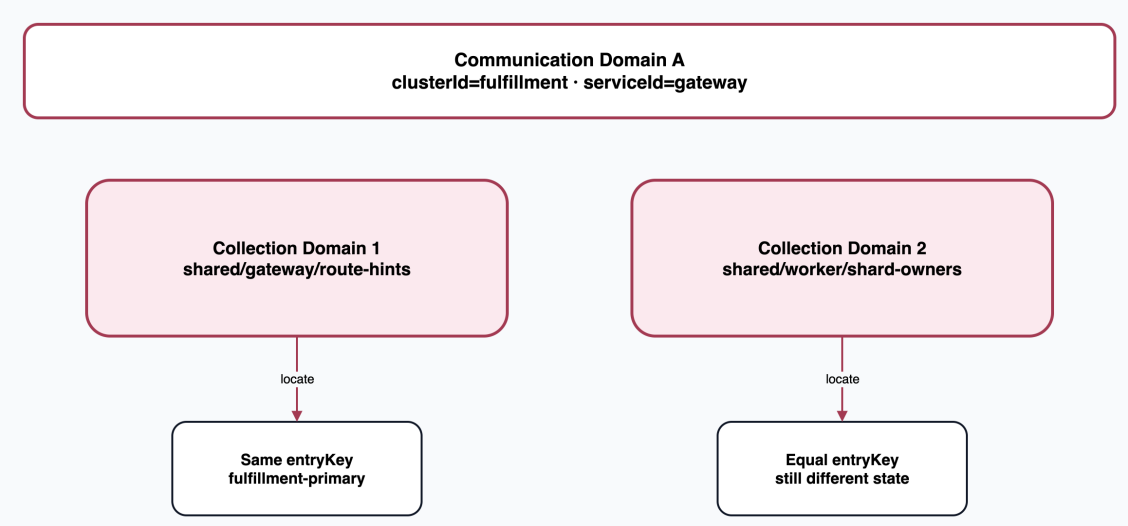

Figure 17: Two-layer isolation of communication and collection domains

Identifier	Scope	Purpose
`clusterId`	Runtime/sync envelope	membership in one deployment cluster
`serviceId`	membership and sync envelope	instances that discover and exchange DSM messages
`tenantId`	collection locator	state namespace or tenant ownership
`applicationId`	collection locator	application subdomain owning the collection

Identifier	Scope	Purpose
`collectionId`	collection locator	stable identity of one collection
Spring `bean-name`	one application context	code selecting a registered handle
`entryKey`	inside one collection	entity inside that collection

An `entryKey` gains meaning with its complete locator. Equal keys under different collection or application IDs do not identify one business object.

A message crosses two identity checks

Membership rejects discovery/messages from a different `serviceId`; `RuntimePlatformSyncService` also validates `clusterId` and `serviceId` in the envelope. After the communication domain accepts it, collection routing resolves the locator and handle.

```
network reachability
  → clusterId/serviceId match
  → collection descriptor/locator match
  → schema/capability match
  → apply entryKey inside target collection
```

Every rejection layer has an observable reason. Missing data cannot be attributed to locator without checking the earlier gates.

The equal-key counterexample in `RuntimeIntegrationTest`

One Runtime registers `shared/gateway/route-hints` and `shared/worker/route-hints`. Both Registers receive `same-key`; one stores `gateway-value`, the other `worker-value`. Each size remains one and each returns its own value. Locator, rather than Java entity type or entry key alone, owns collection identity.

Renaming distributed identity is protocol migration

Changing a bean alias affects one Spring context. Changing locator, cluster ID, or service ID changes distributed communication or state identity. It needs dual-write/migration, a compatibility window, or an explicit cold-start policy rather than an ordinary code rename.

Counterexample and fault injection

- Configure two collections with one locator and expect startup rejection.
- Give two Runtimes different service IDs and expect separate member views.
- Write the same key under different application IDs and expect isolated values.
- Change only the bean alias and confirm locator/wire identity remain unchanged.

Experiment

```
cd submodule/dsm
mvn -q -pl dsm-integration-test,dsm-cluster -am \
  -Dtest=RuntimeIntegrationTest,MulticastClusterMembershipTest \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

Inspect `isolation-cases.json`; each scenario identifies a change to communication domain, collection domain, or application-context name.

Experiment acceptance card

Field	Content
Command	The focused Runtime/membership tests above plus the book asset test
Input or fault	equal key, different locator, different service ID, and duplicate locator

Field	Content
Observable result	locator isolates values; service ID isolates membership; duplicate locator fails fast
Evidence level	E2 single-Runtime isolation plus E3 controlled multi-node membership
This experiment does not prove	Operating-system/network isolation, authorization, or tenant-data compliance

Review

- `clusterId/serviceId` controls communication; locator controls collection state.
- A bean name serves dependency injection and is not wire identity.
- Equal entity types and entry keys do not merge across locators.
- Distributed-identity changes need a migration plan.

Next

Chapter 18 arranges the preceding tests into an evidence ladder and states what each layer proves and leaves open.

Evidence links

- `DsmRuntimeBuilder`
- `RuntimePlatformSyncService`
- `MulticastClusterMembership`
- `RuntimeIntegrationTest`

Chapter 18 —Scope of Layered Test Evidence

Chapter objective: After this chapter, a developer can select evidence for a domain port, real Runtime, multi-node protocol, or network black box and refuse to substitute a lower-layer green result for higher-layer acceptance.

Learning objectives

1. Build an E1–E4 evidence ladder.
2. Record environment, pinned revision, proven claims, and open claims at each layer.
3. Distinguish real collections, controlled fake membership, and real network ports.
4. Define a minimal faithful validation set for all three collection types.

Prerequisites

- The Chapter 15 example and Chapters 16–17 assembly/isolation are complete.
- The Chapter 14 chaos test has run.
- Test count is not treated as a production-readiness score.

Case progress

The fulfillment team has domain, single-Runtime, and chaos tests. Release review asks whether these results prove real ports, client protocols, and process isolation. The answer is stated one evidence layer at a time.

Evidence strength describes scope, not points

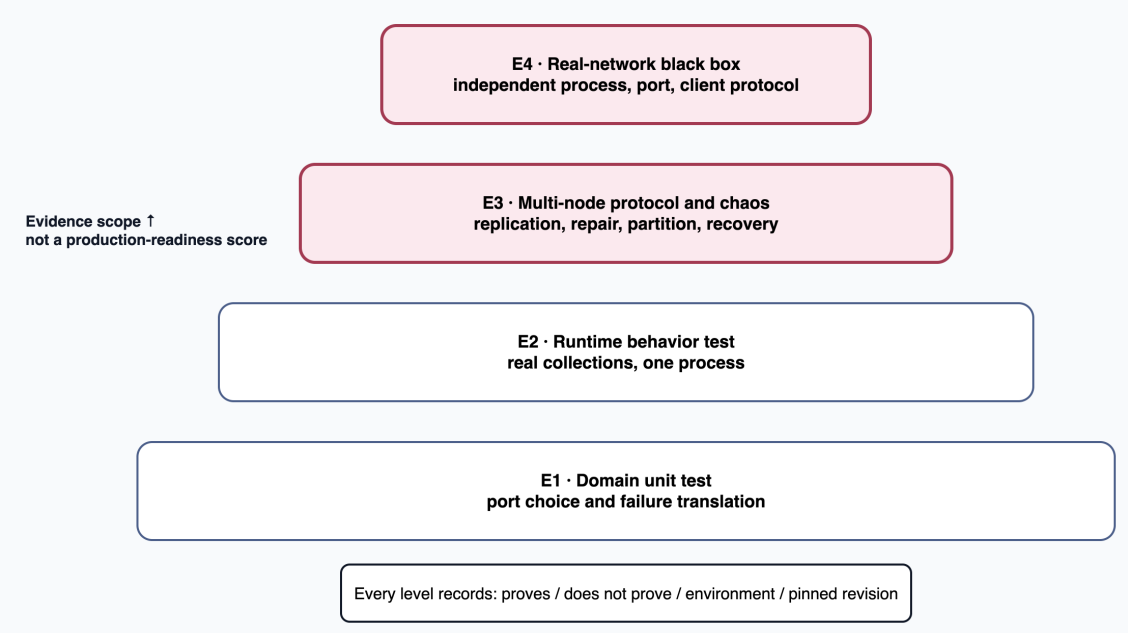

Figure 18: Evidence ladder from domain contract to network black box

Level	Execution surface	Best evidence for	Cannot replace
E1	domain fake/static asset	port language, decision table, failure translation	real collection/protocol behavior
E2	real Runtime, one process	spec, codec, typed handle, lifecycle	multi-node propagation and partition
E3	multi-node protocol/controlled chaos	delta, repair, partition, recovery, member events	independent processes and real ports

Level	Execution surface	Best evidence for	Cannot replace
E4	independent processes, real port, client protocol	process boundary, protocol compatibility, reachability	production capacity and full failure domains

E4 is closer to deployment reality than E3 and still leaves topology, capacity, access, alerts, and operations to production acceptance.

Minimal evidence by collection

Register

- E1: the route hint is rebuildable guidance, not an order fact.
- E2: put/get/remove, resolver, and ChangeStream overflow.
- E3: two-node delta, tombstone, schema mismatch, and repair.
- E4: Chapter 23's Redis-shaped lab observes replication through a real TCP port.

Lease

- E1: the domain result retains uncertain, reason, and fencing token.
- E2: a guard rejects the old token after acquire/transfer.
- E3: autonomous dual holder and quorum fail-closed behavior.
- E4: the actual protected resource validates the token; this book defines the boundary without claiming a ready-made business system.

CRDT

- E1: counts support operations, not billing authority.
- E2: single-Runtime merge and algebra cases.
- E3: multi-node local updates, duplicate/reordered delivery, and repair convergence.
- E4: exposure through a client protocol depends on the application and is not proven by a CRDT unit test.

Every evidence card fixes four facts

1. **Environment and revision:** source, JDK, and Maven context.
2. **Input/fault:** the actual injected condition.
3. **Observation:** test counts, failures, errors, skips, and key behavior.
4. **Open claim:** the question left to the next layer.

`Skipped` and `Tests run: 0` are not passes; compilation alone is not behavior evidence.

Counterexample and fault injection

- Select a nonexistent Maven test while disabling `failIfNoSpecifiedTests` and observe a false green with zero tests.
- Use an E1 fake result to claim network repair and name the missing execution surface.
- Use controlled E3 multi-node results to claim real TCP compatibility and identify the process/port gap.
- Record "all passed" without a revision and observe the loss of reproducibility.

Experiment

Run book-owned E1/E2:

```
cd examples/order-fulfillment-control-plane
mvn clean test
```

Run pinned-source E2/E3:

```
cd submodule/dsm
mvn -q -pl dsm-integration-test -am \
  -Dtest=RuntimeIntegrationTest,TwoNodeIntegrationTest,ChaosIntegrationTest \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

Use `evidence-ladder.json` to confirm that every level contains nonempty `proves` and `doesNotProve` fields.

