Assignment 2 Design Sketch

1. System Overview

This system adopts a client-server architecture based on RESTful API, with its core functions being the aggregation and distribution of weather data in JSON format. Meanwhile, it ensures the consistency of interactions among multiple clients and multiple servers through Lamport clocks, addressing issues such as concurrent request ordering, data expiration cleanup, and crash recovery. The system is generally divided into three core components: **Client (GET Client)**, **Content Server**, and **Aggregation Server**. All components implement the interaction of HTTP GET/PUT requests via Sockets, and the entire codebase is developed using Java.

2. Functional Analysis

There are several **shared design principles** that all components must adhere to:

- **Consistency**: Ensure the order of requests (e.g., PUT → GET → PUT must be executed sequentially, and GET should return the intermediate updated result) through Lamport clocks.
- **Fault Tolerance**: All components must handle failures such as network outages, server crashes, and data format errors, with predictable behaviors.
- **Efficiency**: The expired data cleanup mechanism must be efficient, and multi-threaded request processing should be free from performance bottlenecks.

The following table presents an analysis of the core functions and required constraints for each component:

Component	Core Functions	Key Constraints
Aggregation Server	1. Receive PUT requests from Content Servers, validate and store weather data in JSON format 2. Respond to GET requests from GET Clients and return formatted weather data 3. 30-second expiration cleanup: Remove data from Content Servers that have not communicated for more than 30 seconds 4. Crash recovery: Ensure data files can be recovered after a crash to avoid corruption caused by interrupted updates 5. Lamport clock management: Maintain local clock and sort concurrent PUT/GET requests based on the clock	1. Support thread-safe request processing without race conditions or deadlocks 2. Data storage must be persistent by JSON files 3. Return correct HTTP status codes (200/201/204/400/500)
Content Server	1. Read local text files and convert them into standard JSON format 2. Send PUT requests to the Aggregation Server to upload weather data 3. Verify server responses (e.g., 201/200) to confirm successful data upload 4. Maintain local Lamport clock and carry clock information when sending requests	1. Handle errors such as server inaccessibility and support retry mechanisms 2. Reject upload of weather data without an ID
GET Client	1. Parse command-line arguments (server address, port, optional station ID) 2. Send GET requests to the Aggregation Server to obtain weather data 3. Format JSON data and display it line by line in "attribute-value" pairs 4. Maintain local Lamport clock and update the clock when receiving responses	1. Handle exceptions such as network timeouts and server errors, and support retries 2. Be compatible with multiple URL formats (e.g., `http://server:port`, `server:port`)

3 Component View

3.1 Internal Module Breakdown of Components

(1) Aggregation Server

Module	Function Description
Socket Communication Module	Listens on a specified port (default: 4567, port modification supported via command-line arguments) and processes requests from clients/Content Servers.
Lamport Clock Module	Maintains the server's local Lamport clock. When receiving a request, it updates the clock (takes the maximum value of the local clock and the request clock, then adds 1) and carries the current clock in responses.
Request Processing Module	Parses HTTP requests (distinguishes between GET and PUT), invokes the corresponding business logic, and returns specified HTTP status codes (e.g., 201 Created, 400 Bad Request).
Data Storage Module	Persistently stores weather data in JSON files and Validates the validity of data formats
Expiration Cleanup Module	Maintains the last communication time of Content Servers, and periodically checks and deletes timed-out data.

(2) Content Server

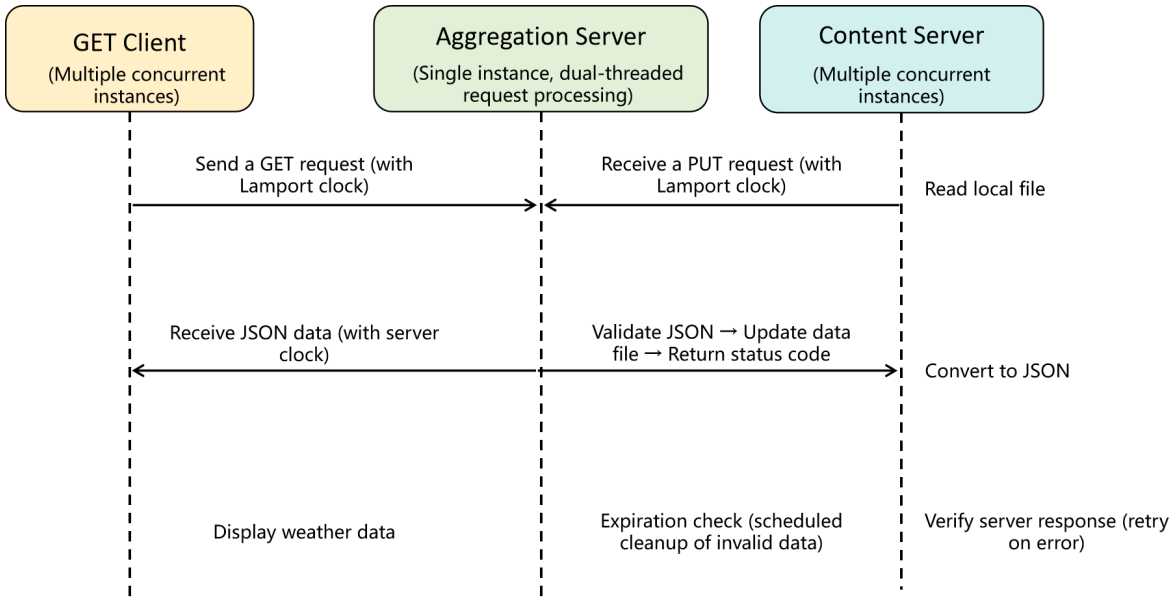
Module	Function Description
Command Line Parsing Module	Parses input parameters (server address/port, local text file path) and validates the legality of parameters (e.g., file existence).
JSON File Module	Reads local JSON files and generates JSON data that can be sent.
Socket Client Module	Sends PUT requests to the Aggregation Server (carrying JSON data and Lamport clock), receives responses, and handles errors (e.g., retries).
Lamport Clock Module	Maintains the local clock: increments the clock before sending a request, and updates it after receiving a response (takes the maximum value of the local clock and the server clock, then adds 1).

