Architectural Design Document

UML Diagrams

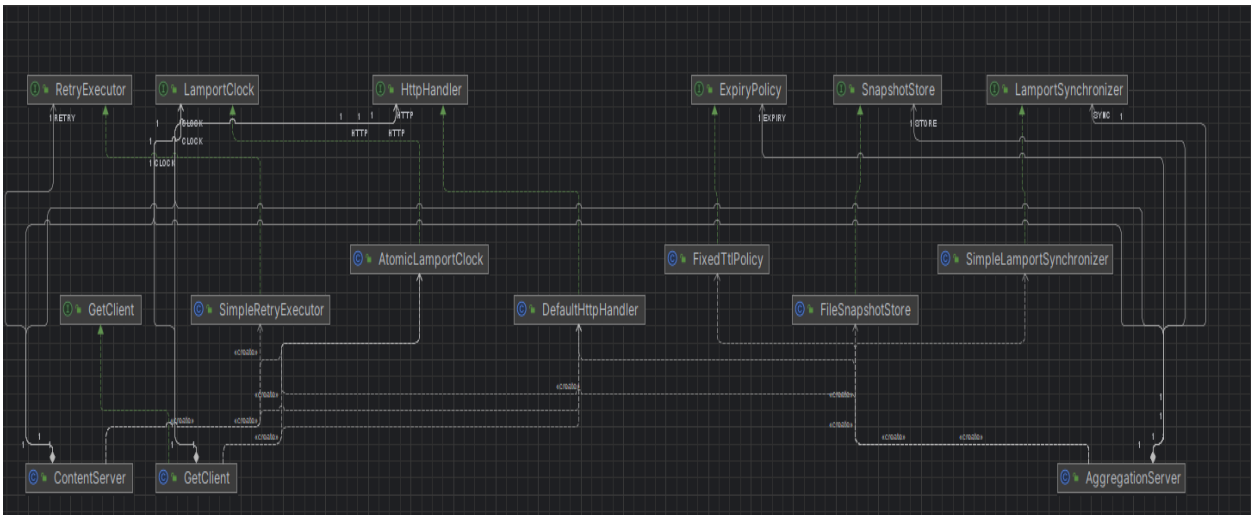
High-Level Architectural Explanation

In my architectural design, I implemented a **distributed weather aggregation system** that maintains **causal consistency** using **Lamport logical clocks**. The system is composed of two main clients — the **ContentServer** and the **GetClient** — which communicate with a central **AggregationServer**. The ContentServer is responsible for sending weather data updates using HTTP PUT requests, while the GetClient retrieves the latest weather information through HTTP GET requests. Both clients attach Lamport metadata (X-Lamport-Clock, X-Lamport-Node) to every request to ensure that all operations follow a logical event order across distributed nodes.