Experiment acceptance card

Field	Content
Command	Two book-example tests plus focused Runtime/TwoNode/Chaos tests
Input or fault	fake port, real single Runtime, two nodes, loss, and partition
Observable result	each layer's green result and count are reproducible; no zero-test or skipped false green
Evidence level	E1, E2, E3; Chapter 23 provides the real-port E4 lab
This experiment does not prove	Production capacity, cross-host deployment, real external fencing, or complete security configuration

Review

- Test level describes execution scope rather than a quality score.
- Every layer states both proven and open claims.
- A fake, one Runtime, multi-node chaos, and a real port do not replace one another.
- Revision and fault input precede any green conclusion.

Next

Arc 5 turns to runtime responsibility. Chapter 19 adds diagnostics, metrics, and trace surfaces for locating distributed differences.

Evidence links

- DSM examples: Spring Boot application test
- `RuntimeIntegrationTest`
- `TwoNodeIntegrationTest`
- `ChaosIntegrationTest`

Arc 4 Recap —From Collection APIs to Application Engineering (Chapters 15–18)

The application layer now depends on a fulfillment coordination port. DSM locators, typed handles, Spring assembly, and test evidence each have a clear owner.

Engineering boundary

```
FulfillmentService
  → FulfillmentCoordination (business port)
    → DSM adapter (failure translation)
      → composition root / Spring auto-configuration
        → Runtime + named typed handles
```

A fake supports fast business-decision tests while the adapter still runs against a real Runtime. Spring Boot is another composition root and does not alter Register, Lease, or CRDT semantics.

Four distinctions

Distinction	Left responsibility	Right responsibility
domain port / DSM handle	business action and result	collection operation and technical failure
bean name / locator	select a bean in one container	distributed collection identity
clusterId/serviceId / locator	communication and sync domain	collection and state domain
test pass / production readiness	evidence inside a fixed surface	deployment, capacity, security, and operations acceptance

How this arc realizes DSM's value

The domain port contains collection APIs, the composition root centralizes locator/codec/lifecycle, Spring Boot provides testable assembly, and the evidence ladder limits every conclusion. This structure reduces integration and evolution cost while keeping DSM types, identity, and failure semantics out of business services.

Transfer checklist

1. Does a business service import `com.leanowtech.dsm.*` outside a composition root or adapter?
2. Does every locator have one source of truth and a schema ID?
3. Does a Lease result retain uncertain, reason, and fencing token?
4. Does Spring fail fast on duplicate locator, missing codec/merger, and invalid Lease settings?
5. Is a cluster/service identity change handled as protocol migration?
6. Does each test layer record `proves` and `doesNotProve`?

Evidence in this arc

- The book example provides an E1 domain-port test and E2 real-Runtime test.
- Spring auto-configuration fixes Runtime, three collection beans, configuration failures, and lifecycle behavior.
- Runtime integration fixes locator isolation, CRDT behavior, and Lease fencing.
- TwoNode/Chaos fixes E3 replication, repair, and partition behavior.

Next

Arc 5 completes the production operating contract: observability, security, lifecycle, and capacity measurement.

LEARNING ARC

Own Runtime Operations

Chapters 19–22

Turn diagnostics, security, lifecycle and capacity validation into inspectable operating contracts.

Chapter 19 —Locate Replica Differences with Observable Signals

Chapter objective: After this chapter, a developer can combine diagnostics, low-cardinality metrics, and traces into falsifiable troubleshooting hypotheses instead of watching one “sync failed”counter.

Learning objectives

1. Read identity, state, member view, and collection summaries from `RuntimeDiagnostics`.
2. Use outcome/reason metrics to separate membership, propagation, repair, Lease, and consumer faults.
3. Correlate one cross-node propagation with trace context.
4. Avoid high-cardinality labels and unbounded diagnostic snapshots.

Prerequisites

- Repair and backpressure from Chapters 12–13 are complete.
- The evidence ladder from Chapter 18 is understood.
- Diagnostic data is an observation, not a new source of business truth.

Case progress

`node-b` has an older route hint than `node-a`. Before restarting anything, the team checks Runtime state, member visibility, locator registration, message-drop/repair outcomes, and a trace for that propagation.

Three observation surfaces form one diagnostic loop

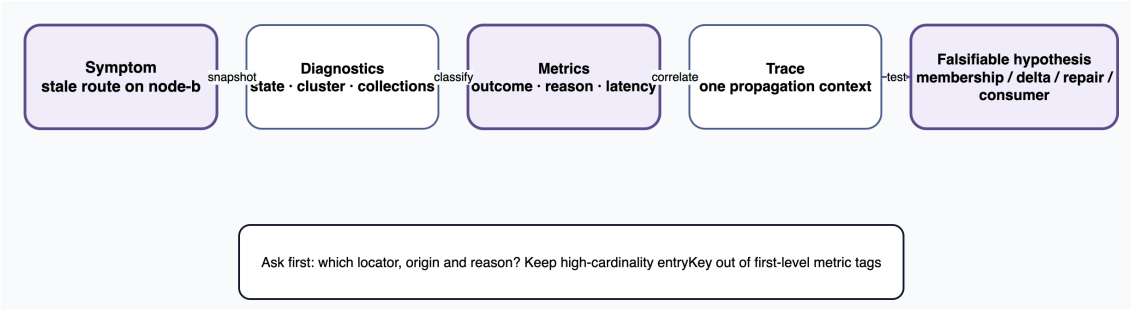

Figure 19: Diagnostic loop from symptom to falsifiable hypothesis

Diagnostics: current state

`runtime.diagnostics()` returns an immutable snapshot with `RuntimeInfo`, lifecycle state, cluster view, total entries, per-collection diagnostics, Lease-specific diagnostics, and last error. It describes what this node sees now, not a time-series history.

Metrics: accumulating outcomes

`DsmMetrics` records sync latency, cluster size, authentication failure, message drop, repair outcome, Lease operations, fencing rejection, and ChangeStream overflow. Stable low-cardinality reason/outcome labels support aggregation and alerts.

Trace: path of one action

`TraceContextHolder` puts current trace context into outbound platform envelopes. Two-node integration verifies propagation on a Register delta. A trace correlates one path; it does not replace metric trends or a diagnostic snapshot.

An executable troubleshooting path

Step	Question	Primary signal
1	Is Runtime ready?	`state`, `isReady`, `lastErrorMessage`
2	Is the node in the right communication domain?	cluster view, service/cluster mismatch metric
3	Is the locator registered with a compatible schema?	collection diagnostics, message-drop reason
4	Did online delta arrive?	sync latency, `REMOTE_DELTA`, trace
5	Did repair detect and handle the gap?	digest/repair outcome, replay/snapshot reason
6	Is only the observer behind?	ChangeStream overflow/callback timeout

Each step produces a falsifiable next hypothesis. Normal membership, rising delta drops, and later repair success support online loss rather than a bad locator.

High cardinality is another failure mode

Arbitrary `entryKey`, full exception text, or trace ID in a metric tag creates time series proportional to business data. `DsmMetrics` requires stable reason/outcome labels. Specific keys and exception context belong in structured logs, audit, or traces.

Counterexample and fault injection

- Overflow a ChangeStream and confirm current entries remain visible in diagnostics while the metric rises.
- Inject 50% control-plane loss and observe a later repair outcome.
- Put a Register under trace context and confirm the outbound envelope carries it.
- Add a random entry key as a tag in the exercise asset and observe cardinality review rejection.

Experiment

```
cd submodule/dsm
mvn -q -pl dsm-runtime,dsm-sync,dsm-metrics-micrometer,dsm-integration-test -am \
 -Dtest=DefaultDsmRuntimeTest,MicrometerDsmMetricsTest,TraceContextHolderTest,RuntimePlatformSyncServiceTest
  -Dsurefire.failIfNoSpecifiedTests=false test
```

Use `diagnostic-runbook.json` to record a snapshot, metric, trace, and next hypothesis for four symptoms.

