Changes and Design Refinements

Brief Explanation

I did all this development process with this chronological order

I began the project by creating the **AggregationServer**, since the entire system depends on it — without a functioning server, no client interactions or data synchronization could occur. Once the server was operational, I implemented the **ContentServer**, responsible for handling PUT operations and communicating with the AggregationServer using various status codes such as 201, 200, 204, 400, and 500 to ensure clear and consistent responses. After completing the PUT logic, I added the **GET operations** to retrieve and display stored weather data, allowing the system to perform both read and write actions effectively. To make the data durable, I integrated **file persistence**, ensuring that stored information would not be lost when the server restarted. I then refactored the code to improve readability and maintainability by creating new **classes and interfaces** to separate concerns and simplify testing. Following this, I implemented **Lamport clocks** to maintain event ordering in a distributed environment, enabling the system to handle concurrent operations deterministically. I also introduced a **30-second TTL (time-to-live)** mechanism so that outdated weather data would automatically expire, keeping the system consistent and fresh. Later, I resolved challenges related to **multiple ContentServers publishing to the same AggregationServer** and ensured **ordering consistency** between concurrent updates. Finally, I refactored long methods, wrote comprehensive **unit and integration tests**, achieved over **75% code coverage using JaCoCo**, and created a **manual test guide** to validate all edge cases. Once all tests passed, I finalized the project by preparing the **UML diagrams and documentation** to clearly illustrate the system's structure and behavior.

Detailed Explanation

1) Initial Approach → Simplified Architecture

What I planned: I first aimed to design a full enterprise-style stack with separate infrastructure, business, and API layers.
What I changed: I simplified the architecture to focus on clarity and testability.
Why: My core goal was **high cohesion** and **loose coupling**. The simpler structure lets me demonstrate the required distributed-systems concepts (HTTP handling, Lamport clocks, TTL, ordering) without overengineering. It also reduces cognitive load for graders and improves maintainability.

2) AggregationServer

What it is: A simple HTTP server that accepts PUT updates and serves GET /weather.json. It applies validation, ordering, TTL checks, and persistence hooks.
What changed: I kept routing lean: one clear PUT entrypoint and one GET endpoint.
Why: Small, focused responsibilities make behavior easy to reason about and test, aligning with high cohesion.

3) ContentServer

What it is: A client that **publishes** content to AggregationServer via PUT. It reads local files (text/JSON), constructs a valid request, and handles status codes.
What changed: It now includes retry/backoff (optional) and sends Lamport metadata headers when applicable.
Why: Realistic publisher behavior (idempotent semantics, handling empty bodies, bad JSON) ensures robust end-to-end testing.

4) GetClient

What it is: A read-only client that requests /weather.json and prints a clean, human-readable view (attribute → value).
What changed: Input parsing supports host:port and path forms; output formatting is intentionally simple for screenshots/marking.
Why: Clear GET output makes it easy to verify ordering, TTL expiry, and overall correctness without debugging.

5) File Persistence

What it is: Saving the last accepted state to disk so the server can recover after restarts.
What changed: I separated persistence from core HTTP logic (e.g., a minimal persistence module/class).

Why: Separation keeps coupling low. Persistence can be swapped or extended without touching request handling.

6) Status Codes and Responses

What it is: Strict mapping of conditions to HTTP codes:

- **201 Created** on the first successful non-empty PUT,
- **200 OK** on subsequent non-empty PUTs,
- **204 No Content** when PUT body is empty,
- **400 Bad Request** for wrong paths/invalid request forms,
- **500 Internal Server Error** for malformed JSON that cannot be processed.
  What changed: I made the mapping explicit and added negative tests.
  Why: Clear status contracts make clients predictable and grading unambiguous.

7) Lamport Clocks Implementation

What it is: Logical clocks to order distributed events from multiple ContentServers. Each request carries a Lamport timestamp; the server updates its clock (receive rule) before applying.
What changed: Request/response headers now include Lamport values; the server uses them to gate visibility/order.
Why: Demonstrates core distributed-systems theory and guarantees a consistent happens-before relation for writes.

8) Refactoring for Status Codes

What it is: I pulled status-decision logic into small, focused functions
What changed: Reduced branching in handlers; clearer unit targets; less duplication.
Why: Improves readability/testability and enforces a single source of truth for HTTP outcomes.

9) Lamport Clock Tests

What it is: Manual and automated checks to confirm clock increments and correct receive rules.
How I verify:

- Start two ContentServers concurrently (A & B).
- Observe GET results and server logs to ensure the order matches Lamport timestamps, not OS scheduling.
  Why: Proves the ordering mechanism works under concurrency, not just in sequential runs.

10) TTL (30-Second Expiration)

What it is: Stored state becomes invalid after **30 seconds** of inactivity.
How I verify: PUT valid data → wait 31s → GET returns **404**.
Why: Forces freshness and demonstrates time-based invalidation—another grading requirement.

11) Replicated ContentServers

What it is: Multiple ContentServers publish to the same AggregationServer.
What changed: The system handles overlapping writes and resolves visibility using Lamport ordering.
Why: Simulates a more realistic multi-publisher environment and proves the server's concurrency correctness.

12) Ordering Consistency

What it is: A barrier that ensures clients won't observe a later write that (logically) happened after their read started.
How I verify (acid test):

1. PUT A.
2. Immediately GET, then (tiny delay) PUT B.
3. The first GET must show **A**; a later GET shows **B**.
   Why: Demonstrates linear-looking reads consistent with the chosen ordering discipline.

---

13) Refactoring Long Methods (>85 Lines)

What it is: I split long handlers into smaller, single-purpose methods (parse, validate, decide status, persist, reply).
What changed: Reduced each method's scope and cyclomatic complexity; improved names and cohesion.
Why: Easier to test, reason about, and maintain; aligns with clean-code principles and grading rubrics.

---

14) Unit Tests & ≥75% Coverage

What it is: Focused tests for parsing, status classification, Lamport updates, TTL expiry, and persistence I/O.
What changed: Added both success and failure cases; covered edge conditions (empty body, bad JSON, wrong path).

Why: Achieves **≥75% line coverage** and, more importantly, meaningful branch coverage on critical paths to increase reliability.

15) UML and Documentation

What it is: A concise UML (class & sequence snippets) plus a short readme on how modules collaborate (ContentServer → AggregationServer → Persistence; GetClient for reads).
What changed: I kept diagrams minimal but accurate to avoid drift.
Why: Diagrams + clear docs help markers quickly understand design intent, responsibilities, and data flow.

Final Reflection

Moving from a heavy multi-layer blueprint to a leaner design improved **clarity**, **testability**, and **demonstrability**. The result still achieves **high cohesion** and **loose coupling**, while directly showcasing distributed-systems fundamentals: **Lamport ordering**, **TTL freshness**, **replication behavior**, and **observable status semantics**.