DS assignment 2

System Architecture

This is a distributed system project designed for simulating a weather station. In this system, we use Java socket programming to create corresponding RESTful API interfaces for each component, and then use the HTTP protocol to facilitate communication between them. The system primarily consists of three components:

1. Aggregation Server (used to simulate the data center of the weather station).
2. Content Server (used to simulate weather station sensors for collecting information).
3. GETClient (used to simulate fetching data from the weather station).

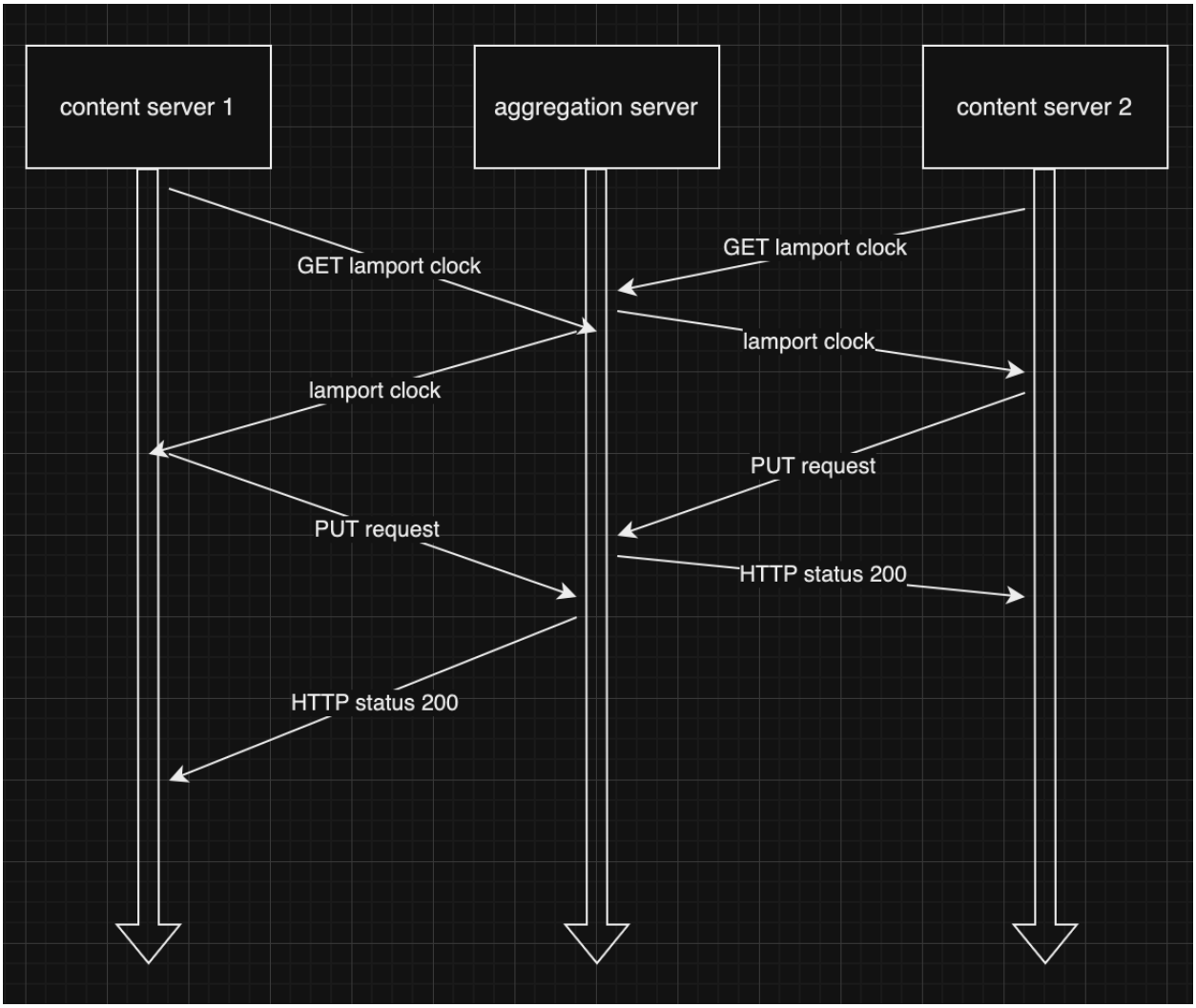
Additionally, there are other classes implemented to provide helper functionality, such as JSON parsing and Lamport clock synchronization.