Experiment acceptance card

Field	Content
Command	The focused Runtime/metrics/trace/chaos tests above plus book assets
Input or fault	stale replica, message loss, repair, overflow, and trace context
Observable result	current state, outcome trend, and single path corroborate without replacing one another
Evidence level	E2 observation implementations plus E3 controlled trace/chaos
This experiment does not prove	Metrics-backend capacity, alert thresholds, production sampling, or on-call effectiveness

Review

- Diagnostics answers current state, metrics shows trends, and trace follows one path.
- Troubleshooting begins with identity, locator, and reason instead of restart.
- Metrics use low-cardinality outcomes/reasons; keys stay in logs and traces.
- Agreement among observation surfaces does not make a business fact correct.

Next

Chapter 20 routes every remote message through admission, authentication, decryption, replay prevention, and abuse control before sync/collection routing.

Evidence links

- `RuntimeDiagnostics`
- `DsmMetrics`
- `MicrometerDsmMetrics`
- `RuntimePlatformSyncServiceTest`

Chapter 20 —Message Security and Convergence Preconditions

Chapter objective: After this chapter, a developer can separate cluster admission, authenticity, confidentiality, replay prevention, abuse control, and audit, and explain the compatibility window during key rotation.

Learning objectives

1. Define ordered security gates for a remote envelope.
2. Distinguish HMAC authentication, payload encryption, and nonce replay prevention.
3. Explain key version, minimum acceptable version, and rotation window.
4. Map every rejection to a structured error, metric, and audit event.

Prerequisites

- Wire envelopes from Chapter 11 and observation surfaces from Chapter 19 are understood.
- TLS/network isolation does not automatically replace message identity and replay protection.
- Security misconfiguration fails closed.

Case progress

The fulfillment cluster begins receiving cross-node deltas. It needs to reject nodes with a wrong token, detect payload tampering, expire old signing keys, stop replay of a valid message, and limit senders that consume validation resources through repeated failures.

Five gates answer five questions

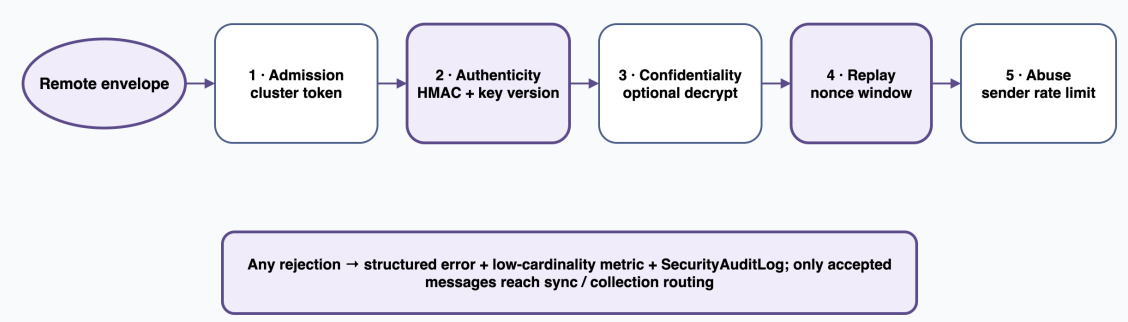

Figure 20: Security gates before a remote message reaches a collection

Gate	Question	Typical component
Admission	Does the node hold the cluster credential?	`SharedKeyClusterTokenValidator`
Authenticity	Was the message signed by a trusted key and left intact?	`HmacMessageAuthenticator`
Confidentiality	Was required payload encryption applied and decrypted?	AES-GCM / ChaCha20 encryptor
Replay	Is the nonce fresh and unseen?	`SlidingWindowNonceTracker`
Abuse	Should a repeatedly failing sender be blocked temporarily?	`SenderRateLimiter`

Encryption does not authenticate a sender, and authentication does not stop replay of a valid packet. One “security enabled” flag hides configuration and diagnosis details.

Key rotation is a window

The security integration test validates rolling migration: old and new signatures coexist temporarily. Receivers choose the verification key by version. After every sender moves, raising `min-acceptable-key-version` rejects old envelopes.

```
phase A: active=1, accept >=1
phase B: publish key 2, send active=2, receive accept >=1
phase C: after all nodes switch, accept >=2
```

Keeping only the new key immediately disconnects nodes not yet rolled. Never raising the minimum leaves exposure from an old key indefinitely.

Nonce window and memory bound

`SlidingWindowNonceTracker` accepts an increasing nonce or an unseen nonce inside the window, and rejects duplicates, stale values, and zero. `maxTrackedSenders` plus LRU eviction prevents unknown senders from consuming unbounded memory. Too small a limit repeatedly forgets senders; too large a limit expands memory exposure.

Audit is not a debug log

`SecurityAuditLog` records token mismatch, signature failure, replay, challenge timeout, sender ban, and fencing rejection. An event contains stable reason, sender, and time without secrets, plaintext keys, or a full sensitive payload.

Counterexample and fault injection

- Tamper with a signed payload and expect authentication failure plus audit.
- Submit one nonce twice and expect replay rejection.
- Require encryption then send plaintext and expect explicit rejection.
- Send enough bad messages to ban a sender and observe recovery after expiry.
- Raise minimum key version to 2 and reject a version 1 envelope.

Experiment

```
cd submodule/dsm
mvn -q -pl dsm-security,dsm-integration-test -am \
  -Dtest='HmacMessageAuthenticatorTest,SlidingWindowNonceTrackerTest,'\
'SharedKeyClusterTokenValidatorTest,SenderRateLimiterTest,'\
'BuiltInMessageEncryptorTest,SecurityIntegrationTest' \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

Use `security-gates.json` to check input, rejection reason, metric, and audit outcome for each gate; secret values are excluded.

Experiment acceptance card

Field	Content
Command	The security unit/integration tests above plus book assets
Input or fault	token mismatch, tamper, plaintext mismatch, replay, sender ban, and old key
Observable result	messages fail closed; structured failure, metric, and audit agree; rolling rotation remains compatible
Evidence level	E2 cryptographic/state components plus E3 security integration
This experiment does not prove	Key custody, certificate infrastructure, production rotation, retention compliance, or audited network boundaries

Review

- Admission, authentication, encryption, replay prevention, rate limiting, and audit are separate responsibilities.

- Rotation needs a dual-version compatibility window followed by a higher minimum version.
- A nonce tracker combines replay protection with a sender-capacity bound.
- Security fails closed without putting secrets into observation data.

Next

Chapter 21 defines startup, registry freeze, readiness, and shutdown so deployment automation does not treat STARTING as RUNNING.

Evidence links

- `HmacMessageAuthenticator`
- `SlidingWindowNonceTracker`
- `SecurityAuditLog`
- `SecurityIntegrationTest`

Chapter 21 —Runtime Lifecycle and Traffic Admission

Chapter objective: After this chapter, a developer can define deployment lifecycle through Runtime state, registry freeze, readiness, and shutdown order without treating the presence of a Spring Context as DSM readiness.

Learning objectives

1. Explain `CREATED → STARTING → RUNNING → STOPPING → CLOSED`.
2. Explain why collection registration freezes at startup.
3. Distinguish liveness, readiness, and diagnostics.
4. Order ingress, background tasks, membership, and resource release during shutdown.

Prerequisites

- Runtime lifecycle from Chapter 2 and Spring assembly from Chapter 16 are complete.
- Diagnostics and last error from Chapter 19 are understood.
- The deployment platform can read readiness independently of process liveness.

Case progress

The fulfillment service begins a rolling deployment. An old instance stops taking new work before releasing resources; a new instance receives traffic only after collection freeze, membership/sync startup, and RUNNING. This avoids a window where the application process exists while the coordination plane is unready.

The state machine is a permission table

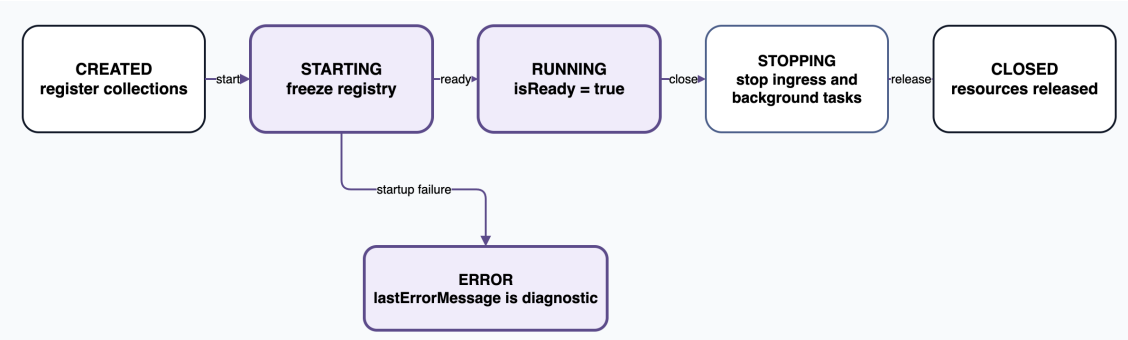

Figure 21: Runtime lifecycle and registry freeze

State	Principal allowed action	Deployment meaning
`CREATED`	register collections and read static identity	no business traffic
`STARTING`	freeze registry and start lifecycle components	readiness false
`RUNNING`	serve local operations and background collaboration	`isReady()` true
`STOPPING`	stop ingress and background tasks	remove from load balancing
`CLOSED`	no further Runtime use	resources released
`ERROR`	read `lastErrorMessage` and recover	no traffic; never report ready

Why startup freezes the registry

A collection descriptor participates in capability/descriptor advertisement and sync routing. Adding one after communication begins creates a dynamic window in which some peers know the contract and others do not. Default registry freezes on start and rejects later registration. Dynamic registration would require an explicit migration protocol.

Liveness and readiness differ

- **Liveness:** process and main thread remain alive.
- **Readiness:** Runtime is RUNNING with required configuration and collections.
- **Diagnostics:** explains unready state such as ERROR and last error.

A context under construction, STARTING Runtime, or ERROR Runtime does not report ready. `DsmRuntimeLifecycle` stops Runtime on context close, while the deployment platform still removes traffic first; a JVM shutdown hook alone is not a graceful-deployment protocol.

Business meaning of shutdown order

1. Set readiness false and stop new traffic.
2. Drain local business requests and short tasks.
3. Stop subscription consumers and repair/sync scheduler ingress.
4. Close Runtime, membership, and transport resources.
5. Record incomplete, uncertain, or compensating business actions.

DSM close releases local resources. It does not roll back committed order transactions or external effects.

Counterexample and fault injection

- Register a collection after Runtime start and expect registry-freeze rejection.
- Return HTTP 200 readiness during STARTING and identify the invalid traffic window.
- Close Spring Context and confirm Runtime is no longer RUNNING.
- Check only liveness after startup failure and observe how an ERROR node could receive traffic.

Experiment