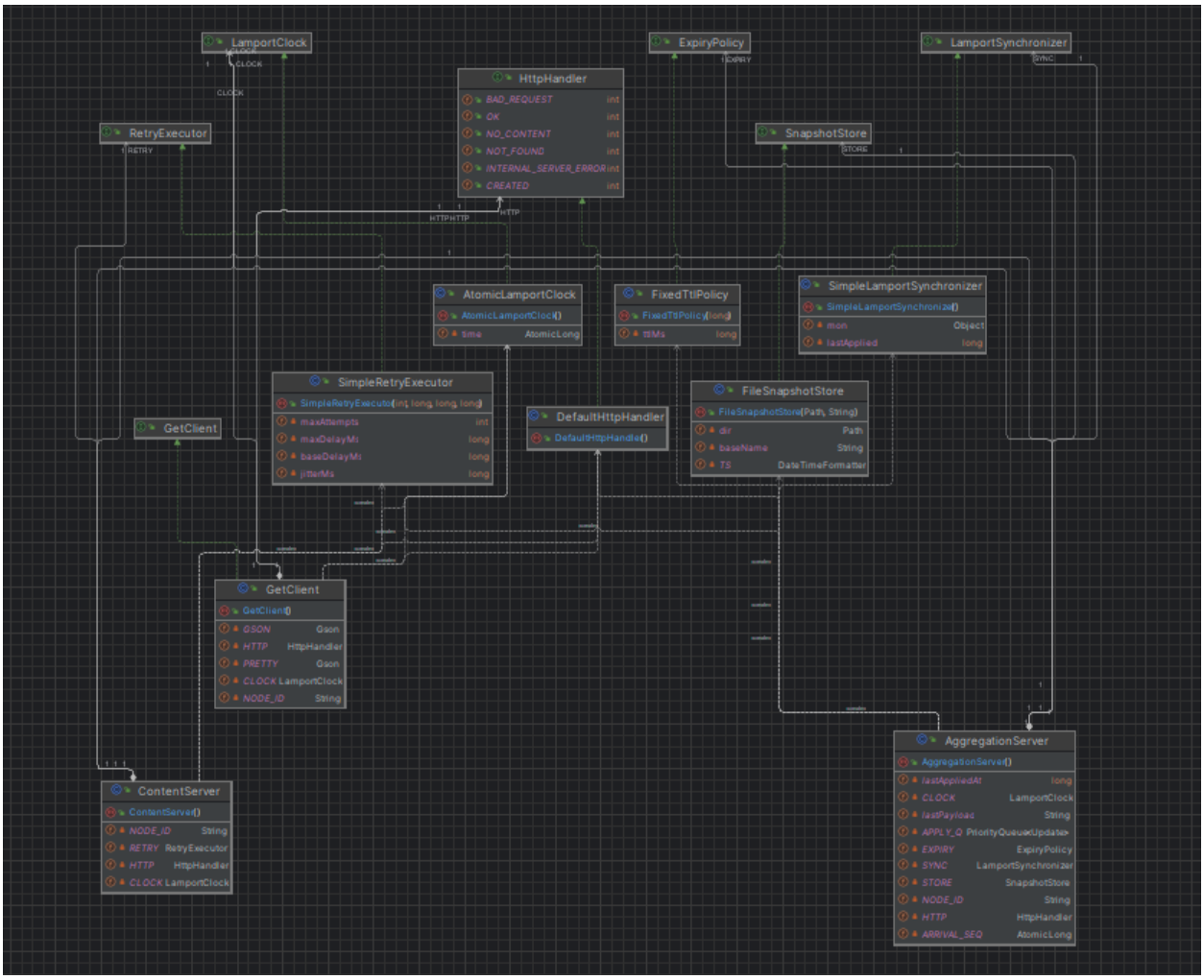
The AggregationServer maintains a **Lamport-ordered apply queue**, where each incoming update is applied sequentially based on its logical timestamp rather than its arrival time. This design guarantees consistency and determinism even when multiple clients send concurrent updates. To support durability, I use a **FileSnapshotStore** to persist each applied state to disk and a **FixedTtlPolicy** to ensure that old data automatically expires after 30 seconds. For synchronization between PUT and GET operations, I implemented a **SimpleLamportSynchronizer**, allowing GET requests to wait until all prior PUT operations with lower Lamport timestamps have been processed.

To handle communication, I designed a **DefaultHttpHandler**, which is responsible for constructing HTTP requests and responses, including automatic Lamport clock updates. On the client side, I added a **SimpleRetryExecutor** to make the PUT process more resilient through exponential backoff and jitter. Together, these components form a lightweight and fault-tolerant architecture that provides reliable state propagation, crash recovery, and causally consistent reads within a distributed environment. My main goal was to achieve **high cohesion** and **loose coupling** across all components.

To handle communication, I designed a **DefaultHttpHandler** responsible for constructing HTTP requests and responses, including automatic Lamport clock updates. On the client side, I added a **SimpleRetryExecutor** to make the PUT process resilient with exponential backoff and jitter. Together, these components form a lightweight, fault-tolerant architecture that provides reliable state propagation, crash recovery, and causally consistent reads in a distributed environment.

