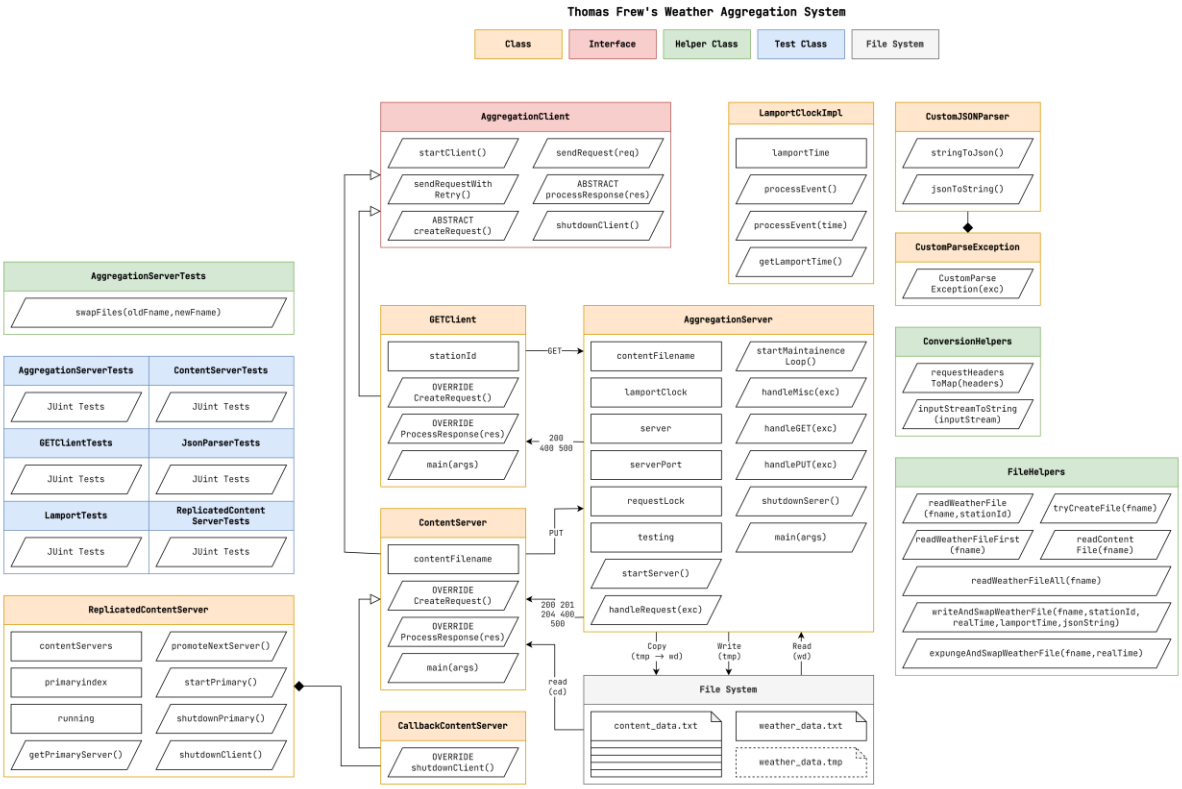
Assignment 2 - Final Design Sketch and Documentation

By Thomas Frew - a1850112

Sketch



Documentation

AggregationClient

An interface for the system's clients (GETClient and ContentServer).

AggregationClients run on a loop that automatically sends commands to AggergationServers every 2 seconds.

LamportClock

An implementation of a LamportClock is used by all clients and servers.

LamportClocks can process internal events, process Messages (which have an associated "time" field and return their own time. Their implementation is based on the Lamport Weak Clock Model.

GETClient

A client that fetches data from the AggregationServer.

GETClients are initialised with their hostname, their AggregationServer's hostname, and, optionally, the ID of the station from which they want to receive content.

If a station ID is omitted, then a GETClient will instead fetch the most recently committed data.

GETClients send a GET request to their AggregationServer once every 2 seconds.