```
cd submodule/dsm
mvn -q -pl dsm-runtime,dsm-spring-boot-autoconfigure -am \
 -Dtest=DefaultDsmRuntimeTest,DefaultDsmCollectionRegistryTest,DsmAutoConfigurationTest,DsmSpringBootSmokeTe
  -Dsurefire.failIfNoSpecifiedTests=false test
```

Complete `lifecycle-drill.json` with allowed action, readiness, and failure handling for each state.

Experiment acceptance card

Field	Content
Command	The focused Runtime registry and Spring lifecycle tests above
Input or fault	start, registration after freeze, context close, and startup failure
Observable result	explicit transitions; late registration rejected; lifecycle stopped after close
Evidence level	E2: real Runtime and Spring ApplicationContext lifecycle
This experiment does not prove	Deployment drain, real load-balancer removal, long-transaction compensation, or forced container termination

Review

- CREATED/STARTING cannot take traffic; readiness begins at RUNNING.

- Registry freeze fixes the collection protocol surface after startup.
- Diagnostics explains ERROR; liveness does not admit it into the service pool.
- Graceful shutdown also depends on business ingress and deployment automation.

Next

Chapter 22 defines capacity evidence with a specific JMH hot path instead of converting one short-run number into a production SLA.

Evidence links

- `DsmRuntimeState`
- `DsmRuntime`
- `DefaultDsmCollectionRegistryTest`
- `DsmRuntimeLifecycle`

Chapter 22 —From Microbenchmark to Production Capacity Evidence

Chapter objective: After this chapter, a developer can select one DSM hot path, fix the JMH environment and parameters, retain raw JSON, and draw a capacity conclusion only under comparable conditions.

Learning objectives

1. Distinguish benchmark-harness smoke, microbenchmark result, and production capacity.
2. Select fixtures for Register, Lease, CRDT, ChangeStream, and security.
3. Record JDK, CPU, parameters, units, and raw result.
4. Separate throughput, latency distribution, and distributed end-to-end performance questions.

Prerequisites

- Backpressure from Chapter 13 and metrics from Chapter 19 are complete.
- Maven/JMH can run.
- A microbenchmark excludes production network, GC configuration, business payloads, and dependencies.

Case progress

“Can DSM handle the traffic?”is too broad for one measurement. The fulfillment team separates RecordCodec latency, Register get, Lease renew, CRDT apply/merge, ChangeStream overflow, and secure-envelope overhead into named hot paths.

Evidence chain for a capacity conclusion

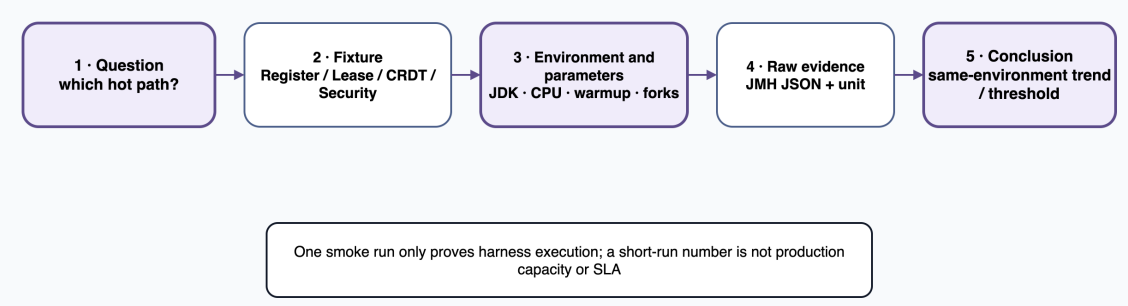

Figure 22: From a hot-path question to reproducible capacity evidence

The benchmark module contains:

- `RecordCodecLatency`, `ProtobufSerializationLatency`
- `RegisterGetLatency`
- `LeaseRenewThroughput`
- `CrdtApplyThroughput`, `CrdtMergeThroughput`
- `ChangeStreamOverflowPolicyThroughput`
- `SecureEnvelopeEncryptionOverhead`
- HLC, striped-lock, and QoS-scheduler hot paths

Each fixture answers only the local question in its name.

Smoke run and measurement run

`BenchmarkHarnessTest` checks request defaults, parameter validation, and JSON structure. A short JMH run proves the harness executes; insufficient warmup, forks, or measurement time makes its numbers unsuitable for a capacity commitment.

Reproducible smoke commands:

