Backend System Design for Data Processing

Atif

Theoretical Approach

This document outlines the detailed design of a backend system for storing and processing high-frequency data. The system is designed to handle tasks such as data ingestion, storage, and processing while ensuring modularity, scalability, and performance.

System Modules and Responsibilities

1. API Module

The API module is responsible for interacting with clients. It provides endpoints for data ingestion and retrieval. This module is implemented using FastAPI.

- **POST /upload**: Accepts JSON payloads containing high-frequency data (e.g., GPS coordinates).
- **GET /retrieve**: Returns processed data summaries, such as average values.

2. Data Storage Module

The data storage module handles the persistence of incoming data. In this design, we use an in-memory database such as Redis or a lightweight file-based storage like SQLite.

- Stores raw sensor or GPS data.
- Ensures efficient retrieval for processing.

3. Data Processing Module

This module performs real-time data processing tasks, such as calculating averages or aggregating data over a specific time window.

- Processes incoming data in batches.
- Calculates summaries, such as average latitude and longitude.
- Handles computational efficiency to support high-frequency data streams.

4. Simulation Module

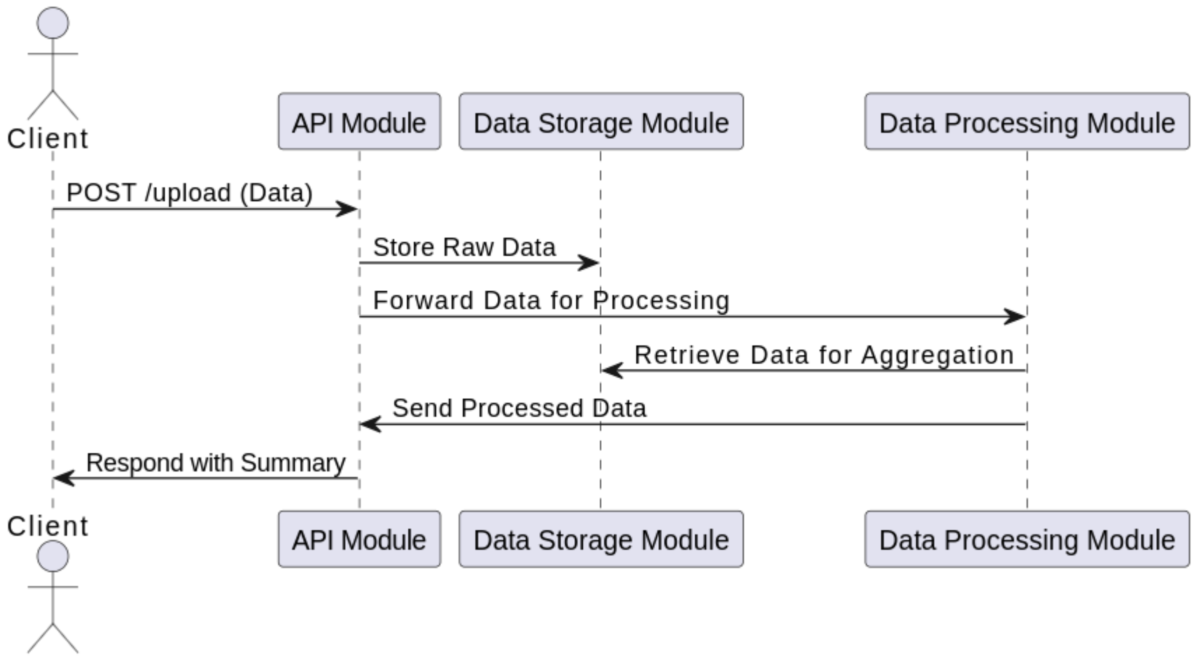
This module generates random high-frequency data to simulate real-world scenarios. For example:

```
{
"timestamp": "2025-01-23T12:00:00Z",
"latitude": 52.5200,
"longitude": 13.4050
}
```