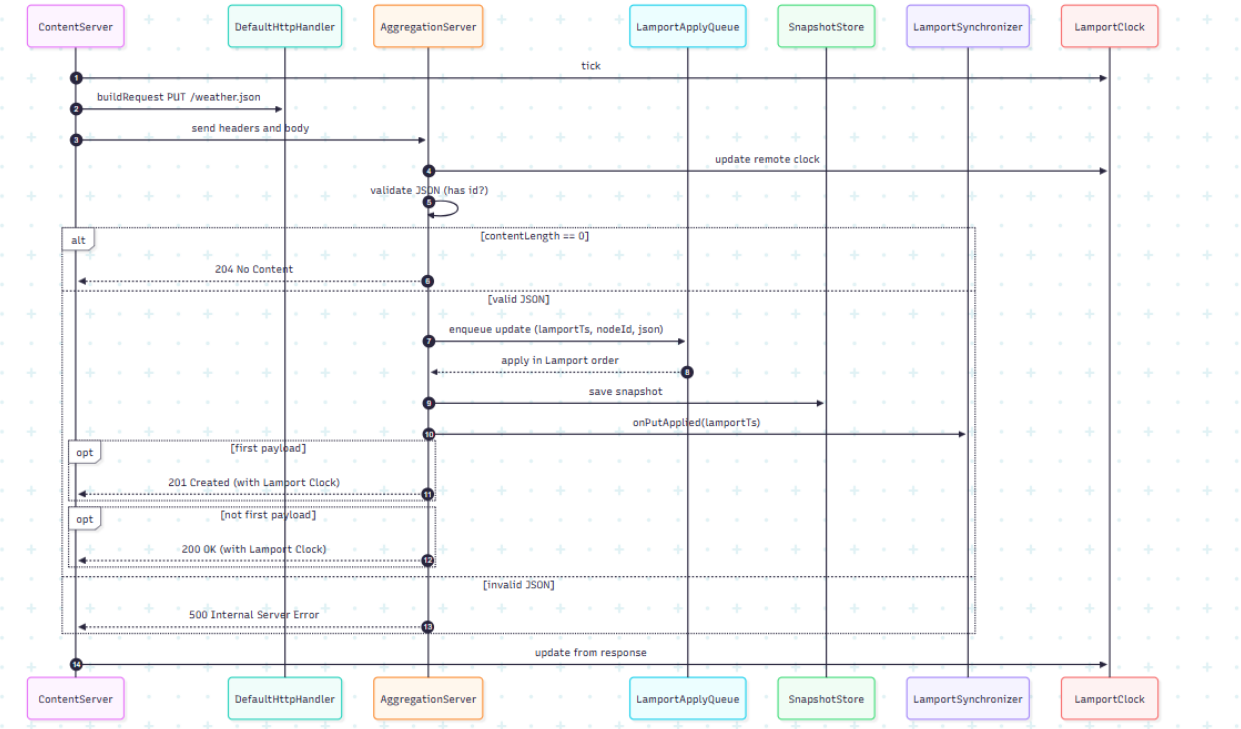


Detailed Architectural Explanation



1. **ContentServer** – I needed a producer that reads a local file, normalizes it to JSON, stamps a Lamport time, and reliably sends a PUT /weather.json to the server with retries.
2. **GetClient (class)** – I wanted a simple consumer that issues GET /weather.json, logs Lamport activity, and pretty-prints whatever the server returns for demo/marking.
3. **HttpHandler (interface)** – I separated raw HTTP wire work from business logic so both clients and the server can build/parse requests/responses the same way.
4. **DefaultHttpHandler** – My concrete, minimal HTTP/1.1 implementation: compose headers, stream bytes, read responses, and centralize server replies with status codes and Lamport headers.
5. **AggregationServer** – This is the authoritative node: it merges Lamport clocks, validates input, applies updates in logical order, serves GETs, enforces TTL, and persists snapshots.
6. **LamportClock (interface)** – I wanted a clean contract for logical time so I could swap implementations if needed.
7. **AtomicLamportClock** – A lock-free, thread-safe Lamport clock using AtomicLong so every component can safely tick() and update().
8. **LamportSynchronizer (interface)** – I needed a tiny abstraction to let GETs wait until all PUTs up to a target Lamport are applied (read-your-writes behavior).
9. **SimpleLamportSynchronizer** – A minimal monitor that tracks lastApplied and blocks until the server catches up; small, testable, and enough for this assignment.
10. **RetryExecutor (interface)** – I abstracted retry logic so any network call (not just ContentServer) could opt into consistent backoff behavior.
11. **SimpleRetryExecutor** – My concrete capped exponential backoff with jitter and Retry-After respect, to harden flaky connections without complicating clients.
12. **ExpiryPolicy (interface)** – I didn't want TTL logic hard-coded in the server; this keeps "freshness" pluggable.
13. **FixedTtlPolicy** – A simple "TTL in ms" policy (30s here) to mark data stale and return 404 once it ages out.
14. **SnapshotStore (interface)** – I abstracted persistence so I can change storage later (file, DB, cloud) without touching server logic.
15. **FileSnapshotStore** – Durable, atomic file snapshots plus a latest.json pointer so I can restore state on restart and show crash recovery.
16. **Lamport-ordered apply queue (Update record + applier thread)** – I needed deterministic, causal application of writes: a single consumer thread drains a priority queue ordered by (lamportTs, nodeId, arrivalSeq).

