Weather Aggregation System - Final Submission

Introduction

The Weather Aggregation System is designed to collect weather data from multiple content servers and provide aggregated data to clients upon request. The system comprises several components, including an Aggregation Server, Content Server, GET Client, Data Store, and utilizes Lamport's logical clock for synchronization.

This document explains the design choices made during the development of the system, provides an overview of the code structure, and discusses the changes made throughout the project.

Design Choices

1. Modular Architecture

I implemented the system using a modular architecture, dividing the functionality into separate classes and components. This approach enhances maintainability, scalability, and testability.

AggregationServer.java: Handles client connections and delegates request processing to ClientHandler.
ClientHandler.java: Processes individual client requests, handling both PUT and GET methods.
ContentServer.java: Reads weather data from files and sends PUT requests to the Aggregation Server.
GETClient.java: Sends GET requests to the Aggregation Server to retrieve weather data.
DataStore.java: Manages the storage of weather data with thread-safe operations.
LamportClock.java: Implements Lamport's logical clock for synchronization.
SimpleJsonParser.java: Custom JSON parser for parsing and generating JSON data.
HttpUtils.java: Provides utility methods for HTTP operations (e.g., sending responses).

2. Thread Safety and Concurrency

Given that the Aggregation Server needs to handle multiple concurrent connections from content servers and clients, I implemented thread-safe mechanisms.

ExecutorService: Used in AggregationServer to manage a pool of threads for handling client connections.
ReadWriteLock: Used in DataStore to synchronize access to the weather data, allowing concurrent reads but synchronized writes.
Synchronized Methods: In LamportClock, methods are synchronized to ensure correct clock updates in a multithreaded environment.

3. Lamport's Logical Clock
To handle synchronization across distributed components, I implemented Lamport's logical clock.

Clock Updates: In ClientHandler, the clock is updated upon receiving data from content servers.
Clock Integration: Both the Content Server and GET Client utilize the Lamport Clock to maintain consistent event ordering.
4. HTTP Protocol Handling
I implemented a simple HTTP-like protocol for communication between clients and servers.

Request Parsing: In ClientHandler, the request line and headers are parsed manually.
Response Formatting: Responses are constructed according to the HTTP/1.1 standard.
Error Handling: Appropriate HTTP status codes are used to indicate success or errors (e.g., 200 OK, 201 Created, 400 Bad Request).
5. Data Storage and Expiration
Persistent Storage: Weather data is stored in JSON format in a file (data/weather_data.json), allowing data persistence across server restarts.
Data Expiration: Implemented in DataStore, data that hasn't been updated within 30 seconds is automatically expired. This ensures that outdated data does not persist indefinitely.
Scheduled Executor: Used to periodically check for and expire outdated data.
6. Testing and Test Isolation
JUnit 5: Used for writing unit tests for each component.
Mockito: Employed for mocking dependencies and isolating tests.
@TempDir Annotation: Used in tests to create temporary directories for test data files, ensuring tests do not interfere with each other and do not depend on specific directory structures.
7. Custom JSON Parsing Implementation
To eliminate reliance on external libraries for JSON parsing and to fulfill the bonus requirement, I implemented a custom JSON parser.

SimpleJsonParser.java: A custom JSON parser that parses JSON strings into Java Map and List objects and generates JSON strings from Java objects. It handles the specific JSON structures used in the application.
Removal of External Dependencies: Removed the Jackson library dependencies from the project, making the application independent of external JSON parsing libraries.
Changes Made During Development
Throughout the development process, I made several changes to improve the system's robustness, maintainability, and functionality.

1. Improved Error Handling
Detailed Error Messages: Enhanced error messages in ClientHandler to provide more specific feedback to clients.
Exception Logging: Added logging for exceptions to facilitate debugging.
2. Proper Resource Management
ExecutorService Shutdown: Ensured that the ExecutorService in AggregationServer is properly shut down to release resources.
Socket Closure: Added finally blocks to ensure sockets are closed after operations, preventing resource leaks.
3. Test Enhancements
Use of @TempDir: Modified tests to use JUnit's @TempDir for temporary directories, resolving issues with writing test data files to non-existent directories.
Additional Test Cases: Added tests for concurrent PUT and GET requests, invalid data handling, and error scenarios.
4. Code Refactoring
Utility Methods: Moved common HTTP response logic into HttpUtils to avoid code duplication.
Code Organization: Improved code readability by organizing methods logically and adding comments.
5. Data Parsing Enhancements
Flexible Data Parsing: Modified ContentServer to handle additional keys and data types, enhancing flexibility in the data files.
6. Custom JSON Parsing Implementation
Implementation of SimpleJsonParser: Replaced the usage of Jackson's ObjectMapper with a custom JSON parser (SimpleJsonParser) in all components, including ClientHandler, ContentServer, DataStore, and GETClient.
Dependency Removal: Removed the Jackson library dependencies from the project (pom.xml), reducing the application's reliance on external libraries.
JSON Parsing Logic: The SimpleJsonParser handles parsing JSON strings to Java objects and generating JSON strings from Java objects, tailored to the specific needs of the application.
Conclusion
The Weather Aggregation System is a robust application capable of aggregating weather data from multiple sources and providing it to clients. The design choices made prioritize modularity, thread safety, and scalability. By implementing thorough testing and proper resource management, the system is reliable and maintainable.

By implementing my own JSON parser (SimpleJsonParser), the system no longer relies on external libraries for JSON parsing, improving the application's independence and fulfilling the bonus requirement. This custom parser handles the specific JSON structures used in the application, ensuring efficient and reliable JSON handling while allowing focus on other aspects of the system.