NKUA Weather Station Network Project

Backend Request Handers, Database System

Version 2

Ronaldo Tsela **(August 2023)**

Contents

List of Figures

List of Tables

Purpose

The purpose of this report is to present the preliminary design stage for the development of a ground-based weather station network and its associated Application Programming Interface (API) and User Interface (UI). The second stage involves designing the backend data handling mechanisms. This includes creating the appropriate structures to receive recorded data from each station within the network and storing it to mass storage devices or forwarding it to the relevant programs or APIs. Additionally, this report outlines the telecommand transmission mechanism used to control specific activities at the stations remotely.

Introduction

As users, we interact with various computer programs and applications through graphical interfaces. When characterizing a good application, we often refer to the beautiful colors, multiple frames, explanatory texts, and vivid graphical interpretation of data. However, these are merely additional soft features, known as views, and do not necessarily determine a good application in terms of performance and efficiency. A robust low-level backend stack is essential for the proper functioning of an application, regardless of its graphical representation or colors.

The backend stack of an application comprises various low-level mechanisms, sub-routines, or even complete programs and applications that manage the system at its core level. Tasks such as data reception from point to point, data storage and forwarding, database management, encryption and security, and translation between different protocols are some of the fundamental operations performed by a backend system.

Applications are constructed using the resources provided by the backend system. These resources, commonly known as Application Programming Interfaces (APIs), enable the application designer to interact with low-level components without needing extensive knowledge about them. Especially in large systems where full stack applications are developed by multi-member teams, this feature proves beneficial for timely and efficient application development.

The project at hand also requires a strong low-level backend system to manage the data links and user requests independently of the user application. This is achieved through a low-level API, as described. Therefore, this report outlines the design and development decisions for the weather station backend data management system.

Objectives

Path management: The backend of a web-based application comprises multiple sub-routines, programs, and files, each dedicated to a specific task within the system. The organization of paths and URLs for accessing these entities is crucial for achieving a well-designed, developed, and maintainable system, ensuring smooth overall operation.

Database schema: The database schema outlines the characteristics and structure of the database intended to store historical data. One of the objectives is to create a straightforward database schema to accommodate the generated data.

Data packet handlers: Data packet handlers are a set of sub-routines/ sub-programs designed to receive data from endpoints, such as stations, and process it appropriately. Additionally, data handlers may also refer to a collection of low-level API functions responsible for managing transactions between the user interface and the backend stack.

Command transmitter and receiver: The command transmitter refers to a low-level program capable of transmitting command packets delivered from the end-user to specific station entities. On the other hand, on-board receiver logic is implemented to read these commands and execute them properly.

Backup: When dealing with web-based applications and data collection, the capability for backup mechanisms in case of malfunctions becomes crucially important.

Traffic and validation test: A traffic test involves creating software testbenches to assess the backends ability to handle multiple endpoint requests and responses simultaneously. During this validation process, all aspects of the hardware, backend interconnection, and system performance will be thoroughly examined. Based on the results, appropriate adjustments will be made to enhance the system's performance and achieve better overall outcomes. At this point the idea of developing a virtual simulator may be very handy.

Requirements

Host: The server for hosting the backend stack is provided by Top.Host [redacted] services. To ensure better compatibility, it is essential to utilize all tools provided by the host when developing the backend interface. Moreover, it is advisable to design all structures in a way that allows for easy translation to another host mechanism in the future, as a good practice for future scalability and flexibility.

Database schema: To store the data collected from the respective weather stations, a relational database must be utilized. The database should be designed to host tables that efficiently and accurately represent time-series data recordings. SQL is employed for creating the database schema to ensure proper organization and management of the data.

Subroutines: The backend is composed of several sub-routines and sub-programs, each tailored to perform specific tasks. PHP scripts are employed to create these structures, offering simple, straightforward, and fast implementations.

