1. **Overview and Components**
    a. Aggregation Server
    Central component that handles weather data aggregation, processes client requests, and manages content server updates
    - Store and manage weather data.
    - Ensure consistency using Lamport clocks.
    - Handle multi-threaded interactions safely.
    - Expire outdated or stale weather data.

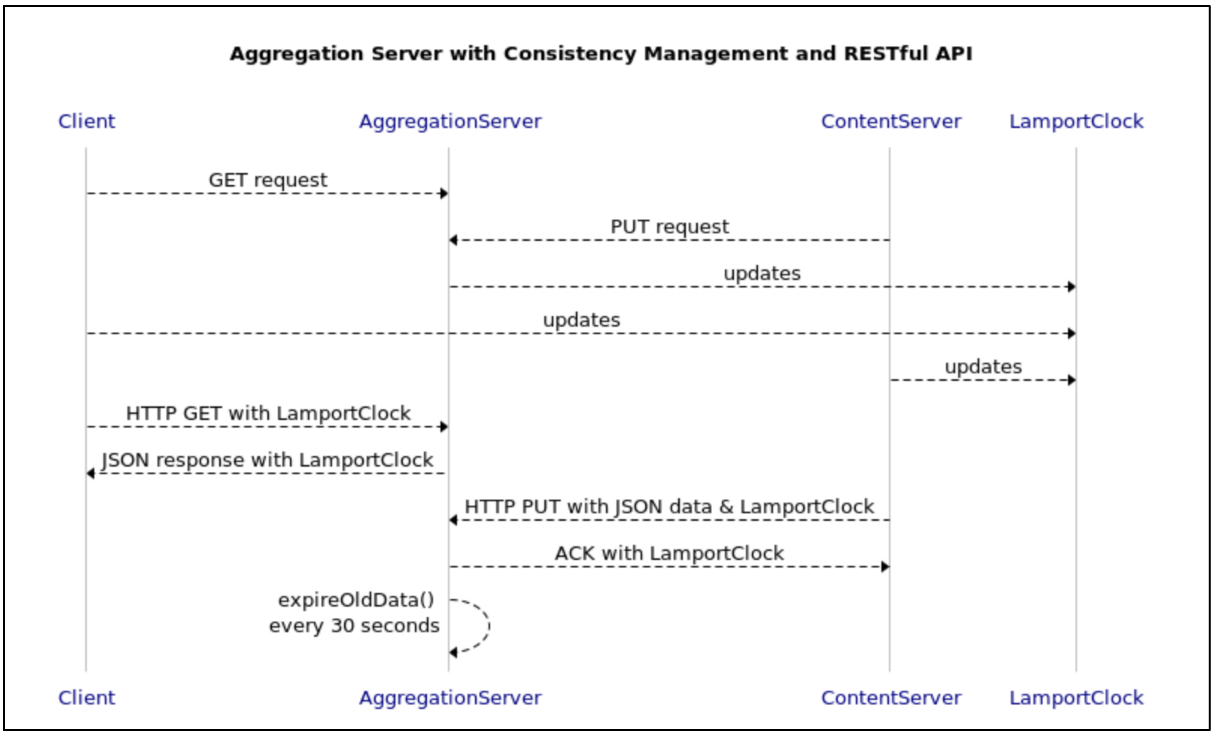
    b. Client
    Requests weather data from the aggregation server
    - Send HTTP GET requests.
    - Handle response data, including Lamport clock timestamp validation.
    - Manage failures and retries.

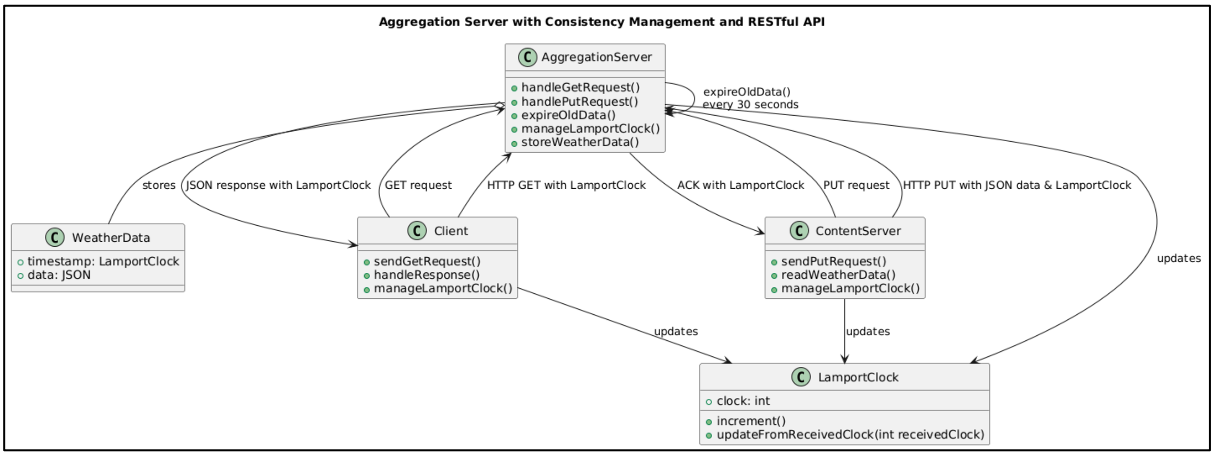
    c. Content Server
    Supplies new weather data to the aggregation server
    - Send HTTP PUT requests with updated weather data.
    - Ensure data integrity and order using Lamport clocks.
    - Maintain communication with the aggregation server to prevent data expiration.

2. **Data Flow and Interactions**
    a. Client Request Flow
    - The client sends an HTTP GET request to the aggregation server.
    - The aggregation server checks its stored weather data.
    - The server responds with the latest weather data, including the associated Lamport clock timestamp.
    - The client validates the timestamp and displays the data.

    b. Content Server Update Flow
    - The content server sends an HTTP PUT request with updated weather data.
    - The aggregation server serializes the incoming PUT requests based on the Lamport clock timestamps.
    - The server updates its stored data, replacing the old information.
    - The server acknowledges the content server with the updated status and timestamp.

    c. Data Expiry and Consistency
    - The aggregation server continuously checks for inactive content servers. If a content server has not communicated within the last 30 seconds, its data is expired and removed.
    - The aggregation server ensures that only the most recent 20 updates are retained, removing older data as necessary.





3. Concurrency and Thread Safety

a. Concurrency Management

- The aggregation server handles multiple simultaneous GET and PUT requests.
- Synchronization is managed using Lamport clocks to ensure that requests are processed in the correct order.

b. Thread Safety

- Locks or synchronized blocks are used to prevent race conditions during data access and updates.
- Careful design ensures no deadlocks by avoiding nested locks and maintaining a clear lock hierarchy.

c. Multi-threading Strategy

- Request Handler Threads: Separate threads handle incoming client and content server requests.
- Data Management Thread: A dedicated thread manages data expiry and cleanup.

- Lamport Clock Update: Each thread is responsible for updating its local Lamport clock upon sending or receiving messages.

4. **Lamport Clock Implementation**
    a. Clock Management
        - Each entity (Client, Content Server, Aggregation Server) maintains a local Lamport clock.
        - Clocks are updated based on send, receive, and process events.

    b. Clock Synchronization
        - When an entity sends a message, it attaches its Lamport clock timestamp.
        - Upon receiving a message, the recipient compares its local clock with the received timestamp and updates its clock accordingly.

    c. Message Tagging
        - All HTTP requests and responses include Lamport clock timestamps in their headers.
        - This ensures that all entities have a consistent view of event ordering.

5. **Failure Management**
    a. Client-Side Failures
        - Clients implement retry logic for failed GET requests.
        - Timeout mechanisms ensure clients do not hang indefinitely.

    b. Server-Side Failures
        - The aggregation server uses fallback mechanisms to handle failure scenarios, such as failing to contact a content server.

    c. Network Failures
        - Both clients and servers have mechanisms to detect and recover from network interruptions.
        - Lamport clocks help ensure consistency even if a message is delayed due to network issues.

6. **Testing Strategy**
    a. Unit Testing
        - Test individual components such as request handlers, data storage, and clock management.

    b. Integration Testing
        - Simulate complete workflows, including client requests and content server updates.
        - Test scenarios with simultaneous GET and PUT requests to validate synchronization.

    c. Concurrency Testing
        - Stress test the server with multiple threads to ensure thread safety.
        - Test for race conditions, deadlocks, and proper handling of expired data.

    d. Failure Testing
        - Simulate network failures, server crashes, and client disconnections to ensure reliable recovery.