If the return status is 200, the requested weather data is printed to stdout. Otherwise, an error code is printed:

- 400: A non-GET request was sent.
- 404: The requested content does not exist.
- 500: The server had an internal error (e.g. malformed JSON).

ContentServer

A client that pushes data to the AggregationServer.

ContentServers are initialised with their AggregationServer's hostname and the filename of their content file.

They send a PUT command to their AggregationServer once every 2 seconds.

The return status of a command is printed:

- 200: The weather data was correctly pushed.
- 201: This is the first weather data file that was created.
- 204: The request is missing weather data.
- 400: A non-PUT request was sent.
- 500: The server had an internal error (e.g. malformed JSON).

ReplicatedContentServer

A client that pushes data to the AggregationServer from several aggregation servers.

ReplicatedContentServers are initialised with the filename of their content file and 1+ AggregationServer's hostnames.

A ContentServer is created for each AggregationServer hostname. The first content server is the list is elected as the "primary" content server, and gets to send data to its AggregationSever. All other ContentServers sit idle.

If the AggregationServer of the primary is unreachable, the next ContentServer is elected as the new primary.

Once the last ContentServer is elected as priamry (and dies), election loops back around, and the first ContentServer becomes the primary once more.

Unlike indivudal ContentServers, a ReplicatedContentServer never goes down. It will keep electing new primaries and trying to contact them forever.

AggregationServer

A server that aggregates weather in a weather_data file (text document) and servers this data to clients.

AggregationServers listen out to packets from AggregationClients. These packets are validated and processed immediately. A simple mutex prevents multiple messages from being processed at the same time.

Processed packets include:

- GET: Sends the requested weather data to a GETClient.
    1. If the client requested data from a specfic station ID, send the data from that station. Send a 404 if this doesn't exist.
    2. If the client did not request data from a specfic station ID, send the data from the station that was most recently updated. Send a 404 if no data is available to send.
- PUT: Commits the send weather data to the weather_data file.
    1. The original weather_data file is copied to a temporary file.
    2. The new weather data is committed to the temporary file.
    3. The temporary file is swapped with the original weather_data file in a single, atomic operation.

AggregationServers also run on a maintainence loop. Every 30 seconds, data older than 30 seconds is "purged" (deleted) from the weather_data file.

This server model has several benefits:

1. Atomic Swaps: Swaps are an atomic file operation, meaning that the server never falls into an "invalid state" if it is interrupted during a swap.
2. Guaranteed Writing: ContentServers re-send their PUT requests if they didn't get a 200/201 response. Therefore, if the server is interrupted during a swap, the request does not get lost.
3. Reliable Information: Since requests are serviced atomically, GETClients will always get the most recent data.
4. Performance: Purging only occurs stricly every 30 seconds, so minimal computational time is spent on upkeep.

Lamport Clocks and Synchrony

We use the following rules to maintain synchrony in the system:

A. All requests are processed as an event in Lamport Time.

B. When two messages arrive simultaneously at an AggregationServer with different Lamport Times, the message with the smaller time is processed first.

C. When two messages arrive simultaneously at an AggregationServer with the same Lamport Time but different protocols, then PUTs are processed before GETs.

D. When two messages arrive simultaneously at an AggregationServer with the same Lamport Time and the same protocol, the message with the lexicographically smaller JSON string is processed first.

E. If a PUT for a particular station arrives with a smaller Lamport Time than another PUT of the same station, that PUT is discarded, and a 500 response is sent.

F. If a GET for a particular station arrives with a smaller Lamport Time than another GET of the same staion, there is no issue, and the operations proceed as normal.

Testing

A lot of effort has been put into producing tests with significant coverage.

AggregationServerTests

Tests for AggregationServer that do not involve serving GETClients or ContentServers. There is very little beahviour for these tests.

1. expungeDataOnStartup: Check that all outdated data is purged on the server's startup.
2. expungeDataRegularly: Run the server and check that data older than 30 seconds is purged every 30 seconds.