Weather data packet handler: The data transmitted by each of the weather stations in the network is directed to one of the hosted data handler mechanisms. This subroutine must be capable of reading the transmitted data, establishing a connection to the database, and storing the data in the appropriate fields.

Telemetry data packet handler: The telemetry data transmitted by each weather station in the network is directed to a different data handler mechanism. This subroutine must have the capability to store the few most recent telemetry packets received by each station in separate, easily accessible files.

Command transmitter: To remotely control some actions of each weather station, a backend sub-routine is utilized. This subroutine must be capable of formatting the user commands into packets with a specific structure, ensuring easy readability by the stations.

Command packets: The command packets must contain the following information: station ID, command ID, and argument. The command structure must be implemented in a way that allows easy distinction when received by the stations. To achieve this, the command packet must include symbols that indicate the initiation and termination of the packet. Some symbols that can be utilized for this purpose are *, ^, <, $, #, @, ~. Importantly, these symbols should be used exclusively for telecommand packets, enclosing the command data within them.

Command receiver: The commands are executed by the motherboard's controller but received by the network adapter. The controller must be programmed not only for transmitting data to the adapter but also for reading from it. The commands are read and executed only when the controller is in the IDLE state.

Commands: The controller must be capable of reading and executing several commands, such as the mode definition which refers to the recording and transmission time interval, system reset, sensor shutdown, and sensor power on.

Multiple station access: The backend must be capable of efficiently managing multiple station access at the same time. Ensuring that different stations transmitting data simultaneously does not lead to any data loss or database access conflicts is crucial for the system's integrity and reliability.

Backup: The backend must incorporate an efficient data backup mechanism to address malfunctions effectively. The backup file should contain all the data stored in the database and must be readily downloadable. This ensures data integrity and facilitates easy recovery in case of any unforeseen issues.

Undependability: The data storage mechanisms must be designed, developed, and operated independently from the various endpoint user interfaces and platforms. This ensures a modular and flexible approach, allowing seamless integration and compatibility with different user interfaces and platforms.

Low level API: *A set of low-level programs must be developed to manage the interface between the user and the backend. This low-level API should include sub-routines for accessing the database to provide daily data, create custom datasets, and generate statistics. Additionally, the low-level API must be capable of formatting and transmitting commands to the backend handlers, as well as reading back telemetry data. This comprehensive API will enhance the efficiency and effectiveness of the user-backend interaction.*

Methodology

Directories

Figure 1 illustrates the directory hierarchy concerning the backend segment of the weather station network data management system. The route URL is [redacted], which currently extends to the station directory. Within this directory, there are two main subroutines: *store_data.php*, responsible for handling weather data packets, and *telemetry.php*, which manages telemetry data packets.

Additionally, this directory contains three more sub-directories: *telemetry_files*, *backup*, and *command*. The *telemetry_files* directory is dedicated to telemetry files created for each station, the *backup* directory holds the backup file, and the *command* directory contains routines and command files for each station in the network. The red arrows in the figure represent the data flow from source to destination.

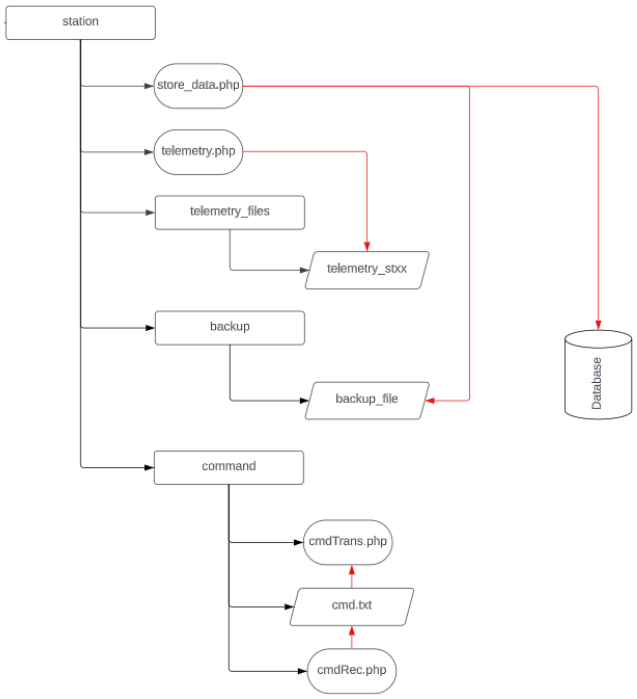

Figure 1 – The backend directory diagram graph.

