DISTRIBUTED SYSTEM

ASSIGNMENT 2: DESIGN

Name: Muhammad Harith Mohd Lukman

ID: A1790387

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 Content Servers.
- Persists weather data to a JSON file (survives crashes and restarts).
- Removes records older than 30 seconds (expiry) and optionally limits to the 20 most recent records.
- Maintains Lamport clock ordering:
    - Updates local clock as max(local, received) + 1
    - Ensures requests are processed in consistent logical order.
- Responds with appropriate HTTP-style status codes.

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 input into valid JSON format.
- Sends a PUT /weather.json request (via socket) to the Aggregation Server.
- Includes Lamport timestamp in request header.
- Retries up to 3 times on failure (ensures at-least-once delivery).
- Each Content Server handles one station file, sending updates continuously.

2. Aggregation Server

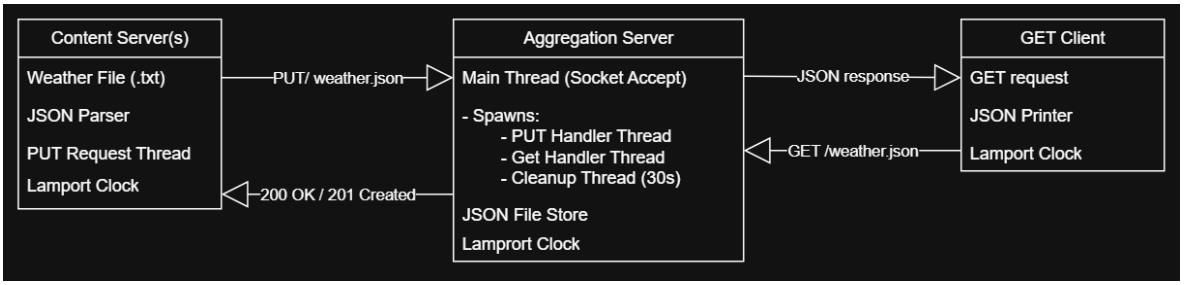
- Listens on port 4567 and accepts concurrent PUT and GET requests.
- Spawns a new thread for each incoming connection.
- Maintains a Lamport Clock for request ordering.
- Stores weather data in memory and persists to weather_data.json (survives restarts).
- Periodic cleanup thread removes:
    - Records older than 30s (stale data).
    - Excess entries (keeps max 20 most recent).
- Responds with HTTP-style codes:
    - 200 OK (update success)
    - 201 Created (new record)
    - 204 No Content (no valid records)
    - 400 Bad Request (invalid payload)
    - 500 Internal Server Error (simulated failure)