```
cd submodule/dsm
mvn -q -pl dsm-benchmark -am test
mvn -q -pl dsm-benchmark -DskipTests exec:exec \
  -Dbenchmark.include=RegisterGetLatency \
  -Dbenchmark.warmupIterations=0 \
  -Dbenchmark.measurementIterations=1 \
  -Dbenchmark.measurementTimeMs=100 \
  -Dbenchmark.forks=0
```

This is a CI/teaching smoke configuration rather than a recommended reporting configuration.

Required result-card fields

Field	Example
revision	DSM `505e7557`
environment	macOS, JDK 25, CPU/memory, power mode
benchmark	full class/method name
fixture parameters	payload, entry count, threads
JMH parameters	warmup, measurement, forks
mode / unit	throughput or sample time; ops/s or ns/op
raw evidence	JSON path and SHA-256
interpretation boundary	excludes network, business IO, cluster contention

Copying one average discards units, error, percentiles, and run conditions.

From microbenchmark to capacity plan

Microbenchmarks reveal hot-path regressions. Production capacity also includes entity size and key distribution, node count, latency/loss, repair traffic, GC, contention, security overhead, observation backends, and workload peaks. Final load tests use independent processes in a production-like environment.

Counterexample and fault injection

- Publish a 100 ms `forks=0` smoke number as an SLA and identify missing conditions.
- Compare different JDK/CPU runs without environment records and observe irreproducibility.
- Infer cross-node repair throughput from `RegisterGetLatency` and identify the execution mismatch.
- Delete the JMH JSON while retaining a Markdown number and observe loss of raw evidence.

Experiment

Run the harness test and one short smoke. Record environment and parameters in `benchmark-run-card.json`. The book validates the card fields and does not freeze the short-run score into prose.

Experiment acceptance card

Field	Content
Command	`BenchmarkHarnessTest` plus JMH smoke for one named fixture
Input or fault	explicit include, warmup, measurement, forks, and environment
Observable result	harness writes JSON with mode/unit; command is reproducible
Evidence level	E2: component benchmark harness and local smoke
This experiment does not prove	Production throughput, P99 SLA, cross-node capacity, peak repair, or cost budget

Review

- A specific hot-path question selects the fixture.
- Smoke proves executability, not capacity.
- A result binds environment, parameters, unit, raw JSON, and revision.
- Production capacity needs independent end-to-end load and fault tests.

Next

Chapter 23 starts Arc 6 with a real TCP port and Redis client to observe DSM replication, repair, and protocol boundaries as E4 evidence.

Evidence links

- `BenchmarkMain`
- `BenchmarkHarnessTest`
- `RegisterGetLatency`
- `SecureEnvelopeEncryptionOverhead`

Arc 5 Recap —Turn Operating Responsibilities into Contracts (Chapters 19–22)

Observability, security, lifecycle, and capacity together form DSM's operating responsibility and directly affect fault detection, isolation, and recovery.

Four operating contracts

Contract	Useful question	Dangerous substitute
Observability	Which identity/locator/origin/reason differs?	"Sync looks wrong; restart it."
Security	Which admission/authentication/encryption/replay/abuse gate rejected?	"Security is enabled."
Lifecycle	Which state accepts traffic; when does registry freeze and close?	"Spring Context is up."
Capacity	Which fixture, environment, and parameters support this trend?	"The laptop produced a large number."

How this arc realizes DSM's value

Shared diagnostics, metrics, traces, security gates, and lifecycle reduce repeated platform work across services. Comparable benchmarks identify hot-path regressions. Each target environment still owns alert thresholds, key custody, traffic drain, and production capacity.

A pre-release path

1. Check RUNNING state, collection registration, and readiness.
2. Verify that diagnostics, metrics, and traces can locate membership/delta/repair/consumer differences.
3. Validate token, signature, encryption policy, nonce, abuse limit, and audit outcomes.
4. Rehearse readiness removal, drain, Runtime close, and incomplete-action records.
5. Run the target microbenchmark under a pinned revision/environment, retain JSON, and keep smoke separate from SLA.

Still unproven

- Observation backend, thresholds, and on-call process work during a real incident.
- Key custody, production rotation, and audit retention satisfy organizational requirements.
- Deployment automation drains real traffic and recovers from forced termination.
- Microbenchmarks represent production networking, peak repair, and business workload.

These gaps become inputs to the Arc 6 E4 lab and final release decision.

Next

Arc 6 adds independent-process/real-port evidence with a Redis-shaped black box, then addresses Federation, alternatives, and the capstone.

LEARNING ARC

Prove Boundaries and Deliver

Chapters 23–26

Complete the capstone through real ports, cross-cluster boundaries, exclusion decisions and evidence.

Chapter 23 —Observe Consistency Boundaries through Real Ports

Chapter objective: After this chapter, a developer can reproduce propagation between instances using two independent RESP clients, real TCP/UDP ports, and repair diagnostics, and explain why convergence cannot revoke success returned during a partition.

Learning objectives

1. Distinguish the RESP command layer, local projection, DSM mutation batch, and replication/repair layer.
2. Observe online delta, late join, and TCP repair over real sockets.
3. Reproduce two successful `SET NX` calls and duplicate List pop.
4. Separate E4 black-box evidence from production deployment acceptance.

Prerequisites

- The partition window from Chapter 14 and evidence ladder from Chapter 18 are complete.
- Java 25 and Maven are available; `redis-cli` is optional for the manual demonstration.
- The example implements a Redis-shaped protocol and is not a Redis-compatible product.

Case progress

The fulfillment team needs a black-box experiment readable by business stakeholders. Node A listens for RESP on `6380`, node B on `6381`; clients see only Redis-style commands and results while UDP live delta and TCP repair propagate deterministic mutation batches.

The experiment boundary

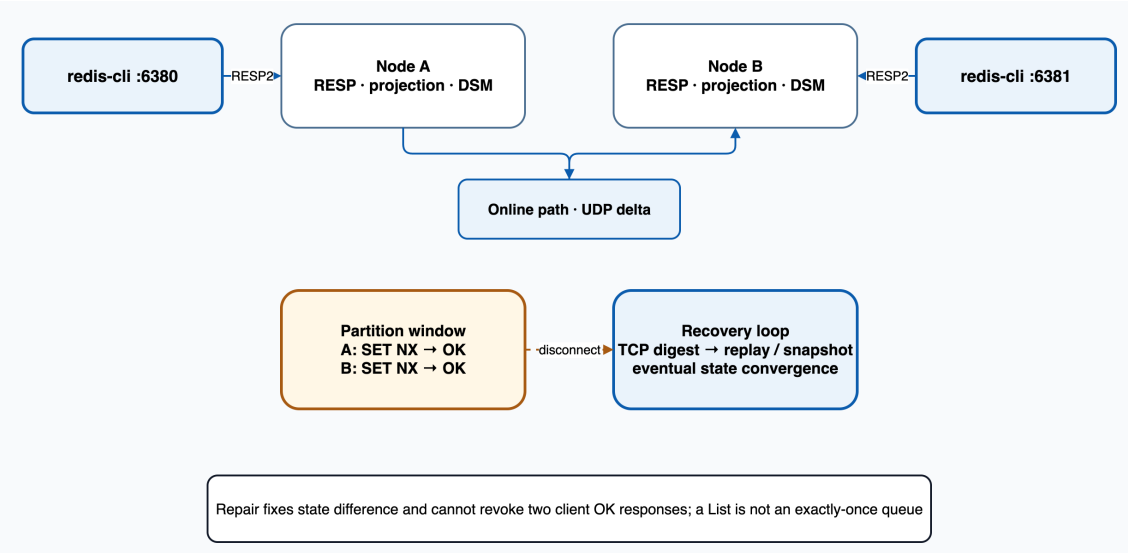

Figure 23: Real RESP ports, partition window, and repair

The protocol layer accepts RESP2 bulk-string arrays and limits transaction queue and mutation-batch size. A projection supplies local String, Hash, List, Set, TTL, and counter views. DSM replicates deterministic mutation batches; it does not pretend that a Redis server is shared memory.

Online and repair paths

While both nodes are online, subsequent writes propagate through UDP live delta. A late or missed node exchanges digests over the TCP data plane and selects replay or snapshot from the available window. `INFO DSM` exposes local

`dsm_repair_state`, last mode, time, peer, and error.

These fields describe one node. They do not form a globally consistent query. `digest_match` means that one comparison found no difference.

Both partition-time OK responses happened

`SET key value NX` checks the ingress node's current projection. After isolating A and B, both can see an absent key and return OK. Recovery and repair converge to one deterministic state without revoking either client's earlier response.

`LPOP/RPOP` also acts on local projection, so both sides can pop one logical element during a partition. The example is unsuitable as an exactly-once queue, global CAS, or inventory-deduction system.

Counterexample and fault injection

- Cut live delta, run `SET NX` for one key on A and B, and record both results.
- Pop one List element on both sides and observe that both clients can receive it.
- Write String and Hash state on A while B is offline, then start B and observe bootstrap repair.
- Send a mutation batch larger than 32 KiB and confirm whole-batch rejection without partial apply.

Experiment

Run the automated black box:

```
cd submodule/dsm-examples/dsm-redis-server
./mvnw -q clean verify
```

Run the optional manual boundary demonstration:

```
cd submodule/dsm-examples/dsm-redis-server
./scripts/run-consistency-boundary-demo.sh
```

Use `redis-boundary-observations.json` to record client result, node-local view, repair outcome, and unsupported conclusion.

Experiment acceptance card

Field	Content
Command	Redis-shaped example `clean verify`; optional boundary script
Input or fault	real RESP socket, UDP live delta, late node, partition, and TCP repair
Observable result	commands are reachable; online propagation and repair are visible; two local successes converge
Evidence level	E4: independent socket/real port/client protocol black box on local loopback
This experiment does not prove	Complete Redis compatibility, cross-host networking, production security, persistence, global CAS, or exactly-once

Review

- Real ports strengthen the execution surface without changing collection semantics.
- Online delta and repair are complementary paths.
- Repair restores convergence and cannot revoke returned business success.
- A familiar protocol is an observation entry point, not a Redis product promise.

Next

Chapter 24 expands to two clusters and states what Register, Lease, and CRDT retain and give up across federation.

Evidence links

- Redis-shaped example boundaries
- `RealNetworkBlackBoxTest`
- `DsmRedisNodeIntegrationTest`
- Consistency-boundary demonstration script

Chapter 24 —Collection Semantics across Clusters

Chapter objective: After this chapter, a developer can describe federation propagation, recovery, and ownership boundaries separately for Register, Lease, and CRDT, and reject claims of automatic global strong consistency.

Learning objectives

1. Identify remote cluster, service, and locator with `FederationTarget`.
2. Distinguish federation semantics for the three collection types.
3. Interpret relay health and binding status.
4. Identify the gap between an in-process baseline and production cross-region transport.

Prerequisites

- Collection choice, repair, and security gates from Chapters 9, 12, and 20 are complete.
- Intra-cluster replication and federation are separate failure domains.
- A cross-cluster link needs independent security, capacity, and recovery validation.

Case progress

The fulfillment team runs independent clusters in two regions. Route hints need cross-region visibility, shard owners are observed remotely, and request counters can merge. One vague "synchronization policy"cannot cover all three invariants.

Three collections cross one boundary with different semantics

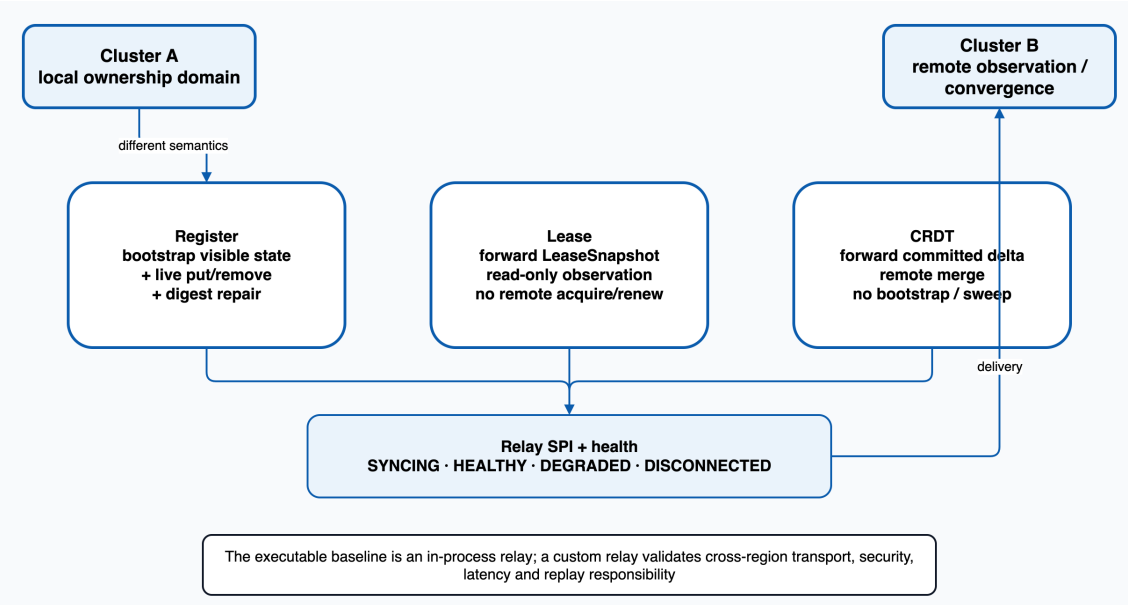

Figure 24: Federation semantics for Register, Lease, and CRDT

Type	Initial state	Live changes	Catch-up path	Cross-region boundary
Register	bootstrap current visible snapshot	forward put/remove	periodic digest repair; rebootstrap if needed	eventual convergence, no cross-region transaction
Lease	no bootstrap	forward `LeaseSnapshot`	no anti-entropy sweep	remote read-only observation; no acquire/renew
CRDT	no bootstrap	forward committed delta	no federation sweep	depends on merge-able/idempotent delta and eventual delivery

A remote Lease snapshot does not create cross-region ownership. Owner and fencing authority remain in the source cluster; the snapshot can support routing or operations observation, not permission for an external effect.

Target is complete identity, not a URL

`FederationTarget` contains `remoteClusterId`, `remoteServiceId`, and `remoteLocator`. The relay finds Runtime by cluster/service and collection by locator. A mismatch at any layer fails explicitly; equal collection names do not imply equal state.

Health qualifies confidence in propagation

A binding is `SYNCING`, `HEALTHY`, `DEGRADED`, or `DISCONNECTED`. Forwarding stops after relay failure; Register performs catch-up bootstrap after recovery. Binding health says nothing about global confirmation of business facts.

The current implementation boundary

`InMemoryFederationRelay` connects independent Runtimes in one process. It executes real bridge semantics for three collections but has no gRPC/HTTP, cross-host link, authentication, retry budget, or production latency. A production relay implements bootstrap, put/remove, repair, Lease snapshot, and CRDT delta, then separately satisfies the security/capacity contracts from Chapters 20–22.

Counterexample and fault injection

- Set a wrong target service ID and observe failure to find the remote Runtime/collection.
- Write Register during relay failure and confirm queued/bootstrap catch-up on recovery.
- Try to infer Lease-acquire permission from a remote snapshot and identify the read-only violation.
- Drop a CRDT delta and note that current federation has no CRDT anti-entropy sweep.

Experiment