Figure 2 illustrates the directory hierarchy associated with the low-level API backend data management system. Building upon the previously mentioned route URL [redacted], the hierarchy extends further to the *api* directory. Inside this directory, is contained the *data* directory, which houses the *stats.php*, *dataset.php*, and *daily.php* subroutines responsible for creating the interface between the user API and the backend.

The *stats.php* program, upon being called, accesses the database, and generates a set of statistics on a daily basis for the recorded weather data from the respective weather station. The generated values include the minimum, maximum, and average values for temperature, humidity, pressure, wind speed, and rainfall, as well as the total rainfall integral.

When the *dataset.php* program is called, it accesses the database and generates a dataset between two input dates for the respective weather station. If there is no data available, an empty list is returned to the caller.

As for *daily.php*, whenever it is called, it generates a dataset containing the daily recorded data. If no data is available, an empty list is returned to the caller. This function can be used with additional logic to provide real-time monitoring. Every time a new recording is stored on the database the *daily.php* script updates the files at hold.

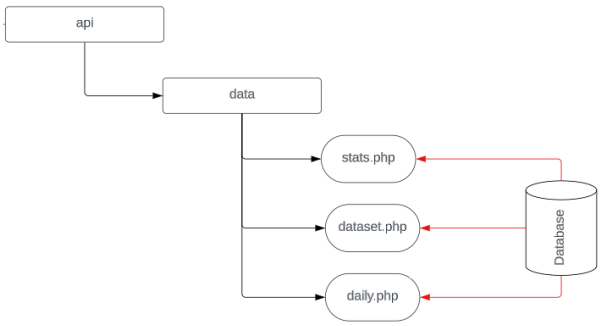

Figure 2 *– The low-level API related backend directory hierarchy*

Database Schema

A database schema is a fundamental component of any database system. It serves as a blueprint for the database system and defines the structure, the organization, and the relationships of the data stored in a database. Essentially, a database schema outlines the logical and physical design of the database, providing a clear representation of how the data is organized and stored.

Tables are the primary building blocks of a database schema. Each column of the table represents an attribute or characteristic of an entity, and each row represents a specific instance or record of that entity. Overall, a table describes an entity. Columns in a table are defined with specific data types. Data types ensure that data is stored in a consistent and structured manner, enabling efficient storage and retrieval of information.

The schema of the database system discussed in this project consists solely of one table. This table holds all the weather recordings that are transmitted by each of the stations. The database is constructed within the Top.Host environment utilizing the InnoDB database engine and the phpMyAdmin management tools. The general database information are given in the following Table 1**.** Table 2 respectively presents the columns of the table *weather_data* that holds this database.

Table 1 *– Database schema*

Database Name	[redacted]
User Name	[redacted]
Password	[redacted]
Tables	weather_data

Table 2 *– The weather_data table features*

Features	Datatype	Description
Station ID	Varchar 4	The unique station identifier. Is of the form *stxx,* where xx refer to a number from 00 to 99
Date	Date	The sample capture date in UTM. Is of the form YYYY-MM-DD
Time	Time	The sample capture time in UTM. Is of the form hh:mm:ss in a full 24 hour cycle mode.
Temperature	Float 64	Captured air temperature in Celsius
Humidity	Float 64	Captured air relative humidity in percentage
Pressure	Float 64	Captured barometric pressure in kPa
Wind Speed	Float 64	Captured wind speed in m/s
Wind Direction	Float 64	Captured wind direction in degrees with reference from North
Rainfall	Float 64	Captured rainfall rate in mm/s