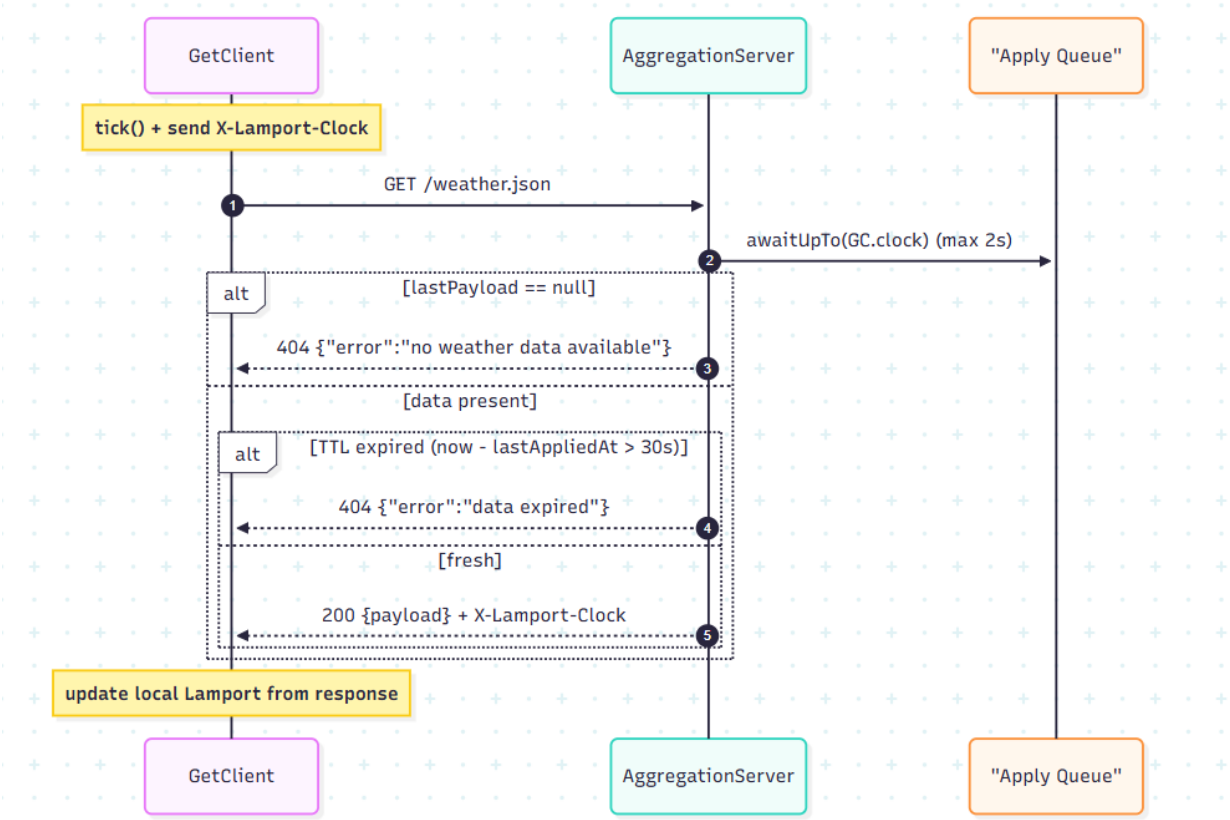
Sequence Diagrams



PUT flow (ContentServer → AggregationServer):

The client reads the file and builds a PUT /weather.json with Lamport headers, then ticks its local clock and sends headers + body. The server updates its Lamport clock from the incoming header, parses the request line, and branches: if Content-Length == 0 it answers **204 No Content**; if the JSON is malformed or missing id it replies **500**. For valid JSON, it enqueues an update into a Lamport-ordered apply queue (ordering by lamport, nodeId, seq), the single applier thread pops it, applies it to lastPayload, persists the snapshot, records lastAppliedAt, and notifies the LamportSynchronizer.onPutApplied(ts). The HTTP status is **201 Created** for the first non-empty payload and **200 OK** for subsequent updates. Finally, the client receives the response, prints

headers, and updates its local Lamport clock from the server's X-Lamport-Clock.



GET flow (GetClient → AggregationServer):

The GET client ticks its Lamport clock, sends GET /weather.json with Lamport headers, and the server first calls awaitUpTo(GC.clock, 2s) on the synchronizer to ensure all PUTs up to that logical time have been applied. Then it branches on state: if there's no payload yet, it returns **404** with {"error":"no weather data available"}; if there