```
cd submodule/dsm
mvn -q -pl dsm-federation,dsm-integration-test -am \
  -Dtest=InMemoryFederationBridgeTest,FederationIntegrationTest \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

Use `federation-contract.json` to check bootstrap, live delivery, repair, and ownership for each collection.

Experiment acceptance card

Field	Content
Command	Federation unit/integration tests plus book assets
Input or fault	bootstrap, live mutation, relay failure/recovery, Lease snapshot, CRDT delta

Field	Content
Observable result	Register catches up; Lease remains observational; CRDT merges; status follows relay health
Evidence level	E2/E3: two independent Runtimes plus in-process relay
This experiment does not prove	Real cross-region transport, security, latency, bandwidth, complete history, or global transaction

Review

- Federation is a cross-failure-domain replication contract, not an extended cluster.
- Register, Lease, and CRDT retain distinct semantics.
- A remote target includes cluster, service, and locator.
- The current relay is an executable baseline, not production networking.

Next

Chapter 25 asks which state should stay outside DSM and when a database, cache, event log, or consensus service fits better.

Evidence links

- `dsm-federation` boundaries
- `FederationBridge`
- `InMemoryFederationBridgeTest`
- `FederationIntegrationTest`

Chapter 25 —Identify State That Does Not Fit DSM

Chapter objective: After this chapter, a developer can select technology from business invariants, failure consequences, and evidence requirements without defaulting to DSM because a problem is low-latency, distributed, or "shared."

Learning objectives

1. Decide whether state is repairable before discussing collection type.
2. Distinguish system of record, cache, history log, consensus coordination, and DSM.
3. Write a reviewable reason for adopting or rejecting DSM.
4. Detect dual-write, unbounded-state, and unobservable-repair anti-patterns.

Prerequisites

- All three collections, partition boundaries, capacity, and federation are complete.
- Business consequences of loss, duplicate, reordering, and divergence can be stated.
- "Do not use DSM"is accepted as a successful design conclusion.

Case progress

The fulfillment team reviews orders, inventory deductions, route hints, shard owners, request counters, and audit events again. The goal is correct ownership of each state rather than larger DSM coverage.

Eliminate unsuitable responsibilities before choosing a collection

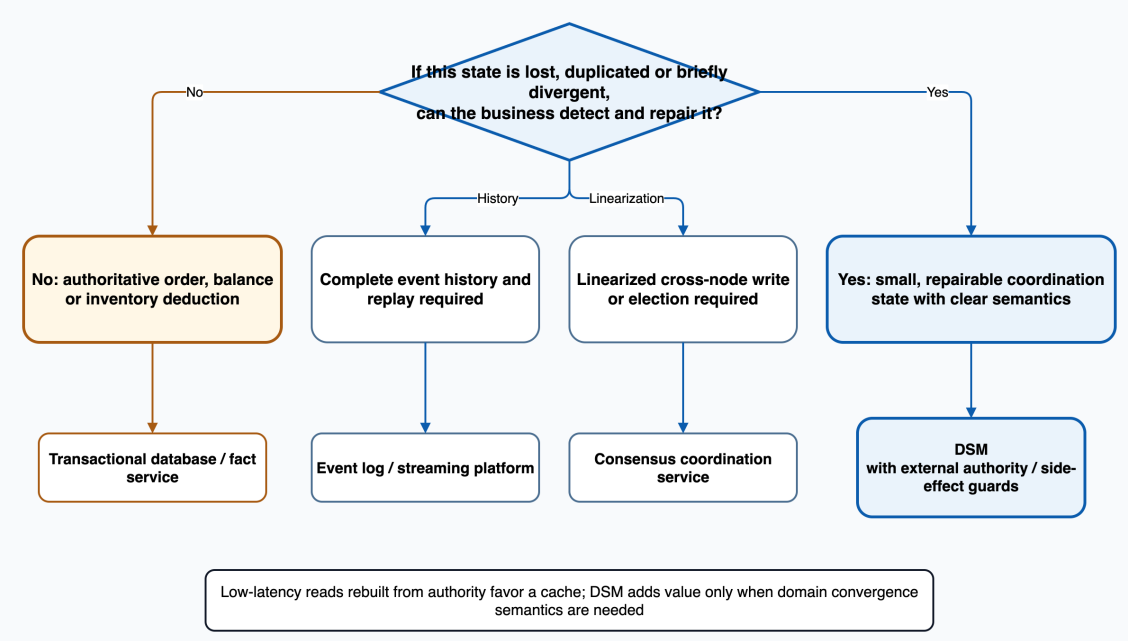

Figure 25: Decision tree for excluding unsuitable DSM state

Requirement	Primary responsibility	Why DSM is not primary
authoritative order/balance/inventory fact	transactional database or fact service	confirmed transactions cannot be rewritten by eventual convergence

Requirement	Primary responsibility	Why DSM is not primary
complete replayable history	event log or streaming platform	DSM centers on current coordination state, not complete history
linearized configuration, election, or strong CAS	consensus coordination service	DSM can permit local success during a partition
low-latency read rebuilt from a fact source	cache	no need for domain merge, repair, and protocol identity
small, repairable coordination state with clear semantics	DSM candidate	Register, Lease, or CRDT may express the invariant

A candidate still needs bounded size, TTL/cleanup, schema evolution, observable failure, repair cost, security, capacity, and operating ownership.

Three direct rejection signals

1. **DSM and database both write authority:** dual write without one authority and compensation adds another divergence layer.
2. **Unbounded state growth:** full order history, logs, or analytical details make lifecycle and memory capacity uncontrollable.
3. **Unexplained failure window:** "eventual consistency"does not answer two OK responses, old tokens, missed deltas, or slow consumers.

Final boundary of the running case

- `route-hints`: Register candidate; health/service management remains authoritative.
- `shard-owners`: Lease candidate; external writes validate fencing tokens.
- `request-counter`: PN-Counter candidate for mergeable observation, not settlement.
- Orders and inventory: DSM rejected as system of record.
- Audit events: immutable history system, not ChangeStream alone.

Counterexample and fault injection

- Store inventory balance in a Register, create partitioned writes, and observe that the eventual winner loses a business deduction.
- Treat ChangeStream as audit history, overflow it, and compare event loss with completeness requirements.
- Treat a CRDT counter as a bill and identify source duplication, reversal, and legal-record gaps.
- Add a custom resolver for pure cache data and compare its complexity with rebuilding from authority.

Experiment

```
npm test -- --test-name-pattern="phase 6 assets"
```

In `technology-selection-review.json`, record invariant, failure cost, source of truth, decision, and counterfactual condition for eight state categories.

Experiment acceptance card

Field	Content
Command	Book static decision-asset test
Input or fault	fact, history, strong coordination, cache, and repairable coordination state
Observable result	every item has an adoption/rejection reason and a condition that would change the decision
Evidence level	E1: design review combined later with target-system runtime evidence
This experiment does not prove	That an alternative product meets SLA, security, cost, or team capability

Review

- Repairability is evaluated before collection type.
- DSM does not own order facts, complete history, or global linearizability.
- Rebuildable low-latency reads start as a cache question; multiple instances alone do not justify a consistency framework.
- Rejection receives the same reproducible reasoning as adoption.

Next

Chapter 26 combines the method into an executable fulfillment control plane with faults, observation, and explicit boundaries.

Evidence links

- Running-case design
- Collection selection
- Consistency commitment boundary
- Capacity evidence contract

Chapter 26 —Deliver a Reviewable Fulfillment Control Plane

Chapter objective: After this chapter, a developer can run the book-owned fulfillment control plane, connect all three collection types to fault experiments, observation signals, and evidence levels, and make a bounded release decision.

Learning objectives

1. Trace from a business contract to a DSM adapter without exposing collection APIs to business code.
2. Accept normal paths and failure windows for Register, Lease, and CRDT together.
3. Re-run book, example, observation, security, lifecycle, federation, real-port, and publication evidence with one script.
4. State what can be released and what still needs deployment validation.