Lamport Clock

Lamport Clock is used to implement timestamp functionality in distributed systems. Both the aggregation server and content server update their local timestamps when sending or receiving messages to ensure event order consistency in a distributed environment. Here, I will discuss two methods for implementing Lamport Clock, compare their advantages and disadvantages, and analyze the implementation method I chose.

Using socket to implement an aggregation server Lamport clock.

Using network communication to transmit event timestamps, each Content Server sends its event timestamp along with the event to other servers, which receive and use these timestamps to update their Lamport clocks. This approach is more suitable for distributed systems.

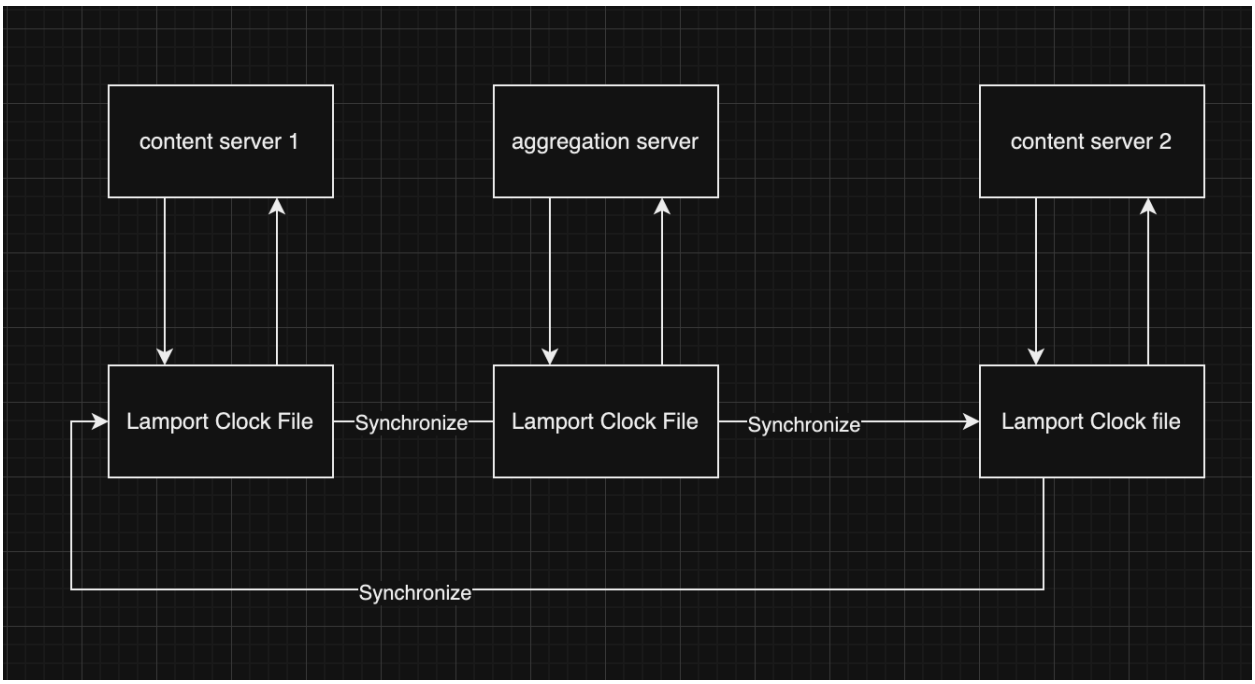


This approach can lead to a situation where every instruction in the Lamport Clock implementation requires a corresponding GET operation to obtain the Lamport Clock. This would double the number of required socket calls, and the network latency incurred when calling the GET instruction could potentially result in inaccuracies in the Lamport Clock.

