Technische Universität Berlin
Chair Mobile Cloud Computing

Fog Computing - Summer Term 2023

Documentation to the Prototyping Assignment

Application for Weather Data Collection

Supervisors: Prof. Dr. D. Bermbach, N. Japke und M. Grambow

Authors: Christopher Hansen
Ettore Carlo Marangon
Tim Michaelis

Abgabedatum: 11. July 2023

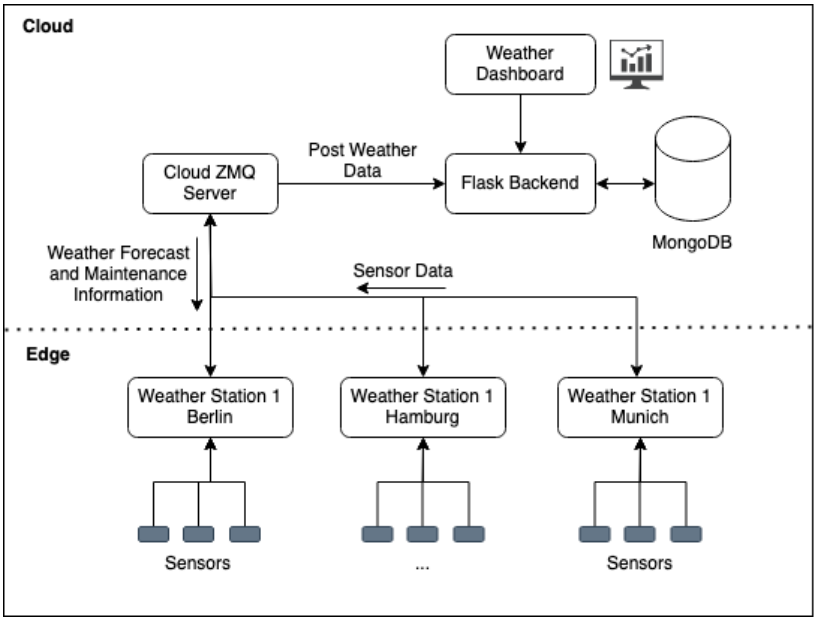
Repository link: https://github.com/haChristopher/tu-fog-computing

Introduction

The project focused on developing an application for weather data collection using simulated sensors. The application is embedded into a fog computing infrastructure which includes three edges and various cloud components. The main goal was to manually implement a reliable message delivery system facing multiple fog-specific challenges.

Architecture

The Architecture consists of three edge nodes representing weather stations in different German cities. Each station is equipped with multiple sensors delivering data about temperature, humidity, and other weather related data. In the cloud we deployed our central server node. The server node is responsible for the communication with the distributed weather station. The stations transfer their sensor readings to the cloud node and in return the cloud node provides the stations with data about weather forecasts such as severe weather events including storms or flooding. Furthermore the server node informs the stations about upcoming maintenance schedules. The cloud node then posts the weather data to our backend which then persists the data in a cloud hosted mongoDB. The backend also serves data to our weather dashboard which shows sensory readings from different weather stations.



Picture 1: Technical overview

Sensors and Data Generation

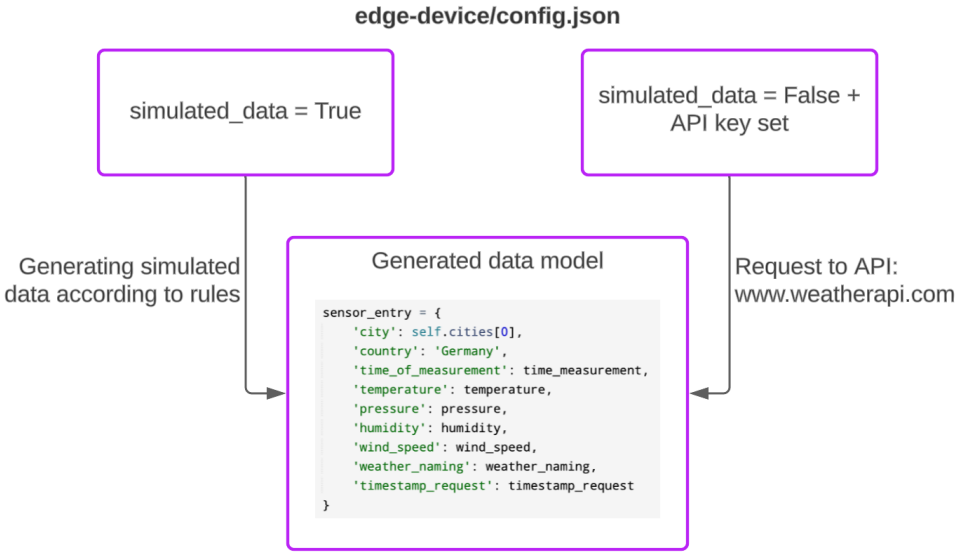
We had two options for sourcing the data: parsing live data from the weather API provided by www.weatherapi.com or simulating the environmental data within a specific range.

