Functional Analysis

This system is a distributed client-server application built using raw Java sockets and JSON messages to aggregate and distribute weather data through a RESTful API model. It consists of three main components: Content Servers, the Aggregation Server, and GET Clients.

1. Content Server

- Reads a weather record from a local text file.
- Converts the data to a valid JSON format.
- Sends an HTTP-like PUT /weather.json request to the Aggregation Server.
- Retries sending data if the server is unavailable (at-least-once delivery).
- Maintains a Lamport clock and includes it in the PUT request headers.
- Expects a status response (201 for new, 200 for update, 204 for empty, 400/500 for errors).

2. Aggregation Server

- Accepts concurrent PUT and GET requests from multiple sources.
- Maintains a persistent JSON store that survives crashes and restarts.
- Stores data only from content servers that have communicated in the last 30 seconds.
- Limits records to the most recent 20 entries and removes outdated ones.
- Handles request ordering using Lamport clocks:
    - Each request includes a clock value.
    - The server updates its own clock: max (local, received) + 1
    - Ensures requests are processed in consistent logical order.
- Returns appropriate HTTP-style status codes for every request.

3. GET Client

- Sends an HTTP-like GET /weather.json request to the Aggregation Server.
- Optionally includes a station ID to filter results (if implemented).
- Parses the received JSON and displays each key-value pair in readable format.
- Also maintains a Lamport clock, updated upon request send/receive.
- Robust to failures (e.g., network disconnects, server not found).

Component View

1. Content Server

- Reads a local weather file (.txt)
- Converts the input into valid JSON format
- Sends a PUT /weather.json request (via socket) to the Aggregation Server
- Includes a Lamport timestamp in the request header
- Retries up to 3 times on failure (to ensure fault tolerance)
- Only handles one station ID per server, but can send one update per execution.

2. Aggregation Server

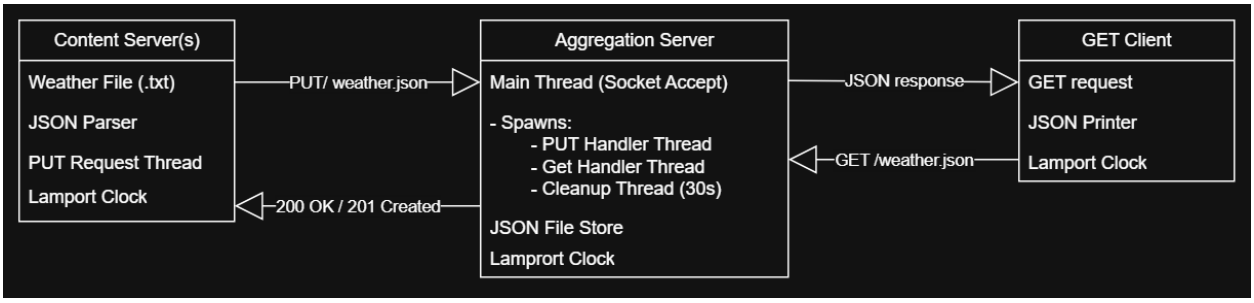
- Listens on port 9090 and accepts concurrent **PUT** and **GET** connections.
- Spawns a new thread to handle each incoming request safely.
- Maintains a Lamport Clock to order concurrent PUT/GET operations.
- Stores weather data **in memory** and persists to a JSON file (crash-safe).
- Periodically runs a **cleanup thread** to remove:
    - Old records (>30 seconds since last update from any Content Server).
    - Excess entries (keeps only the 20 most recent).
- Responds with appropriate HTTP-like status codes:
    - 200 OK (success),
    - 201 Created (new record),
    - 500 Internal Server Error (failures).
    - (400 Bad Request partially supported, but optional).