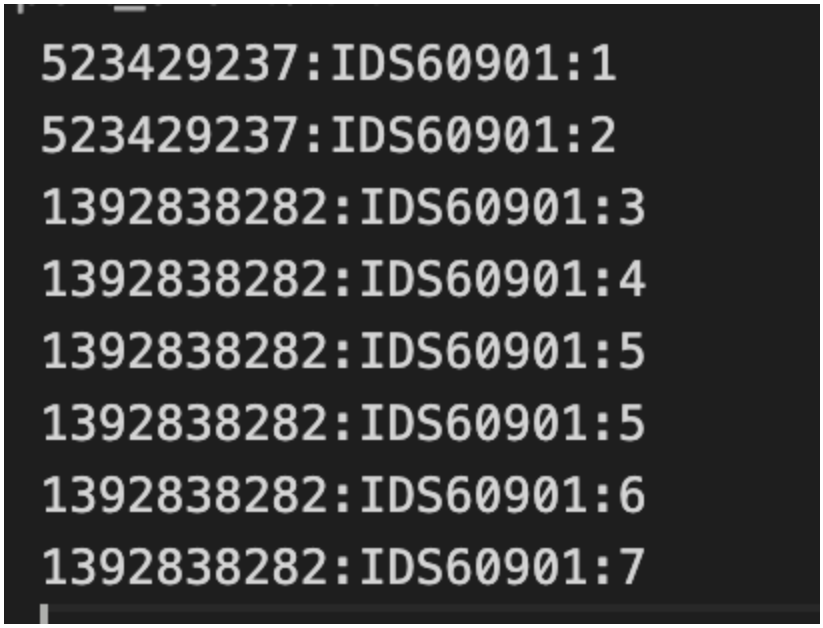
Using file logging to implement Lamport clocks.

In this system, the chosen method for maintaining Lamport Clocks is to create a file where each Content Server records the timestamps of its local events. Then, other Content Servers can monitor changes to the file to learn about the timestamps of events

on other servers and update their Lamport clocks accordingly. This way, each server can update its clock based on the records in the file.



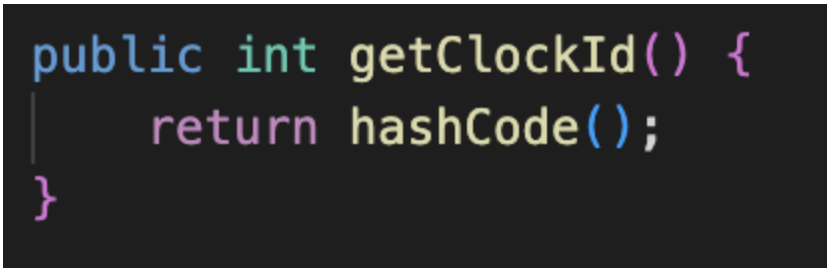
example of Lamport Clock file



All entries are stored in the format of `<hash identifier>:<station ID>:<Lamport Clock>` in the file.

Tie-Breaking strategy

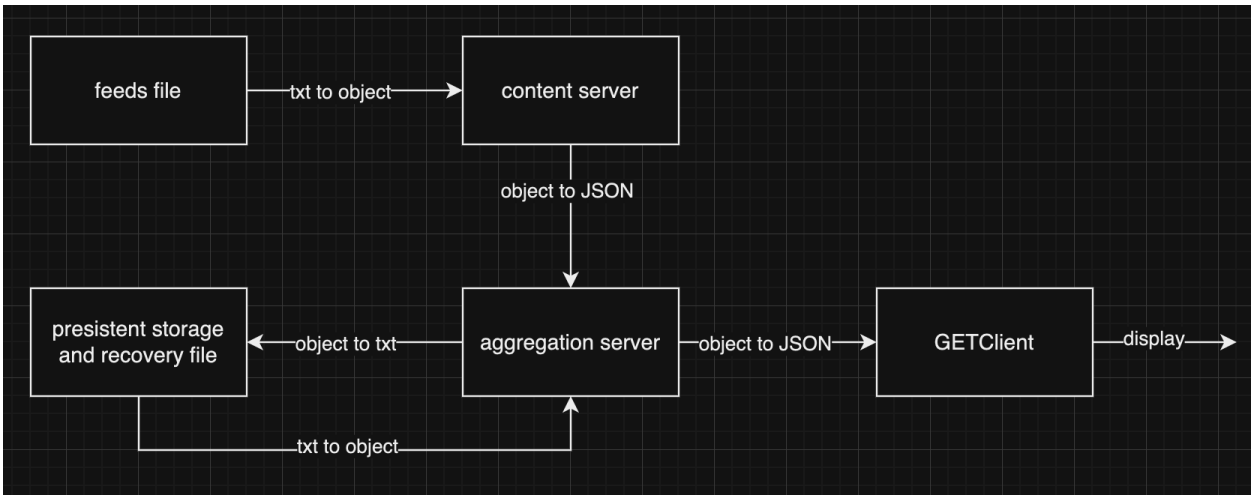
Even with Lamport Clocks in place, we still encounter scenarios where events happen at the exact same timestamp. In such cases, we need a mechanism to establish the global order of these events. We use the hash value derived from the Lamport Clock as the criterion for this determination. In other words, if two events occur on different servers and their clock values are the same, we can use the server's unique identifier to determine which event happened first.