ContentServerTests

Tests for lone (non-replicated) ContentServers.

1. sendFirstData: Send data to an AggregationServer for the first time, and confirm that it is committed.
2. sendRepeatedData: Send data to an AggregationServer many times, and confirm that it is committed.
3. sendDifferentData: Send data from different weather stations and confirm that it is committed.

4. sendDataWithoutValidFields: Fail to send data that lacks any weather fields and ensure the response is 500.
5. sendDataWithoutID: Fail to send data that has some weather fields, but lacks an ID. Ensure the response is 500.
6. sendEmptyJSON: Fail to send data that lacks any data, and ensure the response is 204.
7. regularRequestsSent: Run the ContentServer and ensure data is pushed every 2 seconds.

ContentServerTests

Tests for GETClients.

1. fetchSoleData: Fetch the only entry from an AggregationServer.
2. fetchMostRecentData: Fetch the most recent entry from an AggregationServer.
3. fetchSpecificData: Fetch a specific (not most recent) entry from an AggregationServer.
4. fetchMissingData: Fail to fetch a non-existent entry from an AggregationServer. Ensure the response is 404.
5. fetchNoData: Fail to fetch a non-existent entry from an AggregationServer that has no data. Ensure the response is 404.
6. regularRequestsSent: Run the GETClient and ensure data is fetched every 2 seconds.

JsonParserTests

Tests for my CustomJsonParser.

1. parseOneField: Parse a JSON object with one string field.
2. parseManyFields: Parse a JSON object with multiple string and non-string fields.
3. parseNoFields: Parse a JSON object with no fields.
4. parseFieldWithColon: Parse a JSON object containing a colon within a field value.
5. parseWithWhitespace: Parse a JSON object that has leading and trailing whitespace.
6. failParseNoLeadingCurlyBrace: Fail to parse a JSON object missing its leading curly brace.
7. failParseNoTrailingCurlyBrace: Fail to parse a JSON object missing its trailing curly brace.
8. failParseCommaItem: Fail to parse a JSON object with an unexpected comma inside a field value.

9. parseSimpleString: Parse a simple JSON string (1 key-value pair) into a string.
10. parseComplexString: Parse a complex JSON string (3 key-value pairs, whitespace, colons) into a string.
11. parseEmptyString: Parse an empty JSON object into a string.

LamportTests

Tests for LamportClockImpls.

1. lamportClockStartsAtZero: A Lamport clock starts at 0 when initialized.
2. lamportClockInternalIncrement: Increment a Lamport clock with internal events.
3. lamportClockLoadNewTime: A Lamport clock will update to an incoming time if that time is larger.
4. lamportClockKeepOldTime: A Lamport clock will reject an incoming time if that time is smaller.

ReplicatedContentServerTests

Tests for ReplicatedContentServers.