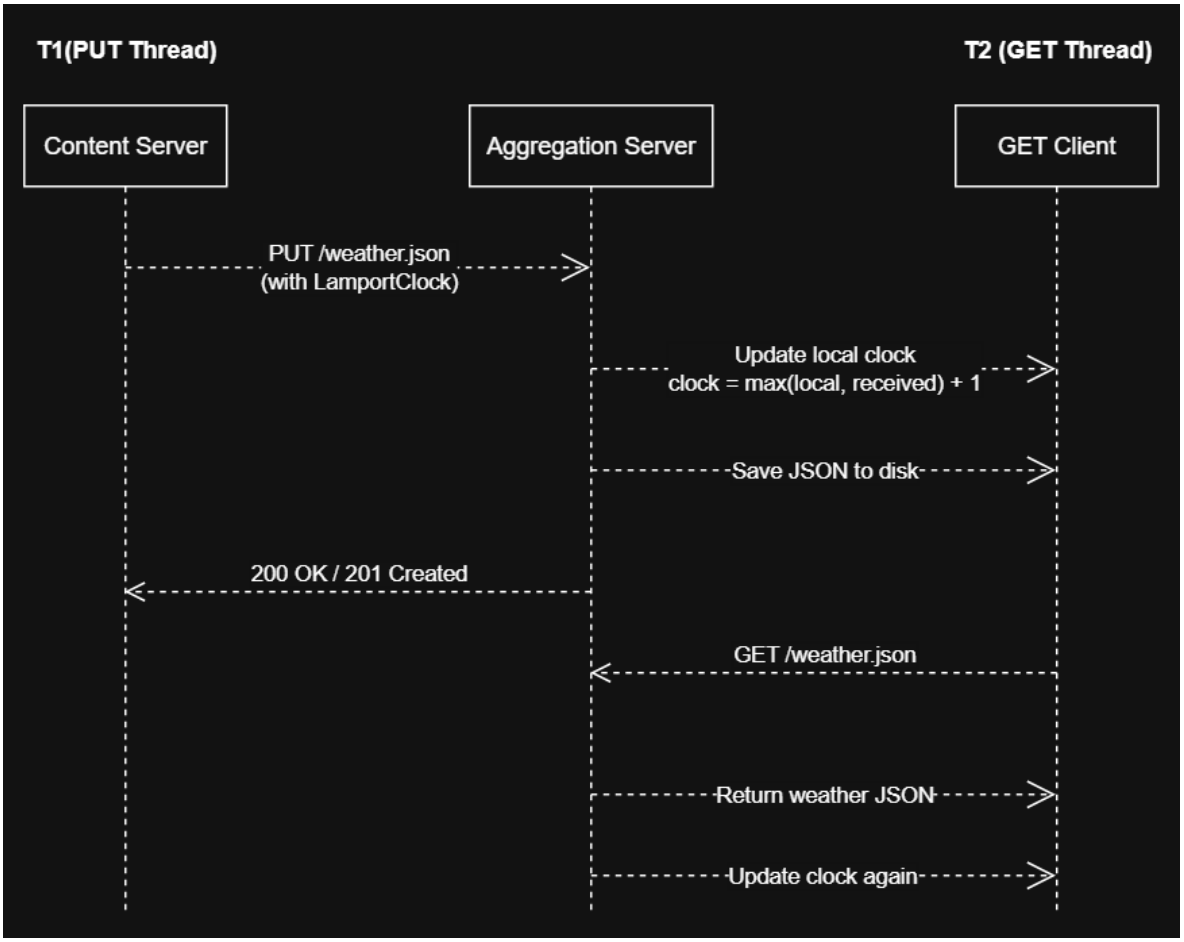
3. GET Client

- Sends a GET /weather.json request.
- Receives JSON response and displays in key-value format.
- Updates Lamport clock using server response.
- Handles unexpected failures gracefully.

Component Interaction Summary

- Content Server → Aggregation Server
    - Sends PUT request with JSON body + Lamport timestamp.
    - Server updates Lamport clock and stores data.
- GET Client → Aggregation Server
    - Sends GET request.
    - Receives JSON data + server Lamport ordering.
- Aggregation Server
    - Persists data (weather_data.json).
    - Runs cleanup (expiry + max 20).
    - Maintains consistent ordering via Lamport clocks.





Multithreaded Interactions & Thread Safety

The Aggregation Server handles multiple simultaneous clients (PUT or GET) using separate threads. This ensures responsiveness and correct ordering under load.

Thread Handling

- A new thread is spawned for every incoming socket connection (new Thread(ClientHandler).start()).
- Each thread independently handles either a PUT or a GET request.
- Lamport clock updates are synchronized to ensure consistency.

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 methods for atomic updates
JSON Storage File	Write to temp file → rename to weather_data.json (atomic replace)
In-memory Data (e.g., server state, content map)	Collections.synchronizedList wrapper
Log files or status messages	Logger with synchronized output

Deadlock Prevention

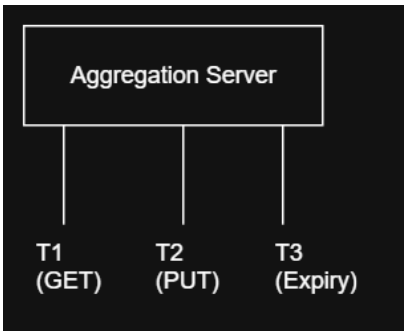
- No nested locks are used.
- Minimal locking scope (only around Lamport clock & JSON writes).
- Shared resources locked in a consistent order when needed.
- Lamport updates are atomic, preventing blocking chains.