JSON parser

JSON parsers are used to convert JSON-formatted data into object forms that the system can process. They play a crucial role in parsing and extracting data, allowing the system to conveniently handle and manipulate JSON data.

In this distributed system, all information transmission adheres to OOP (Object-Oriented Programming) and RESTful principles. This approach makes information handling and invocation extremely convenient. Information within the system exists in three forms: text (used for storage in permanent files), JSON (used for RESTful API transmission between different instances), and weather data objects.



UtilTool provides various conversion functions, including JSON parsing and TXT parsing. Inside the Weather Data Class, functions are also available for converting data into JSON and text strings. These features aid in data transformation and processing within the system, better catering to the needs of different components.

Aggregation Server

Instruction

The Aggregation Server serves as the data center for this weather distributed system. It is responsible for handling HTTP requests from Content Servers and clients, as well as maintaining all feeds from Content Servers.

When starting the Aggregation Server in default mode, it operates on port 4567. This default configuration enables the Aggregation Server to process HTTP requests and manage the incoming data feeds from Content Servers.

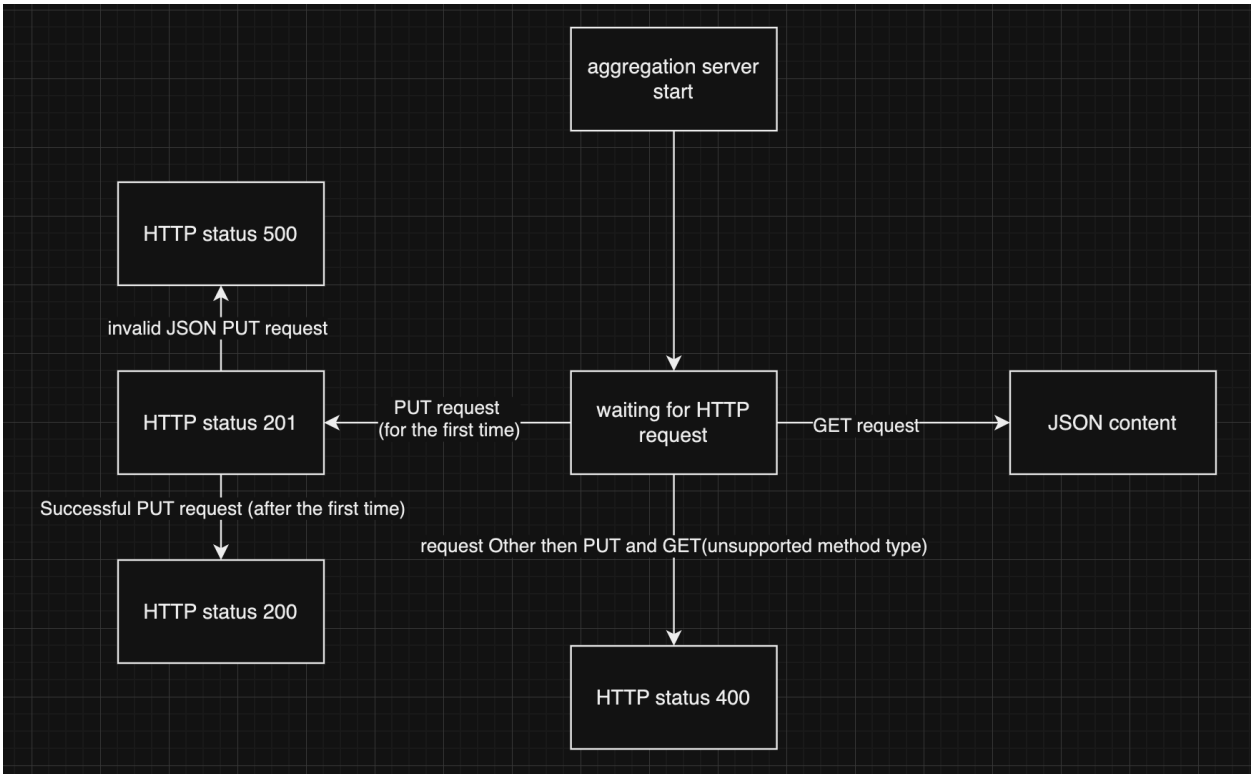
```
java AggregationServer
```

Alternatively, you can use command-line arguments to customize the port for starting the Aggregation Server. This allows you to specify a port of your choice when launching the server to suit your specific requirements.

```
java AggregationServer 8080
```

HTTP Implementation Behaviour

When the Aggregation Server receives an HTTP request, it will generate an HTTP response based on the method type and request body. Please provide more details about the specific responses you'd like to know about, and I'd be happy to help you further.