The API offered a comprehensive set of weather data for different locations, which we utilized to retrieve real-time environmental data from Berlin, Hamburg and Munich. By parsing the API's response, we could directly access accurate and up-to-date information on temperature, pressure, humidity, and wind speed. This approach enabled us to incorporate the most current weather data into our application.

Alternatively, we designed and implemented simulated sensors to generate realistic environmental data. These sensors were programmed to simulate three weather stations located in Hamburg, Berlin, and Munich. Each

station collected data on key weather parameters, including temperature, pressure, humidity, and wind speed. By simulating the data, we were able to generate diverse and dynamic datasets that closely resembled real-world weather conditions.

For real-time data, “simulated_data” must be set to False in our configuration file and an api key for weatherapi.com needs to be provided.



Picture 2: Data generation

Both the simulated and live data approaches provided valuable data sources for our project. The simulated sensors allowed us to have full control over the data generation process, enabling us to test different scenarios and validate our application's performance under specific conditions. On the other hand, the live data parsing from the weather API ensured that our application could access real-time weather data, allowing for more accurate and timely analysis and forecasts. Due to the scope of this project, we agreed on the data structure illustrated in Picture 2.

The cloud server sends generated data about maintenance schedules, weather forecasts and alerts to the edge devices, an example of the incoming data on the edge devices can be seen in Picture 3.

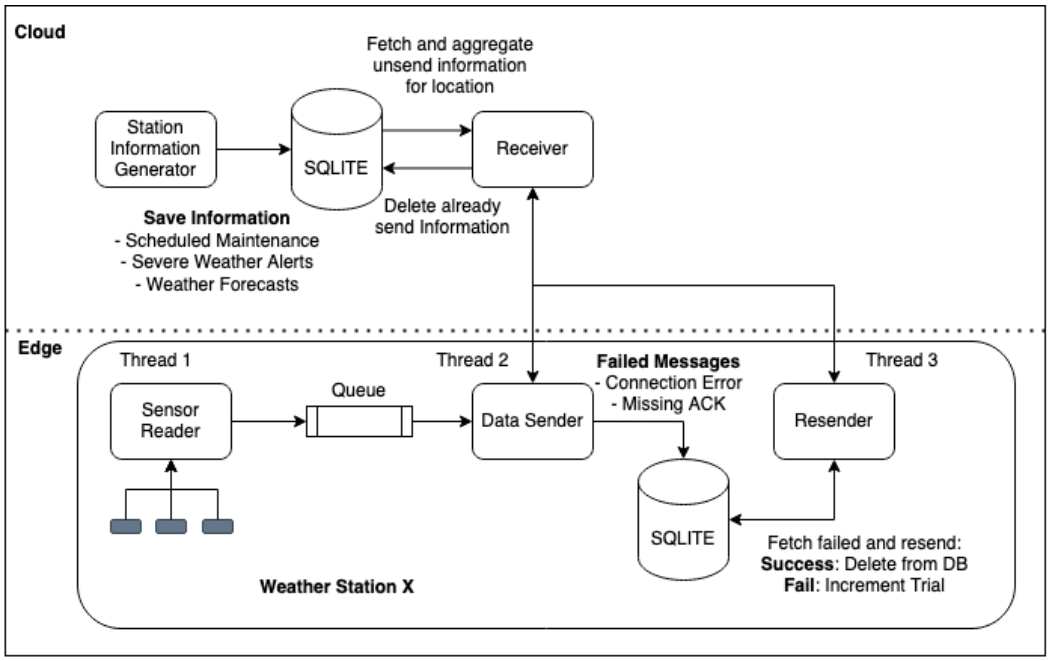
```
tu-fog-computing-edge-client-1  | 2023-07-11 15:42:00,272 - INFO - Weather forecast: Snowy at 2023-07-11 23:30:45
tu-fog-computing-edge-client-1  | 2023-07-11 15:42:00,272 - INFO - Weather forecast: Smoky at 2023-07-20 20:36:21
tu-fog-computing-edge-client-1  | 2023-07-11 15:42:00,280 - DEBUG - Message resent successfully
tu-fog-computing-edge-client-1  | 2023-07-11 15:42:00,707 - DEBUG - Message sent successfully
tu-fog-computing-edge-client-1  | 2023-07-11 15:42:02,287 - ERROR - Alerts: Sandstorm
tu-fog-computing-edge-client-1  | 2023-07-11 15:42:02,287 - INFO - Planned maintenance: repair at 2023-07-16 14:48:59
tu-fog-computing-edge-client-1  | 2023-07-11 15:42:02,287 - INFO - Weather forecast: Cloudy at 2023-07-13 06:10:08
tu-fog-computing-edge-client-1  | 2023-07-11 15:42:02,287 - INFO - Weather forecast: Rainy at 2023-07-12 13:00:14
tu-fog-computing-edge-client-1  | 2023-07-11 15:42:02,291 - DEBUG - Message resent successfully
```

Picture 3: Weather Forecasts, Alerts and Scheduled Maintenance from Server

Reliable messaging

As our solution operates in a distributed fog environment we need to consider that outages can occur and multiple edge devices or server nodes can fail and be not available for a short or long period of time. To address this challenge we implemented several mechanisms to counteract such failures. Each weather station consists of three main threads. One thread takes care of reading consistently the data from the sensors and stores them in a queue. The task of the second thread is to send the newest incoming data, if a failure occurs or connection is not possible the message gets persisted on disk in a local SQLite database. The task of the third thread is to check the data on disk and try to resend the data if it exists. With these three threads we can ensure multiple things: (1)

reading sensor data is never blocked by connection issues, (2) failed data is persisted to disk and gets retried and (3) even if the Station itself fails after a restart the persisted data gets resend.