Cleanup Thread

- A background cleanup thread runs every 30 seconds.
- Removes stale records (>30s since last update).
- Trims excess records, keeping max 20 entries.

Replica Reasoning

- At least 2 Content Server replicas are recommended for resilience.
- If one replica fails, its data expires naturally after 30s.
- Aggregation Server continues serving live data from remaining replicas.
- Ensures high availability and robustness.



This diagram illustrates how the Aggregation Server internally spawns three types of threads to handle concurrent operations:

- **T1 (GET)**: Handles client GET requests.
- **T2 (PUT)**: Handles PUT requests from Content Servers.
- **T3 (Expiry)**: Background cleanup thread that periodically removes stale records.
  This ensures responsiveness, concurrency, and automatic expiry cleanup.

Testing

1. Basic Functionality

1.1 PUT Request (Single Content Server)

- **Setup**
  - Start Aggregation Server
  - Start one Content Server (replica1 weather1.txt)
- **Action**
  - Content Server sends weather record with Lamport timestamp
- **Expected Result**
  - Aggregation Server accepts request
  - Responds **201 Created** on first record
  - Responds **200 OK** on subsequent updates (same station)

1.2 GET Request (Client)

- **Setup**
  - Aggregation Server running
  - At least one Content Server running and pushing updates
- **Action**
  - Client sends GET /weather.json request
- **Expected Result**
  - Server returns latest weather records in JSON
  - Client displays data as readable key-value pairs

2. Multiple Replicas

2.1 Concurrent PUTs

- **Setup**
  - Start Aggregation Server
  - Start replica1 weather1.txt and replica2 weather2.txt
- **Action**
  - Both Content Servers push updates simultaneously
- **Expected Result**
  - Aggregation Server stores and merges both records
  - Client retrieves combined dataset

2.2 Fault Tolerance

- **Setup**
  - Start Aggregation Server, replica1, and replica2
- **Action**
  - Stop replica1 manually (Ctrl+C)

    - Continue running replica2
- **Expected Result**
    - After 30s expiry, replica1's record removed
    - Client only shows active data from replica2

3. Expiry & Cleanup

3.1 30s Expiry

- **Setup**
    - Start Aggregation Server and one Content Server
- **Action**
    - Stop Content Server after updates
    - Wait 30 seconds
- **Expected Result**
    - Aggregation Server removes expired records
    - Client request returns **204 No Content**

4. Error Handling

4.1 Bad Request (400)

- **Setup**
    - Modify weather1.txt to invalid format ({"badField":"oops"})
    - Start Aggregation Server and Content Server
- **Action**
    - Content Server attempts to send invalid JSON
- **Expected Result**
    - Aggregation Server rejects request with **400 Bad Request**

4.2 Internal Server Error (500)

- **Setup**
    - Uncomment simulated failure line in AggregationServer.java
    - Start Aggregation Server
- **Action**
    - Content Server sends PUT request
- **Expected Result**
    - Aggregation Server responds **500 Internal Server Error**

5. Persistence

5.1 Restart & Recovery

- **Setup**

    - Start Aggregation Server + Content Server
    - Ensure data is written to weather_data.json
- **Action**
    - Stop Aggregation Server (Ctrl+C)
    - Restart Aggregation Server
- **Expected Result**
    - Records are restored from weather_data.json
    - Client can still retrieve data after restart

6. Lamport Clock Consistency

6.1 Logical Ordering

- **Setup**
    - Run Aggregation Server and multiple Content Servers
- **Action**
    - Observe Lamport timestamps in logs during concurrent PUTs
- **Expected Result**
    - Each record has increasing Lamport values
    - Server updates local clock as max(local, received) + 1