Fault Tolerance — Presistent Storage and Backup

Persistent storage and backups are crucial components of the weather distributed system. To ensure data persistence and reliability, you've implemented a point-in-time

snapshot strategy. This strategy involves periodically taking snapshots of the system, saving the current system state and data to a reliable storage medium, typically in the form of files. This approach allows for quick recovery in the event of system failures or data loss. Whenever the Aggregation Server receives a new PUT request, a snapshot of the Aggregation Server's system is taken, and the file is stored.

```
weather_data_backup.txt
```

The file is destroyed when the Aggregation Server's runtime ends intentionally, but it won't be destroyed due to unexpected program terminations, such as power outages or network interruptions. Additionally, when the Aggregation Server starts running, it checks the root folder to see if a backup file exists. This check helps determine whether the program exited normally. If a backup file is detected, the server will attempt to recover data from it.

With these measures in place, data persistence and reliability are ensured, and the risk of data loss is significantly reduced.

Heartbeat System — 30s Expired Data

To ensure the reliability and stability of the weather distributed system, we have introduced a heartbeat system. Content Servers periodically send heartbeat signals (simplified as PUT requests in this program) to indicate their active status to other components. Through the heartbeat system, we can monitor the operational status of components in real-time and take prompt action in case of failures or anomalies.

If a Content Server goes without sending a PUT request to the Aggregation Server for more than 30 seconds, we consider the connection with that Content Server as timed out. The Aggregation Server will automatically check all entries from that stationID and remove them from the system and backups. Clients will also be unable to receive information from that stationID until they come back online.

Specifically, every time a PUT request from a Content Server is added to the list, we initiate a multi-threaded timer. If no entry with the same stationID is added within the specified time, it triggers data expiration.

```java
/**
 * Start a timer task to trigger data expiration after a certain time.
 *
 * @param str The identifier for the data.
 */
private void startTimer(String str) {
    Runnable task = () -> {
        dataExpried(str);
    };
    executorService.schedule(task, delay:30, TimeUnit.SECONDS); // 30 seconds
}
```

Content Server

In the weather station distributed system, the Content Server serves as a data provider to simulate data supply. The Content Server uses HTTP PUT requests to provide feeds to the Aggregation Server. We use Java sockets to create a socket and convert weather data extracted from a file into JSON format, which is then sent to the Aggregation Server to facilitate communication.