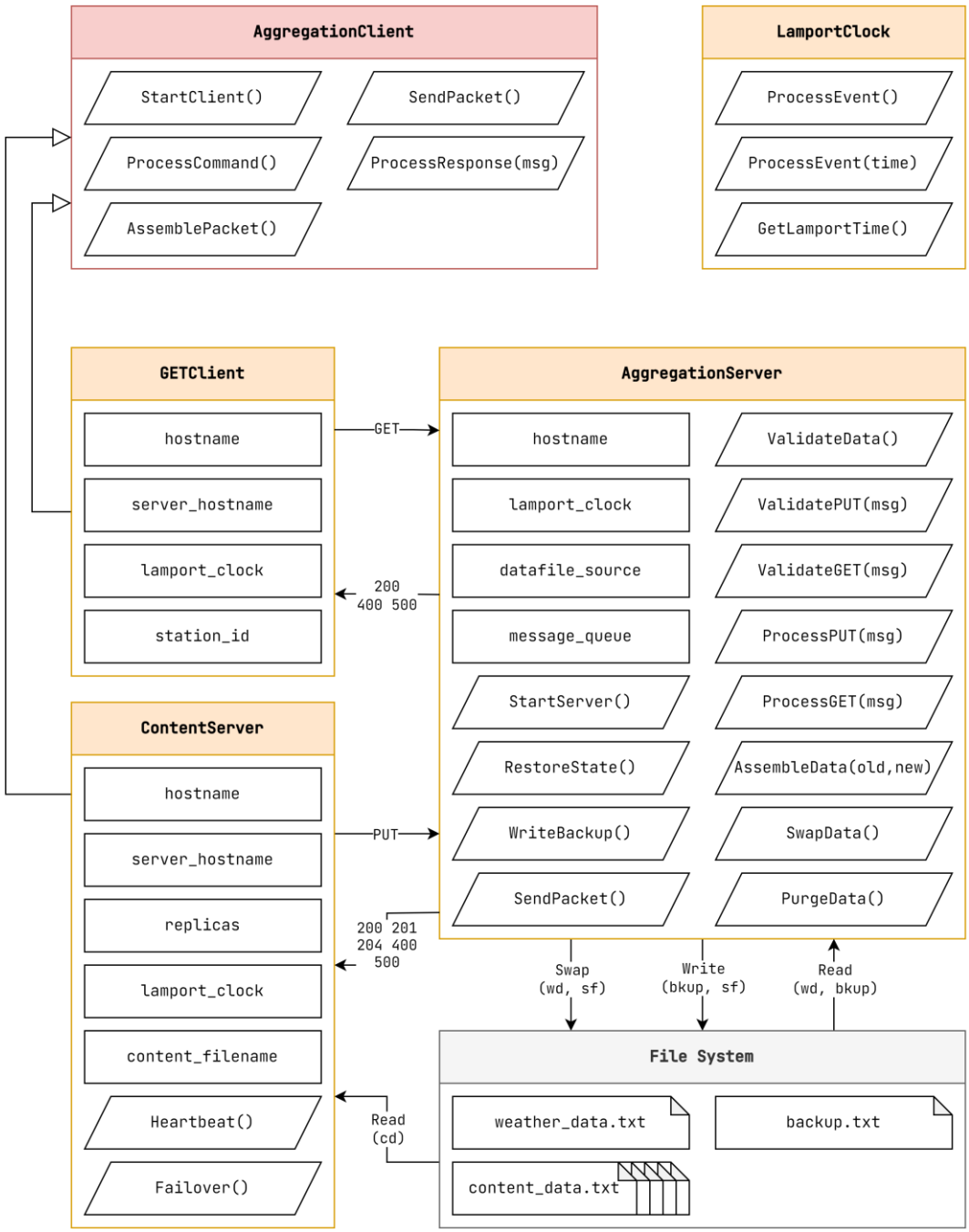
1. initialConfigurationValid: Create a ReplicatedContentServer with one ContentServer and confirm that it is the primary.
2. noFailover: Create a ReplicatedContentServer with two ContentServers. Both ContentServers have AggregationServers to connect to, so no failover should occur.
3. failover: Create a ReplicatedContentServer with two ContentServers. The first ContentServer is missing its AggregationServer, causing failover.
4. doubleFailover: Create a ReplicatedContentServer with three ContentServers. The first two ContentServers are missing their AggregationServers, causing two failovers.
5. failBack: Create a ReplicatedContentServer with two ContentServers. All ContentServers are missing their AggregationServers, causing continued failover between the two.

Assignment 2 - Draft Design Sketch and Documentation

By Thomas Frew - a1850112

For reference, here is the draft copy of my design sketch + proposed tests. This is provided to get a sense of how my project has changed over time.

Sketch



Documentation

AggregationClient

An interface for the system's clients (GETClient and ContentServer).

AggregationClients run on a loop that automatically sends commands to AggergationServers, although users can prompt these manually as well.

LamportClock

An implementation of a LamportClock is used by all clients and servers.

LamportClocks can process internal events, process Messages (which have an associated "time" field and return their own time. Their implementation is based on the Lamport Weak Clock Model.

GETClient

A client that fetches data from the AggregationServer.