(3) GET Client

Module	Function Description
Command Line Parsing Module	Parses the server address/port (supports multiple URL formats) and optional station ID, and validates the parameter format.
Socket Client Module	Sends GET requests (carrying Lamport clock and optional station ID), and handles errors such as network timeouts and server inaccessibility (supports retries).
JSON Formatting Module	Receives JSON data returned by the server and displays it line by line in "attribute-value" pairs.
Lamport Clock Module	Maintains the local clock: increments the clock before sending a request and updates it after receiving a response.

3.2 Component Interaction Relationships

The following diagram illustrates the interaction relationships between various components:



As shown in the figure, the GET Client (with multiple concurrent instances) sends a GET request carrying a Lamport clock to the single-instance, dual-threaded Aggregation Server, which then returns JSON data with a server clock to the GET Client for weather data display. Meanwhile, the Content Server (with multiple concurrent instances) reads a local file, converts it to JSON, and sends a PUT request with a Lamport clock to the Aggregation Server; upon receiving this, the Aggregation Server validates the JSON, updates the data file, and returns a status code, while the Content Server verifies the server response and retries on error. Additionally, the Aggregation Server performs expiration checks to schedule cleanup of invalid data.

4. Concurrency & Thread Safety

It is evident that in this project, the competition arising from reading and writing the JSON file maintained by the aggregation server constitutes the primary bottleneck of the system, and this is a resource-intensive task. Therefore, we intend to implement the solution based on Java's non-blocking I/O capabilities. To ensure that the aggregation server responds to requests in sequence according to Lamport Clock timestamps and guarantees thread safety (thus preventing deadlocks), a concurrent

request mechanism featuring **dual threads + a first-in-first-out (FIFO) buffer queue** is designed here, adhering to the producer-consumer design pattern.

Specifically, when the aggregation server starts up, it creates two threads: one for handling connections and the other for sending responses. When the aggregation server receives a request from a GET Client or Content Server, the connection-handling thread (acting as the "producer") first distinguishes the type of the request, creates a corresponding request handler, and stores this handler in the buffer queue. These handlers hold the socket handle of the connection, enabling them to send responses to the client. Meanwhile, the response-sending thread (acting as the "consumer") continuously checks the buffer queue, retrieves the pending response handlers, and invokes the corresponding functions to send responses.

This design delivers three key advantages: First, the aggregation server can simultaneously accept a large volume of access requests, ensuring high concurrency. Second, since read and write operations on the weather data file are exclusively handled by a single thread, issues like resource contention and data inconsistency are eliminated, which significantly simplifies the program logic. Third, it provides an elegant mechanism to determine whether a Content Server has been disconnected for longer than a specified duration—specifically, the aggregation server maintains a Map where the keys are Content Server IDs and the values are request handlers, with the Map automatically sorted based on the Lamport Clock of the handlers. When a Content Server sends multiple requests, only the modification timestamp and carried data of its corresponding handler are updated. This allows the system to check for expiration simply by comparing the aggregation server's current time with the time difference since the request handler was last modified. Additionally, for the same Content Server, if it sends requests frequently, only the latest one will be processed, effectively saving the aggregation server's resources.

5. How many server replicas need and why

Java's non-blocking Socket IO (based on NIO's `SocketChannel` and `Selector`) theoretically supports more than 100,000 concurrent connections. Therefore, for the design scheme proposed in this paper, only one aggregation server is required, while for Content Servers, at least one is needed, with no upper limit on the number.

6. Testing Strategy

The testing process, structured to follow the logic of unit testing preceding integration testing and local testing preceding remote testing, consists of three distinct phases: local unit testing, local integration testing, and remote integration testing.

6.1 Local Unit Testing

Within a local environment, individual units—such as functions and classes—across the client, content server, and aggregation server undergo rigorous testing. This phase focuses on validating the functionality of each isolated unit to ensure code reliability and quality. For tests involving network interactions, a mock approach is employed, where local components stand in for corresponding servers or clients to facilitate testing.

6.2 Local Integration Testing

Building on units that have passed unit testing, integration testing is conducted in a local setting. Its primary goal is to verify the collaboration and interaction between different units within the client, content server, and aggregation server, identifying potential interface mismatches or dependency-related issues. Similar to unit testing, network-dependent scenarios utilize the mock method, replacing

external servers or clients with local components.

6.3 Remote Integration Testing

In this phase, system components initiate network requests in a predefined sequence to test the entire system as an integrated entity. The objective is to validate cross-component collaboration and interactions while simulating real-world operational conditions. This includes assessing communication flows, data transmission integrity, and error-handling mechanisms across the system.