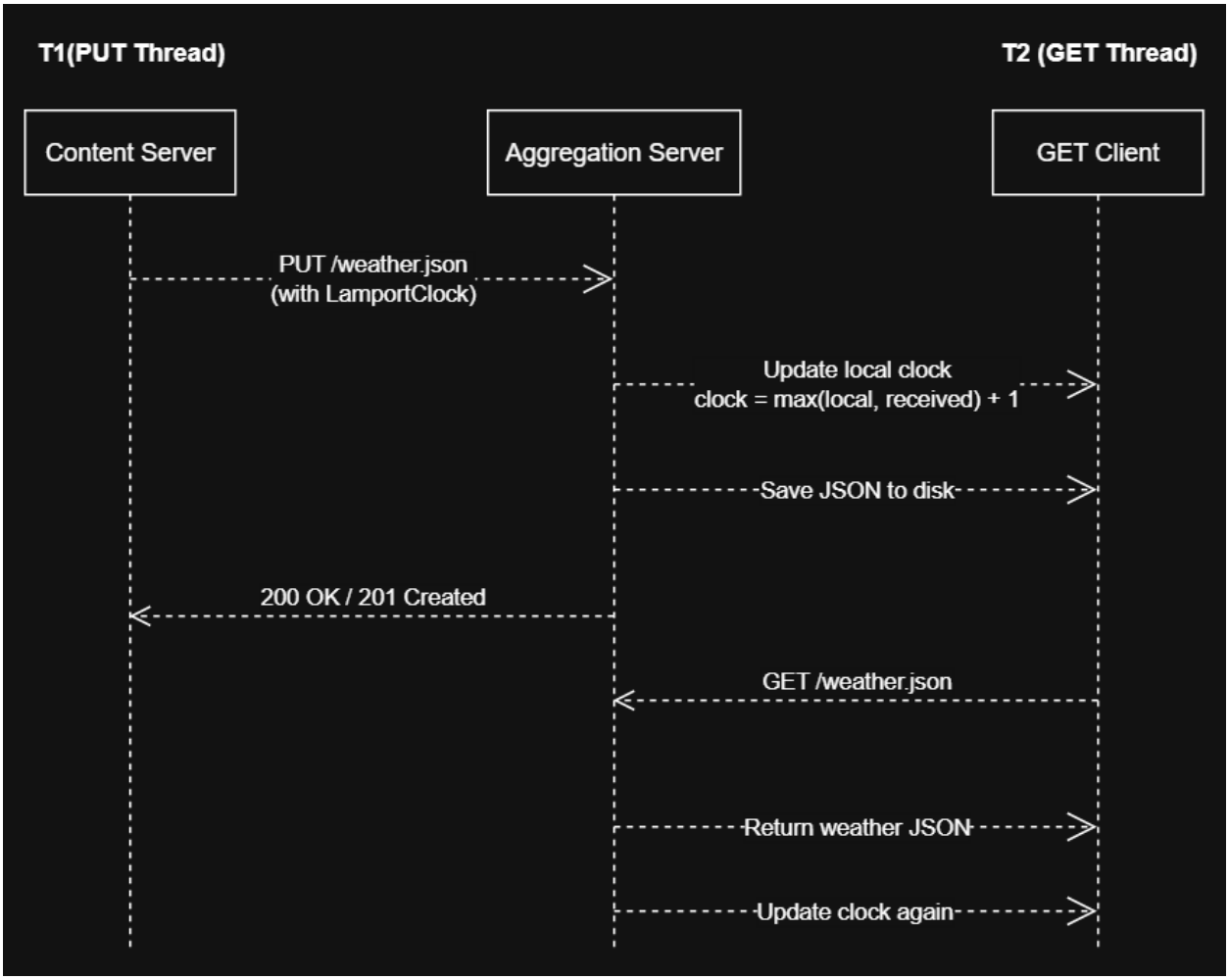
3. GET Client

- Sends a GET /weather.json request to the Aggregation Server
- Parses the received JSON and displays it in a readable key-value format
- Updates its own Lamport clock using server response.
- Designed to handle unexpected failures gracefully

Component Interactions Summary

- **Content Server → Aggregation Server**
    - PUT request
    - Includes JSON body and Lamport timestamp
- **GET Client → Aggregation Server**
    - GET request
    - Receives JSON weather data
- **Aggregation Server**
    - Stores + updates file
    - Manages expiry & clock ordering


Content Server(s)
Weather File (.txt)
JSON Parser
PUT Request Thread
Lamport Clock
PUT/ weather.json
200 OK / 201 Created
Aggregation Server
Main Thread (Socket Accept)
- Spawns:
- PUT Handler Thread
- Get Handler Thread
- Cleanup Thread (30s)
JSON File Store
Lamprort Clock
JSON response
GET /weather.json
GET Client
GET request
JSON Printer
Lamport Clock


T1(PUT Thread)
T2 (GET Thread)
Content Server
Aggregation Server
GET Client
PUT /weather.json
(with LamportClock)
Update local clock
clock = max(local, received) + 1
Save JSON to disk
200 OK / 201 Created
GET /weather.json
Return weather JSON
Update clock again

Multithreaded Interactions & Thread Safety

The Aggregation Server is designed to handle multiple simultaneous clients (GET or PUT), each using its own dedicated thread. This architecture ensures that the server remains responsive under load and handles concurrent operations correctly.