GETClients are initialised with their hostname, their AggregationServer's hostname, and, optionally, the ID of the station from which they want to receive content.

They send a PUT command to their AggregationServer once every 30 seconds, but this can be manually called by writing "GET".

The return status of a command is printed alongside fetched weather data or error details:

- 200: The weather data was correctly fetched.
- 400: The request was malformed.
- 400: A non-PUT request was sent.
- 500: The server had an internal error (e.g. malformed JSON).

ContentServer

A client that pushes data to the AggregationServer.

ContentServers are initialised with a hostname, their AggregationServer's hostname and the filename of their content file.

They send a PUT command to their AggregationServer once every 30 seconds, but this can be manually called by writing "PUT".

The return status of a command is printed alongside error details, if applicable:

- 200: The weather data was correctly pushed.
- 201: This is the first weather data file that was created.
- 204: The request is missing weather data.
- 400: A non-PUT request was sent.
- 500: The server had an internal error (e.g. malformed JSON).

ContentServers are actually a collection of ContentServer replicas, with one “primary”.

The primary runs the event loop and handles all messaging. Its replicas check on its status and make sure it is OK. If the primary fails, another server elects itself as the primary.

AggregationServer

A server that aggregated weather data and services AggregationClients.

AggregationServers listen out to packets from AggregationClients. These packets are validated, placed in a message queue and later processed. A response is then sent back to the initiating client.

Processed packets include:

- GET: Calls SwapData, then sends weather data to a GETClient.
- PUT: Stages data for a commit (queues it).

AggregationServers also run on a 30-second loop, where the following maintenance behaviours are executed:

- PurgeData: Data older than 30 seconds is purged from the server.
- SwapData: Staged data is “swapped into” the main weather file. A new weather file is assembled from all staged commits, and then it is swapped with the existing one. Then, a backup is created to restore from in the case of a crash.

This server model has several benefits:

5. Fast Restoration: Complete backups are written after a swap occurs, allowing for rapid restoration in case of a crash.
6. Reliable Information: Since a swap is performed whenever the user requests data, they will always get the most reliable information.

7. Performance: Swaps only occur every 30 seconds, not once every update, so the server minimises its computational load.

Lamport Clocks and Synchrony

We use the following rules to maintain synchrony in the system:

G. All messages are processed as an event in Lamport Time.

H. All internal operations (purge, swap, restore) are processed as an event in Lamport Time.

I. When two messages arrive simultaneously at an AggregationServer with different Lamport Times, the message with the smaller time is processed first.

J. When two messages arrive simultaneously at an AggregationServer with the same Lamport Time but different protocols, then PUTs are processed before GETs.

K. When two messages arrive simultaneously at an AggregationServer with the same Lamport Time and the same protocol, the message with the lexicographically smaller hostname is processed first.

Testing

LamportClock

- New clock starts at 0.
- Processing event increments clock.
- Processing event mutates clock appropriately.

GETClient

- Client gets correct data from a GET request.
- Client gets correct data from a GET request with specific station identification.
- Client gets the correct error codes for bad requests (400 and 500).
- Client sends messages at regular intervals.

ContentServer

- Server successfully pushes data to a server.

- Server gets a 201 from a PUT request, as the first PUT request.
- Server gets a 200 from subsequent PUT requests.
- Server gets the correct error codes for bad requests (204, 400 and 500).
- Server sends messages at regular intervals.
- Server fails over to a backup when it is terminated,
- Server can fail over multiple times to several backups.
- Replicas agree on the primary server. The primary accepts that it is a primary.

AggregationServer

- Server can process correct PUT requests.
- Server can process malformed PUT requests.
- Server can process correct GET requests.
- Server can process malformed GET requests.
- Server can swap data into the main file from commits.
- Server’s backup file is correct after a swap.
- Server’s backup file is reloaded after a restart.
- Data older than 30 seconds is purged.
- Data is purged and swapped on a regular schedule.
- Data is swapped when a client requests for data.