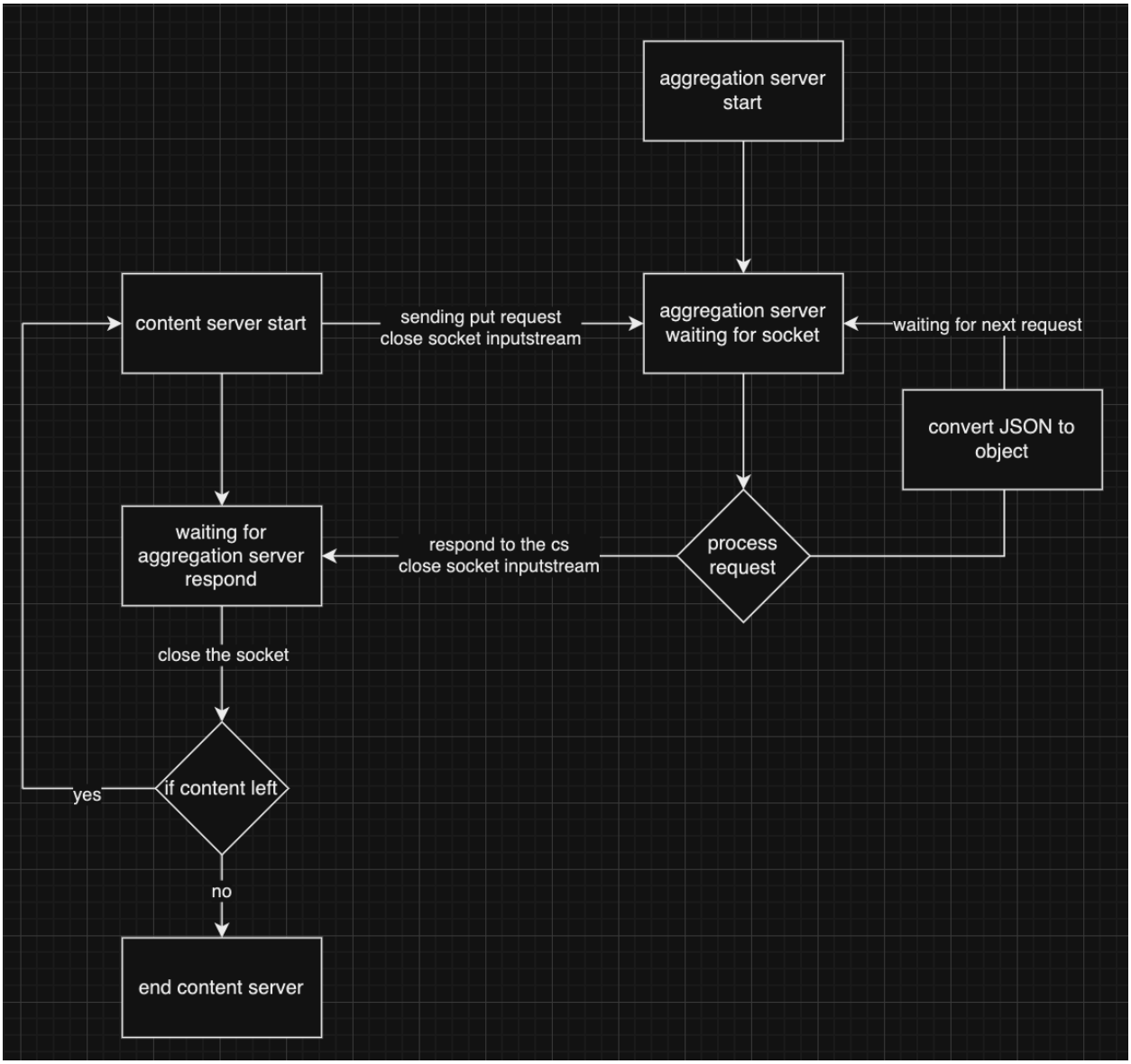
To start the Content Server, you need to provide the following parameters: (server address, port number, and file location). Here's an example of how to start the Content Server:

```
java ContentServer <server_address> <port_number> <file_location>
```

In this command, replace `<server_address>` with the Aggregation Server's address, `<port_number>` with the port number for communication, and `<file_location>` with the path to the file containing weather data. This command will initiate the Content Server and establish the necessary connection with the Aggregation Server.

```
java ContentServer localhost:4567 contentserverfeeds_1.txt
```

In our content server, to prevent socket deadlock, communication between the content server and the aggregation server follows the following linear pattern and continues in a loop. Each time a "put" request is made, a new socket is used, ensuring that there are no memory leaks. The content server initially starts and reads feeds from the file location specified in the command line arguments. It then enters a "put" request loop to communicate with the aggregation server, running until all the data from the feeds is used up, at which point it stops.



Fault Tolerance — Retry Connection on Disconnect

The communication between the content server and aggregation server is carried out using HTTP PUT requests. In the event of a lost connection during the communication process, the content server will automatically retry the operation until the connection is reestablished, and the data is successfully sent, or until a timeout occurs.

This fault-tolerant mechanism ensures the reliability and stability of the system. Even in cases of an unstable network or unreliable connections, data can still be transmitted and processed without disruption.

GETClient

GETClient is a component used to simulate the retrieval of weather station data. It does this by sending HTTP GET requests to the aggregation server to obtain weather data. Users need to specify the station ID that GETClient should retrieve in the command line parameters. When starting GETClient, you need to provide the server address and port number in the following format, which is consistent with the content server:

Here's an example of how to start GETClient:

```
java GETClient localhost:4567 IDS60901
```

GETClient will send a GET request to the aggregation server and retrieve the latest weather data for the specified station from the response. The retrieved data will then be displayed in the command line.

Automated Testing

To facilitate automated testing, we developed a suite of testing scripts. These scripts automatically run test cases and compare results against expected outcomes. Additionally, we designed a Makefile for automated code compilation and test execution. Our testing endeavours affirm that the system operates effectively in diverse scenarios and exhibits favorable performance.

aggregatio-2content.sh-get.sh	test with start 1 aggregation, 2 content server, 1 get client
aggregation-	test with 1 aggregation server and 1 content server

content.sh	
aggregation-content-get.sh	test with 1 aggregation server 1 contetn server and 1 get server
stree_test	start 20 ccontent server 1 aggregation server