Thread Handling

- A new thread is spawned or reused for every incoming socket connection.
- Each thread independently handles either a PUT or GET request.
- The server uses Java thread pools (e.g., ExecutorService) or basic Thread class for simplicity.

Shared Resource Protection

To ensure safety when threads access shared resources (e.g., the JSON file, Lamport clock, server memory), the following are enforced:

Shared Resource	Protection Mechanism
Lamport Clock	synchronized method or AtomicInteger for atomic updates
JSON Storage File	Write to temp file → rename to main file (atomic replace)
In-memory Data (e.g., server state, content map)	Access via ConcurrentHashMap or synchronized block
Log files or status messages	Buffered writer with synchronized access (if logging to file)

Deadlock Prevention

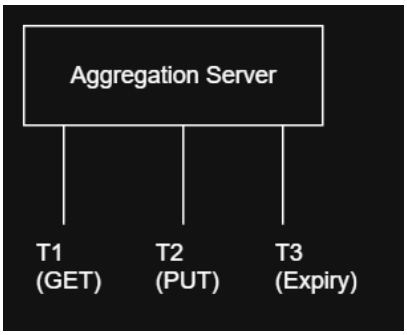
- No nested locks are used
- Minimal locking scope (only around critical sections)
- Shared resources are locked in a fixed order (if needed)

Cleanup Thread

- A timer thread or ScheduledExecutorService runs every 5–10 seconds
- Checks for inactive content servers (last update > 30s)
- Removes stale weather entries from persistent store

Replica Reasoning

- At least two content server replicas to simulate fault tolerance
- If one goes down, the Aggregation Server continues to serve data from the remaining replicas
- Allow system remain available and resilient



Testing

1.Basic Functionality Tests

PUT Request Test (Single Content Server)

- mvn exec:java "-Dexec.mainClass=au.edu.adelaide.ds.assignment2.ContentServer" "-Dexec.args=replica1 weather1.txt"
- Expect: Aggregation Server accepts PUT request with weather data, returns 200 OK or 201 Created.
- Result:

GET Request Test (Client)

- mvn exec:java "-Dexec.mainClass=au.edu.adelaide.ds.assignment2.Client"
- Expect: Client retrieves and display key-value weather data.
- Result:

2.Multiple Replicas Test

Concurrent PUTs from Two Content Servers

- mvn exec:java "-Dexec.mainClass=au.edu.adelaide.ds.assignment2.ContentServer" "-Dexec.args=replica1 weather1.txt"
- mvn exec:java "-Dexec.mainClass=au.edu.adelaide.ds.assignment2.ContentServer" "-Dexec.args=replica2 weather2.txt"
- Expect: Aggregation Server accepts updates from both replica1 (e.g., Adelaide) and replica2 (e.g., Sydney)
- Result:

3.Fault Tolerance & Recovery

Simulate Fault by Terminating One Replica

- Steps:
  1. Start Aggregation Server
  2. Start replica1 and replica2
  3. Confirm both appear in client
  4. Manually terminate replica1
  5. Run client again
- Expect: replica1 data become stale and removed, replica2 continues sending updates
- Result:

4. Expiry & Cleanup Verification

Test 30s Expiry of Records

- Setup:
  1. Let content servers run, then stop all of them
  2. Wait for 30 seconds
  3. Run GET client
- Expect: Aggregation server should remove outdated records (>30s)
- Result:

5. Lamport Clock Consistency

Lamport Clock Update & Ordering

- Observation from Client Logs:
  - Each record had increasing Lamport timestamps
  - Server consistently followed: Lamport Clock = max(local, received) +1
- Result:

6. Error Handling/ Retry Test

Simulate PUT Retry

- Temporarily disabled the Aggregation Server, then ran Content Server
- Expect: Content Server retries on failure (up to 3 times with delay)
- Result: