1. **Overview and Components**
   1. 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* Expire outdated or stale weather data.
  1. Client

Requests weather data from the aggregation server

* Send HTTP GET requests.
* Handle response data, including Lamport clock timestamp validation.
* Manage failures and retries.
  1. 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.

1. **Data Flow and Interactions**
   1. 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.
  1. 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.
  1. 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.

A diagram of a software process

Description automatically generatedA diagram of a server

Description automatically generated

1. **Concurrency and Thread Safety**
   1. 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.
  1. 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.
  1. 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.

1. **Lamport Clock Implementation**
   1. Clock Management

* Each entity (Client, Content Server, Aggregation Server) maintains a local Lamport clock.
* Clocks are updated based on send, receive, and process events.
  1. 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.
  1. 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.

1. **Failure Management**
   1. Client-Side Failures

* Clients implement retry logic for failed GET requests.
* Timeout mechanisms ensure clients do not hang indefinitely.
  1. Server-Side Failures
* The aggregation server uses fallback mechanisms to handle failure scenarios, such as failing to contact a content server.
  1. 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.

1. **Testing Strategy**
   1. Unit Testing

* Test individual components such as request handlers, data storage, and clock management.
  1. Integration Testing
* Simulate complete workflows, including client requests and content server updates.
* Test scenarios with simultaneous GET and PUT requests to validate synchronization.
  1. Concurrency Testing
* Stress test the server with multiple threads to ensure thread safety.
* Test for race conditions, deadlocks, and proper handling of expired data.
  1. Failure Testing
* Simulate network failures, server crashes, and client disconnections to ensure reliable recovery.