Prerequisites

- Chapters 1–25 are complete.
- Java 25, Maven/Wrapper, and Node.js 22+ are available.
- DSM `0.1.2` is installed in the local Maven repository, or a full DSM build runs first.

Case progress

The fulfillment control plane publishes healthy routes, claims shards, and counts accepted requests. Business code depends only on `FulfillmentCoordination`; `DsmFulfillmentModule` owns Runtime, locators, codecs, and typed handles inside the composition root.

Acceptance follows a chain

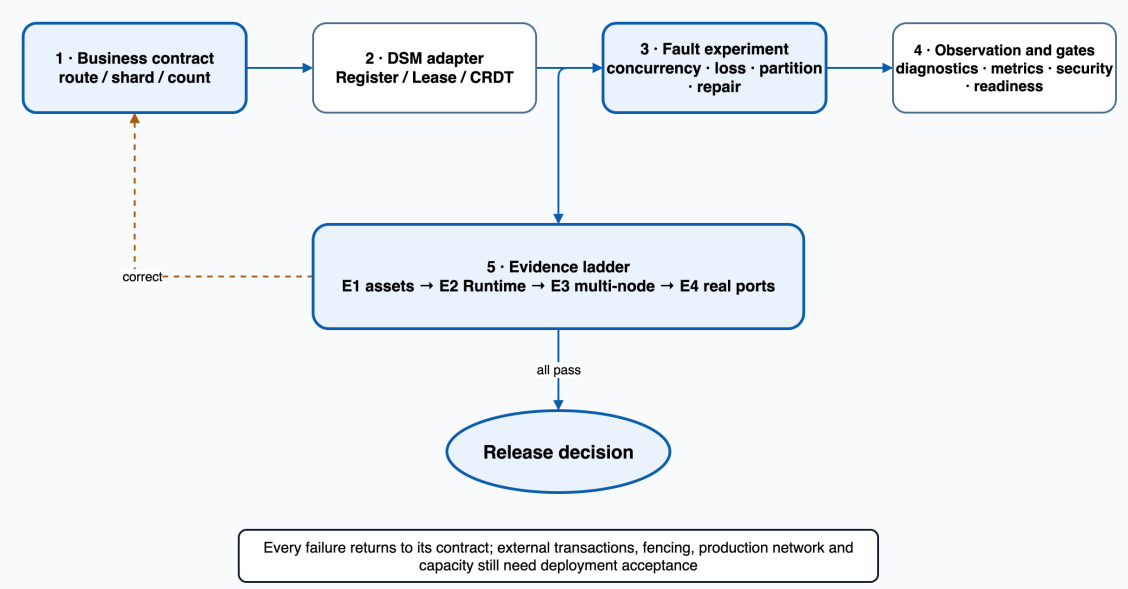

Figure 26: Contract, fault, and evidence chain for the capstone

Layer	Acceptance question	Return on failure
business contract	Does the caller see route, claim, count, uncertain, and fencing only?	domain port and invariant

Layer	Acceptance question	Return on failure
collection adapter	Does each collection express the intended state?	locator, codec, and collection choice
fault experiment	Are concurrency, partition, loss, and repair outcomes explicit?	failure contract and external guard
operating responsibility	Can the system diagnose, reject untrusted messages, and ready/stop correctly?	Arc 5 operating contracts
evidence level	On which execution surface was each claim validated?	traceability matrix and command

Final realization of DSM's value

The capstone compresses the book into one executable path. Domain ports express route publishing, shard claiming, and request counting. The DSM adapter reuses Register, Lease, CRDT, propagation, and repair. Tests and observation surfaces state the execution environment in which each mechanism holds.

Business teams avoid rebuilding these coordination foundations in every service while retaining responsibility for order facts, external fencing, partition-time response policy, and target-environment acceptance. A learner completes the path by selecting or rejecting DSM from an invariant, running normal and fault experiments, explaining observations, and converting uncovered scope into deployment tasks.

Normal path is not final acceptance

The book Maven example validates a domain fake and real standalone Runtime. DSM integration validates multi-node propagation, repair, security, and federation. The Redis-shaped example validates real ports. Together they still leave production networking, external transactional fencing, deployment drain, and capacity SLA open.

One-command acceptance

```
node scripts/verify-capstone.mjs
```

The script runs serially so Maven reactors do not share `target` directories concurrently. It stops on the first failing command and names every evidence layer. Finally it re-exports Draw.io diagrams and builds both online editions and PDFs so content and publications share one candidate state.

Release-decision template

Confirmed: On pinned revisions, the domain-port boundary holds; all three collections run; controlled multi-node propagation/repair, security/federation, and loopback RESP tests pass; chapter assets, links, diagrams, and publication builds are reproducible.

Deployment acceptance still needed: production topology, cross-host networking, external-store fencing, metrics backend/on-call process, key custody, real drain, data scale, peak repair, capacity, and disaster recovery.

Every open item has an owner, environment, command/probe, threshold, and failure response. A book green result cannot close it.

Counterexample and fault injection

- Change an asset without its manifest, or return `DsmRegister` from business code, and observe gates reject evidence drift or technical-type leakage.
- Turn loopback Redis evidence into a cross-host production claim, or run Maven reactors against shared `target` directories in parallel, and observe boundary review or serial gating prevent a false green.

Experiment

1. Run `node scripts/verify-capstone.mjs`.
2. Check each claim, evidence item, and boundary in `capstone-acceptance.json`.
3. Create a separate acceptance card for the target deployment; the book card does not establish production completion.

Experiment acceptance card

Field	Content
Command	`node scripts/verify-capstone.mjs`
Input or fault	three collections, domain port, repair/security/lifecycle/federation, real RESP socket, and publication pipeline
Observable result	all pinned evidence passes serially; a failing command exits nonzero; boundary card is complete
Evidence level	Combined E1–E4 selected per claim, without applying the highest level to every conclusion
This experiment does not prove	Target organization and production environment have approved release

Review

- Delivery is a chain of business contract, runtime implementation, fault experiment, evidence, and boundary.
- One script improves reproducibility without raising evidence level by itself.
- Every release decision binds revision, environment, and open claims.
- Completion means both buildable modules and a learner who can reproduce experiments and make a boundary-consistent decision.

Next

Use the appendices for APIs, Spring properties, troubleshooting, terminology, and evidence. When the DSM revision changes, Appendix E identifies chapters requiring review.

Evidence links

- DSM examples: Basic Usage
- Capstone verification script
- Content traceability matrix
- Runnable example matrix

Arc 6 Recap —Graduate with Boundaries and Evidence (Chapters 23–26)

DSM provides explicit semantics, recovery paths, and evidence boundaries for suitable shared coordination state. The corresponding systems remain authoritative for business facts.

Four graduation decisions

1. **Protocol:** real ports prove reachability; Redis-shaped is not a Redis product and does not change partition semantics.
2. **Scope:** federation preserves different contracts for three collections and creates neither a global transaction nor cross-region Lease ownership.
3. **Exclusion:** authoritative facts, complete history, strong coordination, and pure cache return to systems fitted to those duties.
4. **Release:** evidence level follows the claim and separates current green results from deployment acceptance.

How this arc realizes DSM's value

Real ports, federation, technology exclusion, and the capstone test the limits of DSM's value. The team can name consistency work reused from DSM, business/production responsibilities remaining outside, and the next decision: adopt, reject, or validate further.

Final mapping of the running case

State	Final choice	External responsibility retained
route hint	Register	health authority, TTL/cleanup, bad-route handling
shard owner	Lease	external fencing, uncertainty, partition policy
request counter	PN-Counter	mergeable observation only, no settlement
order and inventory	reject DSM as authority	transaction, audit, compensation
cross-region coordination	federation selected per collection	real relay, security, capacity, recovery

Verifiable learning outcomes

- Select Register, Lease, CRDT, or rejection from a business invariant.
- Explain the difference among local commit, remote visibility, repair, and business success.
- Locate problems with diagnostics/metrics/trace while controlling metric cardinality.
- State honest security, lifecycle, capacity, and federation boundaries.
- Run the capstone while keeping book evidence separate from target deployment acceptance.

Final sentence

Distributed shared memory does not disguise remote access as local access. DSM gives inevitable divergence explicit collection semantics, recovery processes, and evidence boundaries so business teams can understand it, engineering systems can handle it, and experiments can verify it.

Appendix A —Runtime and Collection API Quick Reference

This appendix locates entry points. It does not replace chapter semantics, failure windows, or evidence boundaries. APIs are pinned to DSM `505e7557` / `0.1.2`.

Runtime

Action	Entry point	Key constraint
Build	`DsmRuntimeBuilder.builder()`	fix `clusterId`, `serviceId`, membership, and security/observation extensions
Register Register	`runtime.register(spec)`	complete before `start()`; locator is unique
Register Lease	`runtime.leaseRegister(spec)`	provide codec, entity factory, and Lease policy
Register CRDT	`runtime.crdt(spec, merger)`	provide update/state codecs, initial state, and merger
Start	`runtime.start()`	freezes registry; success enters `RUNNING`
Diagnose	`runtime.diagnostics()`	immutable local snapshot, not a global view
Readiness	`runtime.isReady()`	true only in `RUNNING`
Close	`runtime.close()`	releases local resources; does not reverse business effects

Shared collection identity

```
CollectionLocator = tenantId / applicationId / collectionId
CollectionSpec = locator + schemaId/fingerprint + codec + consistency tier
```

Equal `entryKey` values identify the same logical state only inside the same communication domain and locator. An incompatible schema is rejected instead of guessed.