Picture 4: Reliable messaging overview

Picture 5 shows a sample from the SQLite DB on one of the weather stations storing messages which were not able to be sent. Every entry has a unique id, a timestamp, a flag showing if it was sent, the number of sending attempts and then actual message body. In the picture you can see that the Resender always tries sending the oldest message first. In the below scenario the server was completely offline. All messages with a send flag set to one get periodically deleted.

	id	message	send	send_attempt	Timestamp
1	71	e_of_measurement": 1688817101, "temperature": 22.6, "p	0	5	2023-07-08 11:51:42
2	72	ıe_of_measurement": 1688817101, "temperature": 24.1, "p	0	1	2023-07-08 11:51:44
3	73	e_of_measurement": 1688817102, "temperature": 24.2, "p	0	1	2023-07-08 11:51:46
4	74	e_of_measurement": 1688817146, "temperature": 20.5, "p	0	1	2023-07-08 11:52:27
5	75	ıe_of_measurement": 1688817147, "temperature": 22.7, "p	0	1	2023-07-08 11:52:29
6	76	e_of_measurement": 1688817181, "temperature": 25.6, "p	0	1	2023-07-08 11:53:02
7	77	e_of_measurement": 1688817193, "temperature": 21.3, "p	0	1	2023-07-08 11:53:14
8	78	e_of_measurement": 1688817193, "temperature": 24.6, "p	0	1	2023-07-08 11:53:16

Picture 5: SQLite sample Edge Client

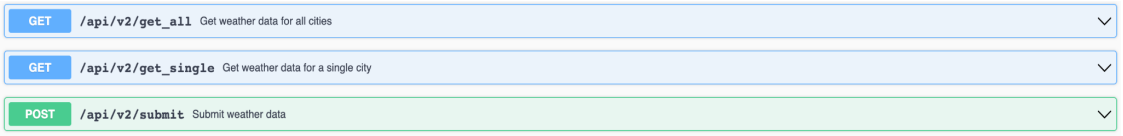
On the server side we implemented a similar pattern. All scheduled maintenance and alerts for each weather station gets stored on disk with a timestamp. In case a weather station is offline the information piles up in the database. Once the station is back on and sends its first da package the server aggregates the saved information on disk and sends them to the client as a batch. Below is an example from the server-side table.

	id	message	send	city	Timestamp
1	1	ɔrecast": [{"id": 1, "message": "Partly cloudy", "time": "2023-07-13 19:05:27	0	Hamburg	2023-07-08 12:06:47
2	2	023-07-11 10:57:40", "level": "MEDIUM"}], "maintenance_schedule": [{"id":	0	Hamburg	2023-07-08 12:06:48
3	4	age": "inspection", "time": "2023-07-09 04:38:34"}], "weather_forecast": [{	0	Munich	2023-07-08 12:06:50
4	5	nessage": "repair", "time": "2023-07-11 07:53:49"}], "weather_forecast": []}	0	Hamburg	2023-07-08 12:06:51
5	6	ısage": "repair", "time": "2023-07-12 03:21:36"}], "weather_forecast": [{"id"	0	Hamburg	2023-07-08 12:06:52
6	7	cast": [{"id": 1, "message": "Thunderstorm", "time": "2023-07-15 06:32:49"	0	Munich	2023-07-08 12:07:50
7	8	nessage": "repair", "time": "2023-07-14 16:17:06"}], "weather_forecast": []}	0	Hamburg	2023-07-08 12:07:51

Picture 6: SQLite sample Cloud Server

Backend

Our backend implementation for the Fog Computing project revolves around creating a weather data API using the Flask framework and MongoDB for data storage. The API supports two major endpoints: data submission and data retrieval. The "submit_data" endpoint handles POST requests, where users can submit weather data for a specific version of the API. We validate the received JSON payload, ensure the presence of required fields, and then insert the data into our MongoDB database. For data retrieval, we have implemented two endpoints. The "get_single" endpoint accepts GET requests and retrieves weather data for a single city based on the provided city name. The "get_all" endpoint retrieves weather data for all cities available in the database. In both cases, we query the database, sort the results based on the time of measurement, and return the data in JSON format. Our backend code also includes error handling, such as returning appropriate HTTP status codes and error messages for scenarios like invalid requests, missing data, data not found and more.



Picture 5: Swagger Documentation of our API

Dashboard

The application provides a dashboard. It visualizes the collected weather data for each variable (temperature, pressure, humidity, and wind speed) and city (Hamburg, Berlin, and Munich). The diagrams showcase the trends of the variables over the last ten received timestamps for each city. It allows users to track the temperature, pressure, humidity and wind speed fluctuations and identify patterns or anomalies.