Data Packet Handlers

The recorded weather data is transmitted from the stations to the host server via the application layer using an HTTP post on *http[redacted]station/store_data.php?*. The *store_data.php* subroutine receives the transmitted data packets, sets a timestamp, writes these values to the backup file, and then attempts to establish a connection with the database to insert the data into the *weather_data* table in the respective slots. In case the connection with the database is unsuccessful, the data will still be recorded in the backup file and not lost, and an error indicator flag will be returned, indicating an invalid SQL transaction. However, there is not yet designed any mechanism for updating the database in such cases. This is done manually by the administrator.

The transmitted telemetry data from the stations to the host server also utilizes the HTTP application layer protocol by posting the data on *http:[redacted]station/telemetry.php?*. Unlike weather data, telemetry data packets are not stored in a mass storage structure like a database in a dedicated table. Instead, only the few most recent telemetry packets (currently 1) are written into an appropriate file. The file is named *telemetry_stxx*, where xx refers to a number between 00 to 99, indicating a unique identifier for each station present on the network. The content of the file follows a CSV format and includes the date and time, internal temperature, bus voltage, bus current, solar panel voltage, heartbeat, and operation mode of the station.

Telecommands

Telecommands are an essential feature that was not present in the previous version of the weather station network. They involve the transmission of small data packets to the stations for controlling internal operations. Telecommands, along with telemetry, play a crucial role in the effective station operation and run-time debugging. Telecommands are represented in the form of *$command id, argument$.* Table 3 presents the command ids and the corresponding arguments used for controlling various parameters of the station. Each station is programmed to receive telecommands from a specific file related to each station separately.

Table 3 *– The set of telecommands*

Command ID	Argument	Description
0	0	Reset the system and set the operation to default
1	0	NORMAL MODE operation. This is used for 3min interval between each sensor reading.

1	1	FAST MODE operation. This is used for 5 min intervals between each sensor reading
1	2	SLOW MODE operation. This is used for 10 min intervals between each sensor reading
1	3	POWER SAVING operation. This is used for 30 min intervals between each sensor reading.
2	0	Shutdown the sensor and data transmission. The station freezes the sensor reading and data/ telemetry transmission and waits in only read mode. Each reading is done every 30 minutes.
2	1	Exit the shutdown mode. The station exits the shutdown mode and initiates the system for reading again data and transmitting the recordings with the last modified parameters.

Backup Mechanism

The backup mechanism implemented relies solely on recording the weather data in both the respective database and the backup file, providing dual data redundancy. The backup file can be easily accessed through the path *http:/[redacted]station/backup/backup_file*. However, the current version does not include any mechanisms for automatic database restoration in case of such data losses, leaving the task to the administrator.

Verification

To thoroughly assess the backend system, a virtual simulator is created. This application simulates the behavior of multiple weather stations within a virtual environment, providing the flexibility to test and observe responses under diverse scenarios, which may not be easily achievable with physical units.

The simulator is built with a straightforward program that generates parallel threads, each representing station entities, effectively constructing a potential network. Within this virtual environment, the threads can transmit randomly generated data through both the data and telemetry links, while also processing telecommands from the user controller and executing the corresponding actions, as the physical devices are intended to.

The verification procedure encompasses tests to evaluate the maximum traffic capacity, measure the delay between requests and responses, and assess the feasibility of the current architecture. The results obtained from the verification process are presented in Table 4, providing valuable insights into the system's performance and reliability.

It is important to note that for testing the performance of the aforementioned transactions a python script is employed which means an additional latency is added because of the nature of the executable program.

Table 4 *– Verification results*

Maximum Number of Stations in Parallel	99
Average Response Delay – Telemetry	255.9
Average Response Delay – Dataset	334.3
Average Response Delay – Daily Data	100.6
Average Response Delay – Statistics	450.8
Average Response Delay – Command	70.9