Register

Operation	Meaning	No guarantee of
`put(entity)`	commit a current candidate locally	remote visibility on return
`get(key)`	value visible to this node	linearized global read
`remove(key)`	commit removal locally	completed removal on every peer
`all()` / `size()`	query local projection	exact global snapshot
`changes(options)`	bounded observation stream	lossless historical log

A resolver remains deterministic; dependence on argument order is rejected or backed by a commutativity proof.

Lease

Operation	Result focus
`acquire`	`granted`, `uncertain`, holder, expiry, fencing token, reason

Operation	Result focus
`renew`	token/holder remains valid and membership/mode permits renewal
`transfer`	new owner and higher fencing token
`release`	releases coordination state; does not revoke external writes

The external resource validates the fencing token. AUTONOMOUS mode can temporarily produce dual holders during a partition; QUORUM fails closed during unstable membership or missing majority.

CRDT

Action	Meaning
`apply(update)`	merge a local update into state and produce a propagatable delta
`state()`	merged state visible on this node

A merger is commutative, associative, and idempotent. A PN-Counter fits mergeable operational counts and does not automatically become a financial fact.

Common builders

- `CollectionSpecBuilder.register(...)`
- `LeaseCollectionSpecBuilder.lease(...)`
- `CrdtCollectionSpecBuilder.crdt(...)`

Set a stable locator and `schemaId`, then supply typed codecs. Construct specs in one assembly boundary rather than across business services.

Source entry points

- `DsmRuntime`
- `CollectionLocator`
- `DsmRegister`
- `DsmLeaseRegister`
- `DsmCrdtCollection`

Appendix B —Spring Boot Configuration Quick Reference

Properties begin at `dsm.*` and bind to `DsmProperties`. Defaults support local startup and are not production recommendations; deployment reviews network, security, capacity, and failure domains.

Top-level identity

Property	Purpose	Gate
`dsm.cluster-id`	cluster identity	required; blank fails startup
`dsm.service-id`	service family/communication isolation	equal across peers; not a tenant ID
`dsm.node.id`	node identity	random UUID by default; configure explicitly for stable diagnostics
`dsm.node.host` / `port`	advertised address	reachable from peer network

Membership

Property group	Key fields	Meaning
`dsm.cluster.mode`	`STANDALONE` / `MULTICAST` / `UNICAST`	selects membership implementation
`dsm.cluster.multicast.*`	group, port, heartbeat, failure threshold	constrained by multicast support
`dsm.cluster.unicast.*`	gossip port/interval, seeds/DNS, failure threshold	seed is discovery entry, not permanent leader

High-risk Lease decisions also check quorum and stability rounds; member count alone does not establish ownership.

Runtime and sync

`dsm.runtime.*` governs lifecycle/threads; `dsm.sync.*` governs propagation, repair, batching, and QoS. A timeout, queue, batch, or repair-period change also updates capacity experiments and fault drills.

Security

Property surface	Question
token/challenge	who enters the communication domain?
authentication/key version	who signs, and which minimum version is accepted?
encryption	is ciphertext required, with which algorithm/key version?
nonce window/senders	what replay window and sender-memory bound apply?
rate limit/ban	how do repeated failures throttle and recover?

Secrets come from deployment key management and stay out of the book, logs, metric tags, and repository configuration.

Collections

Each `dsm.collections[]` contains at least:

```
- bean-name: routeHintsCollection
  tenant-id: shared
  application-id: gateway
```

```
collection-id: route-hints
schema-id: route-hints/v1
type: REGISTER
consistency-tier: REGISTER
entity-type: com.example.RouteHint
```

Type	Additional requirement
Register	record-derived codec, or `codec-bean` for non-record
Lease	entity factory, Lease mode/TTL/quorum configuration
CRDT	update/state codecs, initial state, and merger bean

The canonical bean name is `dsmCollection:<tenant>/<application>/<collection>`; `bean-name` is an additional business alias. Duplicate locator, missing codec/merger/factory, or invalid ranges fail fast.

Validation entry point

```
cd submodule/dsm
mvn -q -pl dsm-spring-boot-autoconfigure -am \
  -Dtest=DsmAutoConfigurationTest,DsmSpringBootSmokeTest \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

- `DsmProperties`
- Chapter 16 configuration asset

Appendix C —Troubleshooting Decision Tree

Preserve evidence first, then narrow the fault in the order identity → lifecycle → membership → route → delta → repair → observer. Restart changes evidence and is not the first diagnostic action.

Entry decisions

Symptom	Inspect first	Next hypothesis
process alive, requests fail	`state`, `isReady`, `lastErrorMessage`	Runtime unready or startup configuration failed
peer completely invisible	cluster/service, member view, port	communication domain, seed, or network mismatch
one collection differs	locator, schema, collection diagnostics	locator/fingerprint mismatch
occasional online loss	message-drop reason, sync trace	delta path loss; repair pending
difference remains after repair	digest, replay window, snapshot result	source choice, regression guard, or handler failure
Lease rejects	mode, stability, quorum, token	unstable view, missing quorum, or old token
subscriber misses events	overflow/callback timeout	observer lag rather than collection-state loss
remote message rejected	admission/auth/decrypt/replay/rate limit	gate configuration or hostile behavior
federation unhealthy	target identity, relay/binding status	missing remote locator or unavailable relay

Fast inspection order

1. Record revision, node identity, time window, and exact locator.
2. Capture diagnostics before clearing errors or restarting.
3. Check low-cardinality operation/outcome/reason metrics.
4. Trace one operation; keep trace ID in log/trace rather than metric tags.
5. Review security audit without exposing token, key, or sensitive payload.
6. Compare peer capability/schema before choosing replay, snapshot, or configuration repair.
7. Record which business successes remain irreversible and which effects need compensation.

Common false greens

- `Tests run: 0`.
- Skipped tests hidden by an exit-code-only summary.
- Compile-only reported as behavior validation.
- In-process fake reported as real network.
- Local `get` reported as remote visibility.
- Convergence reported as no dual success during partition.
- JMH smoke reported as production SLA.

Minimal evidence record

```
revision:
environment/topology:
claim:
command/probe:
observed result:
does not prove:
next action / owner:
```

Related chapters

- Chapter 12: repair
- Chapter 19: diagnostic loop
- Chapter 20: security gates
- Chapter 21: lifecycle
- Chapter 23: real-port lab

Appendix D —Glossary and Commonly Confused Boundaries

Term	Meaning in this book	Do not conflate with
DSM	distributed shared-memory foundation for repairable coordination state	distributed-database replacement
Runtime	`DsmRuntime` owning identity, registry, lifecycle, and collaboration components	the JVM process itself
locator	`tenantId / applicationId / collectionId`	collection name alone
Register	current visible value per key with deterministic convergence	linearizable KV
Lease	time-bounded ownership, mode, and fencing token	distributed transactional lock
CRDT	mergeable state with commutative, associative, idempotent merge	arbitrary business data made correct automatically
local commit	local path accepts and updates projection	visibility on every peer
live delta	normal online propagation path	synchronous strong replication
repair	replay/snapshot of current replica state after digest	complete business-history replay
convergence	equivalent visible state after recovery	only one success during partition
membership	members and states observed by this node	instantaneous global truth
fencing token	monotonic token used by an external resource to reject an old owner	DSM blocking every external write automatically
ChangeStream	bounded state-change observation stream	lossless audit log
diagnostics	immutable runtime snapshot from one node	authoritative business fact
Federation	collection-semantic propagation across clusters	global transaction or one cross-region Lease owner
E1–E4	execution surfaces from static assets to real ports	quality score or production-maturity level

Five key boundaries

- `put` confirms local commit; remote visibility needs separate evidence.
- Repair restores convergence and does not revoke a returned OK.
- Lease supplies a token; the external resource validates it.
- A test proves assertions under a revision, environment, and topology, not production readiness automatically.
- Federation propagates collection semantics and does not create a cross-region transaction.

The complete editorial terminology standard is in the project glossary.

Appendix E —Capability and Evidence Index

Pinned revisions

Source	Revision	Baseline validation
DSM	`505e7557`	complete Maven suite
DSM Examples	`0eca40d`	Basic and Redis-shaped suites

The table uses short revisions; the content traceability matrix records full SHAs. A revision change reopens this index for review; old counts and conclusions do not carry forward.

Capability index

Capability	Chapters	Evidence	Level / open boundary
Runtime	2–3, 21	Runtime tests	E2 one process
two-node Register	4–6	TwoNode test	E3 multi-node
Lease/fencing	7, 14	Lease/chaos	E2/E3; external fence
CRDT	8	Runtime integration	E2; propagation later
membership	10, 17	cluster/Multicast	E2/E3; lab network
delta/repair	11–12	sync/planner/chaos	E3 repair
ChangeStream	13	buffer/Register tests	E2; no audit history
Spring	16	auto-config/Basic	E2 app context
observation	19	metrics/trace/chaos	E2/E3; backend open
security	20	security tests	E2/E3; key custody
benchmark	22	JMH smoke	E2; no capacity SLA
RESP black box	23	Redis `clean verify`	E4 loopback socket
Federation	24	bridge tests	E2/E3; in-process
business adapter	15, 26	book example	E1/E2

This index uses short evidence labels for publication width. Full test classes and commands remain in the content traceability matrix and the corresponding chapters.

One-command entry point

Run the publication gate:

```
node scripts/verify-capstone.mjs
```

It serially validates book contracts, the book-owned example, Federation, and the Redis-shaped black box. A complete publication candidate also runs the full DSM, Basic, Redis, diagram, online-book, and bilingual PDF gates listed in final review.

Control-plane links

- Content traceability matrix
- Runnable example matrix
- Quality gap ledger
- Chapter target map