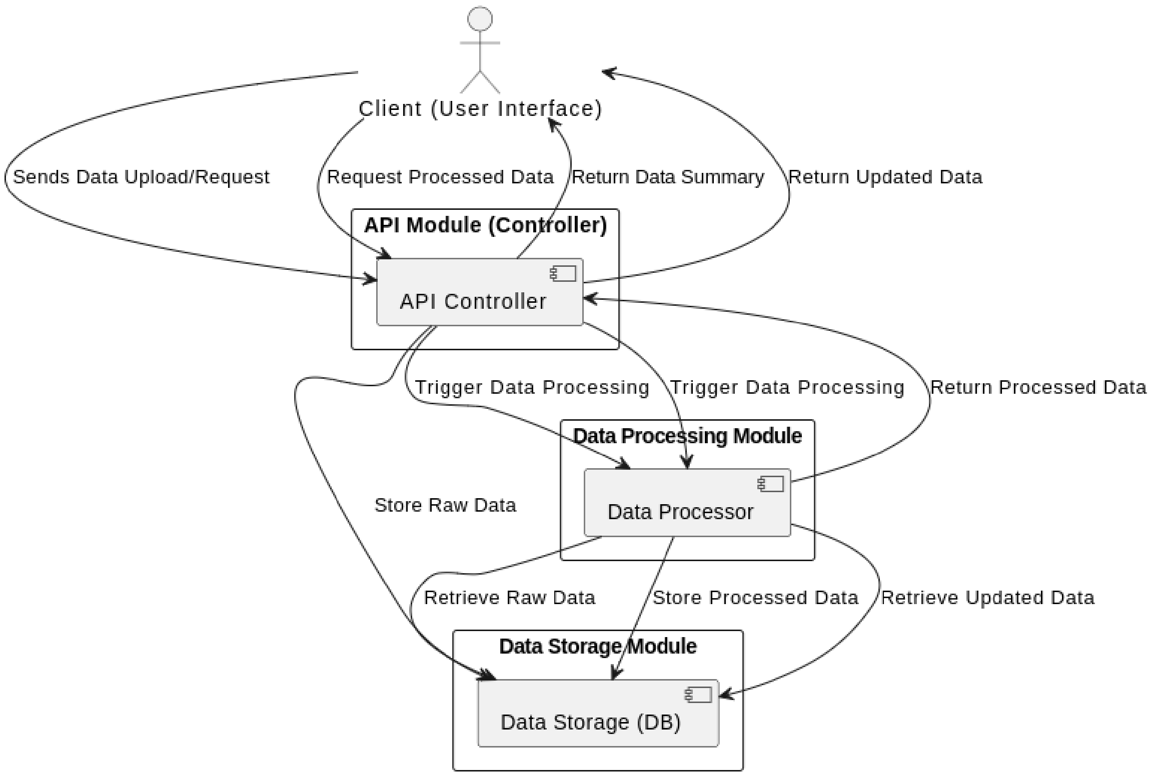
5. Testing Module

This module includes scripts to test the API by sending simulated data and validating the responses. It uses Python's `requests` library for making HTTP calls.

Data Flow Diagram



High Level Design



1.Practical Approach

The backend system was implemented using **Flask**, a lightweight web framework in Python, to handle two API endpoints:

- `/upload`: Accepts GPS data in JSON format via POST requests and stores it in a file (`incoming_data.json`).

- `/retrieve`: Processes the stored data (e.g., calculates the average latitude and longitude) and returns the results via GET requests, while also saving the processed data in a file (`processed_coordinates.json`).

This system simulates real-time data streaming by integrating a data generator and an API tester script.

2. Key Components

a. Flask Backend Code

Below is the core implementation of the Flask backend:

Listing 1: Flask Backend Implementation

```python
from flask import Flask, request, jsonify
import json
import os

app = Flask(__name__)

# File paths for storing incoming and processed data
INCOMING_DATA_FILE = "incoming_data.json"
PROCESSED_DATA_FILE = "processed_coordinates.json"

# POST endpoint to store incoming data
@app.route('/upload', methods=['POST'])
def upload_data():
    data = request.get_json()
    if not data:
        return jsonify({"error": "Invalid data"}), 400

    # Append data to the file
    if os.path.exists(INCOMING_DATA_FILE):
        with open(INCOMING_DATA_FILE, 'r') as f:
            existing_data = json.load(f)
    else:
        existing_data = []

    existing_data.append(data)

    with open(INCOMING_DATA_FILE, 'w') as f:
        json.dump(existing_data, f, indent=4)

    return jsonify({"message": "Data uploaded successfully"}), 200

# GET endpoint to retrieve processed data
@app.route('/retrieve', methods=['GET'])
def retrieve_data():
    if not os.path.exists(INCOMING_DATA_FILE):
        return jsonify({"error": "No data available for processing"}), 400

    # Load data and calculate averages
    with open(INCOMING_DATA_FILE, 'r') as f:
        data = json.load(f)

    if not data:
        return jsonify({"error": "No data available for processing"}), 400

    total_latitude = sum(entry['latitude'] for entry in data)
    total_longitude = sum(entry['longitude'] for entry in data)
    count = len(data)

    processed_data = {
        "average_latitude": total_latitude / count,
        "average_longitude": total_longitude / count,
        "data_count": count
    }

    # Save processed data to a file
    with open(PROCESSED_DATA_FILE, 'w') as f:
        json.dump(processed_data, f, indent=4)

    return jsonify(processed_data), 200
```

```
if __name__ == "__main__":
    app.run(debug=True)
```

b. API Tester Integration

The API tester sends real-time GPS data to the backend and validates its functionality by:

1. Sending POST requests with generated random data.
2. Sending GET requests to verify that processed results are correct.

Key snippet from the tester:

Listing 2: Random GPS Data Generation

```
def generate_random_gps_data():
    from datetime import datetime, timezone
    import random
    return {
        "timestamp": datetime.now(timezone.utc).isoformat(),
        "latitude": round(random.uniform(-90, 90), 4),
        "longitude": round(random.uniform(-180, 180), 4)
    }
```

3. Challenges Faced

1. **Handling Real-Time Data**:
   - **Challenge**: Simulating continuous real-time data ingestion while ensuring data integrity.
   - **Solution**: Added buffering and file-based storage to handle bursts of data.
2. **Ensuring Robust Error Handling**:
   - **Challenge**: Preventing crashes due to malformed or missing data.
   - **Solution**: Implemented input validation and clear error messages in the backend.
3. **Testing API Endpoints**:
   - **Challenge**: Validating end-to-end functionality with asynchronous POST and GET requests.
   - **Solution**: Created a well-structured API tester script with retry mechanisms and logging.